# Automatic Music Classification with jMIR

Cory McKay

Music Technology Area
Department of Music Research
Schulich School of Music
McGill University, Montreal

Submitted January 2010

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

# Abstract

Automatic music classification is a wide-ranging and multidisciplinary area of inquiry that offers significant benefits from both academic and commercial perspectives. This dissertation focuses on the development of jMIR, a suite of powerful, flexible, accessible and original software tools that can be used to design, share and apply a wide range of automatic music classification technologies.

jMIR permits users to extract meaningful information from audio recordings, symbolic musical representations and cultural information available on the Internet; to use machine learning technologies to automatically build classification models; to automatically collect profiling statistics and detect metadata errors in musical collections; to perform experiments on large, stylistically diverse and well-labelled collections of music in both audio and symbolic formats; and to store and distribute information that is essential to automatic music classification in expressive and flexible standardised file formats.

In order to have as diverse a range of applications as possible, care was taken to avoid tying jMIR to any particular types of music classification. Rather, it is designed to be a general-purpose toolkit that can be applied to arbitrary types of music classification. Each of the jMIR components is also designed to be accessible not only by users with a high degree of expertise in computer-based research technologies, but also by researchers with valuable musical expertise, but perhaps less of a background in computational research. Moreover, although the jMIR software can certainly be used as a set of ready-to-use tools for solving music classification problems directly, it is also designed to serve as an open-source platform for developing and testing original algorithms.

This dissertation also describes several experiments that were performed with jMIR. These experiments were intended not only to verify the effectiveness of the software, but also to investigate the utility of combining information from different types of musical data, an approach with the potential to significantly advance the performance of automatic music classification in general.

## Sommaire

La classification automatique de la musique est un vaste domaine de recherche, multidisciplinaire par nature, et qui donne lieu à des avancées significatives tant du point de vue scientifique que du point de vue des applications commerciales. La présente dissertation s'articule autour d'un thème central qui est la conception et le développement de jMIR, une suite originale de logiciels qui sont à la fois puissants, flexibles et accessibles. Ces outils peuvent être utilisés pour concevoir, partager et appliquer une grande variété de technologies de classification automatique de la musique.

jMIR permet à l'utilisateur d'extraire de l'information significative des enregistrements audio, des représentations musicales symboliques et des informations culturelles disponibles sur l'Internet; de se servir des technologies d'apprentissage automatiques afin de construire des modèles de classification, de compiler automatiquement des profils statistiques, de détecter les erreurs de métadonnées dans les collections de pièces musicales; d'effectuer des expériences sur de larges corpus de musique; et enfin de répertorier et distribuer de l'information essentielle à la classification automatique de la musique sous des formats expressifs, standardisés et flexibles.

jMIR est plutôt conçu comme une boîte à outils d'usage général pouvant être appliquée à n'importe quel type de classification de la musique. Chaque élément de jMIR est aussi conçu pour être accessible autant à des utilisateurs experts en technologies de l'information qu'à des, chercheurs dont l'expertise musicale précieuse ne serait pas doublée d'une formation en technologies de l'information. Bien que jMIR intègre un jeu d'outils prêts à résoudre directement des problèmes de classification de la musique, il permet également de s'en servir comme d'une plate-forme ouverte de développement logiciel ainsi que de validation de nouveaux algorithmes.

Enfin, la présente dissertation décrit plusieurs expériences réalisées au moyen de jMIR. Ces expériences avaient non seulement pour but de vérifier la pertinence de cet environnement logiciel, mais également d'investiguer les bénéfices qu'apportent l'utilisation conjointe d'informations musicales de nature diverse, cette dernière approche ayant le potentiel de faire avancer de façon significative la classification automatique de la musique en tant que discipline de recherche.

# Table of Contents

# List of Figures

# List of Tables

## Acknowledgements

First and foremost, I would like to thank Professor Ichiro Fujinaga for his invaluable ideas, insights and advice regarding all aspects of this research. I would also like to thank those with whom I have co-authored publications relating to various parts of this research, including John Ashley Burgoyne, Professor Philippe Depalle, Rebecca Fiebrink, Beinan Li, Daniel McEnnis, Elliot Sinyor and Jessica Thompson. Daniel McEnnis in particular played an essential role in co-designing and co-implementing jAudio, and was entirely responsible for OMEN. The valuable work of Jessica Thompson and John Ashley Burgoyne on the implementation of ACE 2.0 and ACE XML 2.0, respectively, have been essential in helping jMIR continue to grow beyond the original scope of this dissertation.

I would also like to thank Mark Zadel for facilitating the development of jWebMiner by providing the implementation details of his own web mining research, Professor Douglas Eck for providing access to his lab's audio research database and the staff of the Marvin Duchow Music Library for their help in providing CDs to be included in the Codaich audio dataset. Thanks also to Madeleine McKay for designing the jMIR logo, and to Professor Philippe Depalle and Donald McKay for their help with the French version of this dissertation's abstract. I would also like to thank the faculty, staff and students of the McGill University Music Technology Area, who have provided an ideal environment for discussion, collaboration, creativity, research and learning.

This research would not have been possible without the generous financial support of the *Social Sciences and Humanities Research Council of Canada* (SSHRC). I would also like to thank the *Fonds Québécois de la recherche sur la société et la culture* (FQRSC) for their financial support during my master's research, which served as the basic foundation upon which this dissertation was built. Thanks also to the *McGill Alma Mater Fund* and *The Centre for Interdisciplinary Research in Music Media and Technology* (CIRMMT) for providing travel funding, and thanks to the *Canada Foundation for Innovation* for providing the funds to purchase computing equipment. Special thanks also to the *Andrew W. Mellon Foundation* for funding continuing jMIR development as part of the *Networked Environment for Music Analysis* (NEMA) project.

Finally, above all, I am grateful for the invaluable love and support of Deanna Regan Toews, Madeleine McKay, Erin McKay, Donald McKay and Thérèse d'Amour.

# 1. Introduction and background

## *1.1 General overview*

### 1.1.1 Introduction

Automatic music classification is a process that consists of computers automatically mapping musical information of some kind to categories of interest. A few examples of relatively low-level, although certainly far from trivial, types of music classification include the identification of particular instruments, pitches and rhythmic patterns in audio signals. On a somewhat higher level, one might identify chords, chord progressions, melodies, structural segmentations and so on. It is also possible to design systems that can map pieces of music to still higher-level categories, such as in the cases of classification by composer, performer, genre, style, mood, listening scenario or preference profile. These are just a few examples of the many possible types of music classification that can be performed, of course, but they do help to illustrate the variety and scope of applications that can be addressed by automatic music classification and the range of benefits that can be reaped from the successful implementation of accurate and consistent classification software.

Research on automatic music classification has significant value from both academic and commercial perspectives. Not only can the results of various types of classification be very useful in and of themselves, such as in helping to label and organize large music collections, but automatic music classification can also play an essential role in accomplishing component parts of many large and wide-ranging tasks. For example, a successful system for automatically transcribing audio signals to symbolic musical scores would likely consist of numerous sub-systems that each perform automatic classification in order to solve related sub-problems, such as voice segmentation, note onset detection, pitch tracking, tempo induction and so on.

In practice, many types of music classification are very difficult to perform consistently and accurately. This is often the case both for human classifiers and for computers programmed with pre-defined classification heuristics. In cases such as classification by genre or mood, for example, there are very few precise, clear and consistent rules delineating the musical qualities and characteristic of each category. Indeed, it can sometimes be difficult even to come up with a generally accepted set of categories to categorize pieces of music into.

Fortunately, automated pattern recognition techniques provide a very useful empirical way of dealing with such issues by learning to recognize statistical regularities in collections of music, and thus effectively building categorization models that can map musical data to appropriate categories. Tools that can allow researchers to quickly, easily and effectively apply such approaches could prove to be very valuable to researchers involved in diverse areas of musical inquiry.

The overall goal of this dissertation is thus the provision of powerful, flexible, accessible and original tools that can be used to develop, share and apply techniques associated with automatic music classification. A suite of software tools called jMIR has been produced in fulfillment of this goal. The jMIR components enable users to extract meaningful information from audio recordings, symbolic musical representations and cultural information relating to music that is available on the Internet; to automatically build classification models capable of mapping music to categories that are meaningful for particular research applications; to automatically collect profiling statistics and detect metadata errors in musical collections; to perform experiments on a large, stylistically diverse and well-labelled collections of music in both audio and symbolic formats; and to store and distribute information that is essential to automatic music classification in expressive and flexible standardised file formats.

In order to have as diverse a range of applications as possible, jMIR is very purposefully not tied to any particular type of music classification, but is rather designed to be a general-purpose toolkit that can be applied to arbitrary types of music classification. Each of the jMIR components is also designed to be accessible not only to users with a high degree of expertise in computer-based research technologies, such as researchers in electrical engineering and machine learning, but also to researchers with very valuable musical expertise but less of a background in computational research, such as some musicologists, music theorists and music psychologists.

It is hoped that providing music researchers with sophisticated empirical tools for studying music in traditionally unorthodox ways will help them to glean important new insights into music. An important advantage of software such as jMIR is that it allows researchers such as musicologists and music theorists to perform empirical research on a much wider range of art, folk and popular musics of the world than is typically possible using the type of heuristics-based computational approaches traditionally used in music informatics. Furthermore, these tools also facilitate the automated extraction and processing of information from audio and cultural information sources, an important break from the symbolic emphasis of traditional music informatics. It is also hoped that the insights and needs made apparent by these types of research will in turn help to expand the scope of current research in automatic music classification.

It is important to note that jMIR is intended as more than just a set of to ready-to-use tools that can be used to solve classification problems directly, although this is certainly an important intended use of jMIR. The software is also designed to serve as an open-source standardized platform for developing and testing original algorithms that can be used to diversify and improve automatic music classification in the future. It is hoped that this will help to facilitate the rapid spread of new approaches as they are developed, reduce redundant duplication of work, facilitate the integration of approaches contributed by researchers in diverse fields, help to promote inter-institutional research collaboration in general and aid in the development of technologies allowing researchers to exceed the performance of existing automatic classification systems.

The jMIR development process involved the design of novel data extraction and processing algorithms; the collection, standardization and implementation of existing algorithms; the design and implementation of original infrastructure and interfaces for performing and researching diverse types of automatic music classification; and the analysis, design and refinement of new and existing research methodologies. The jMIR components were also used in several experiments, as described in Chapter 9. These experiments were intended not only to verify the effectiveness of jMIR, but also to investigate the utility of combining information from different types of musical data sources.

Each of the jMIR components may be used independently, or they may be used together as an integrated whole, depending on the needs of individual users. This independence is reflected in the structure of this dissertation, with the result that each of the jMIR components is presented in its own chapter, from Chapter 3 to Chapter 8. The multidisciplinary nature of the jMIR project involved elements of many disciplines, including digital signal processing, music theory, web-based data mining, machine learning, musicology and knowledge management. Each of the chapters therefore provides background information from the fields that are the most relevant to its particular jMIR component.

Chapter 2 provides a brief review of psychological and musicological research on human music classification. Although this chapter does not deal directly with any specific jMIR components, it is included in this dissertation because many automatic music classification applications essentially involve the simulation and prediction of the types of classifications produced by humans. It is helpful to have some fundamental insights into how humans perform such classifications in order to accomplish this, even if the actual processes used by computers to arrive at corresponding classifications may be entirely different.

This first chapter is intended to provide an overview of and context for the jMIR project, as well as to present appropriate background information relevant to the project as a whole. It also provides more details on many of the issues raised in this introductory Section 1.1.1. Section 1.1.2 introduces certain basic vocabulary and fundamental concepts that are essential to understanding automatic music classification. Section 1.1.3 then provides a brief overview of each of the jMIR components, and Section 1.2 details the chapter by chapter structure of this dissertation.

Section 1.3 provides a general introduction to music information retrieval, a field of which automatic music classification is a sub-discipline, and then continues on to provide a relatively detailed discussion of the nature, types, applications and advantages of automatic music classification. The differences and similarities between music classification and similarity measurement are also discussed, an issue that is revisited in Chapter 2. Of particular significance, Sections 1.3.6 and 1.3.7 summarize other systems that have been developed for performing automatic music classification research.

Section 1.4 discusses the overall design principles that motivated the development of jMIR. This section also provides an overview of how the jMIR components as a whole fulfill these design principles.

Section 1.5 reviews some of the essential theoretical and research contributions of this dissertation. This is done in order to highlight them against the backdrop of the engineering aspects of the jMIR project.

### 1.1.2 Essential concepts in automatic music classification

It is useful to briefly explain the core procedures and terminology associated with automatic music classification before proceeding to describe jMIR in more detail, as a familiarity with these essential aspects of automatic music classification will greatly facilitate this description. The information provided in this sub-section is very much a high-level overview, and more detailed information is available in Section 1.3 and, to an even greater extent, in the individual chapters of this dissertation that pertain to the various aspects of automatic music classification.

Most automatic music classification systems make use of *machine learning* to at least some extent, and it is a central aspect of many systems. Machine learning involves using computer algorithms to automatically build, or learn, a model that can be used to perform some task. From the perspective of automatic music classification, this model allows different entities of interest, called *instances,* to be classified by the system into the categories of interest, called *classes*. Classes could be, to give just a very few examples, names of composers, types of musical genres or musical pitches. These classes may be treated as unrelated entities, or they may be organized into a *class ontology.* Instances can be many different things, depending on the application domain, such as musical pieces, short musical excerpts, the collected works of a musician or composer, etc.

The essential advantage of machine learning is that it enables computers to automatically learn solutions to very difficult problems, even when there is no pre-existing knowledge of the solution to a given problem. Machine learning also eliminates the difficult and time-consuming need to explicitly formalize and implement potentially very complex solutions even when they are known. Machine learning can thus provide an excellent solution for many music classification problems, since effective classification models are both often poorly understood and very complex.

There are many varieties of machine learning, but in general a class of techniques called *supervised learning* tend to be the most appropriate for most music information retrieval problems.[1] Supervised learning requires first acquiring a set of instances, called *ground-truth data,* that have been labelled with model class labels that are known to be correct. The machine learning algorithm that is being used is then trained on this data, thereby automatically learning to associate particular characteristics of ground-truth instances with the classes that they are labelled with. This is, at least from a very high-level perspective, similar in concept to how a human might be trained to classify instances by observing exemplars that he or she is told belong to particular categories.

If all goes well, the classification model built by the machine learning algorithm will have learned associations between the characteristics of the ground-truth training instances and their labelled classes that may be generalized so that it can classify new unlabelled instances correctly based on their own characteristics. Ground-truth data is typically divided into training, testing and validation sets in order to attempt to verify that such generalization has occurred.

The labelling of ground-truth data with correct class labels is essential to the training of successful models. The association of ground-truth instances with various other kinds of data can also be helpful as well. The term *metadata,* which can be defined as data about data, is used to refer to any kind of data that instances are annotated with. Unfortunately, metadata annotations mined from the Internet tend to be inconsistent and error prone, so it is often necessary to correct them before they can be used in machine learning problems.

There are three primary types of musical information that traditionally serve as instances in automatic music classification:

- **Audio data:** Digital representations of physical audio signals. These are typically stored in formats such as MP3s, WAVs and FLACs.
- **Symbolic musical representations:** Representations of sound based on abstract symbols that are musically meaningful, such as the music notation used in scores. File formats such as MIDI, OSC, Music XML and Humdrum store symbolic data.

[1] The reasons for this, alternative approaches and exceptions to this general rule of thumb are discussed in Chapter 6.

- **Cultural data:** Information that is related to the music of interest, but is not a direct representation, abstract or otherwise, of the actual sound comprising the music. The Internet provides the most commonly exploited source of cultural data, as it makes available resources such as edited metadata repositories, unedited listener tags, playlists and searchable web pages in general. Other potential sources of cultural data include expert writings such as album reviews, statistics such as sales statistics, surveys, the results of psychological experiments and images of album art.

As noted above, machine learning algorithms must be provided with characteristics of instances that encapsulate sufficient information to learn and apply classification models. Such measurable characteristic pieces of information that can be extracted from instances and provided as the percepts of machine learning algorithms are called *features,* and the particular choice of features to extract from instances can be essential to the success of the learned classification model. The use of too many features can pose problems, as described in Chapter 6, so it is often necessary to apply *dimensionality reduction* algorithms in order to effectively reduce the size of large feature sets while minimizing the consequent loss of relevant information.

It can be useful to group features into the three following classes:

- **Low-level content-based features:** Spectral or time-domain information extracted directly from audio signals. Most features of this type do not provide information that seems intuitively musical, but they can have significant discriminating power when processed by computers, and they can also sometimes have psychoacoustic significance. Mel-Frequency Cepstral Coefficients, Zero Crossings and Signal RMS are examples of such features.

- **High-level content-based features:** Information extracted from symbolic or audio data that is formulated in such a way that it is meaningful to musically trained humans. Measures of the amount of chromatic motion, the amount of rubato, the instruments present and information associated with song lyrics are examples of such features.

- **Cultural features:** Sociocultural information outside the scope of the musical content itself. In practice, the most commonly used cultural features tend to consist of statistics that can be automatically mined from the web, such as playlist cooccurrences of various musical tracks or sales statistics.

Not coincidentally, these feature types roughly coincide to the three types of data described above.

The basic procedures involved in a typical automatic music classification task are summarized in Figure 1.1.

### 1.1.3 Overview of the jMIR components

As noted above in Section 1.1.1, jMIR is a suite of software tools and resources developed for use in research associated with automatic music classification. jMIR is designed not only to be used directly for performing automatic music classification-related tasks like feature extraction, pattern recognition and experimental validation and investigation, but also as a platform for developing and sharing new technologies, such as new features or machine learning techniques.

Each of the various jMIR components is designed to address a particular task or set of tasks related to automatic music classification:

- **ACE XML:** A set of standardized file formats that can be used to store information such as extracted feature values, feature metadata, class labels associated with instances, miscellaneous instance metadata and class ontologies. These file formats allow information to be transferred between the jMIR software components,[2] and are also proposed as a general standard for use in storing and communicating data associated with automatic music classification beyond the context of jMIR.

[2] For the sake of general compatibility the jMIR software components can also read and write Weka ARFF (Witten and Frank 2005), the current de facto file format standard in the MIR community, but ACE XML is strongly recommended as an alternative due to its many relative advantages, as discussed in Chapter 7.

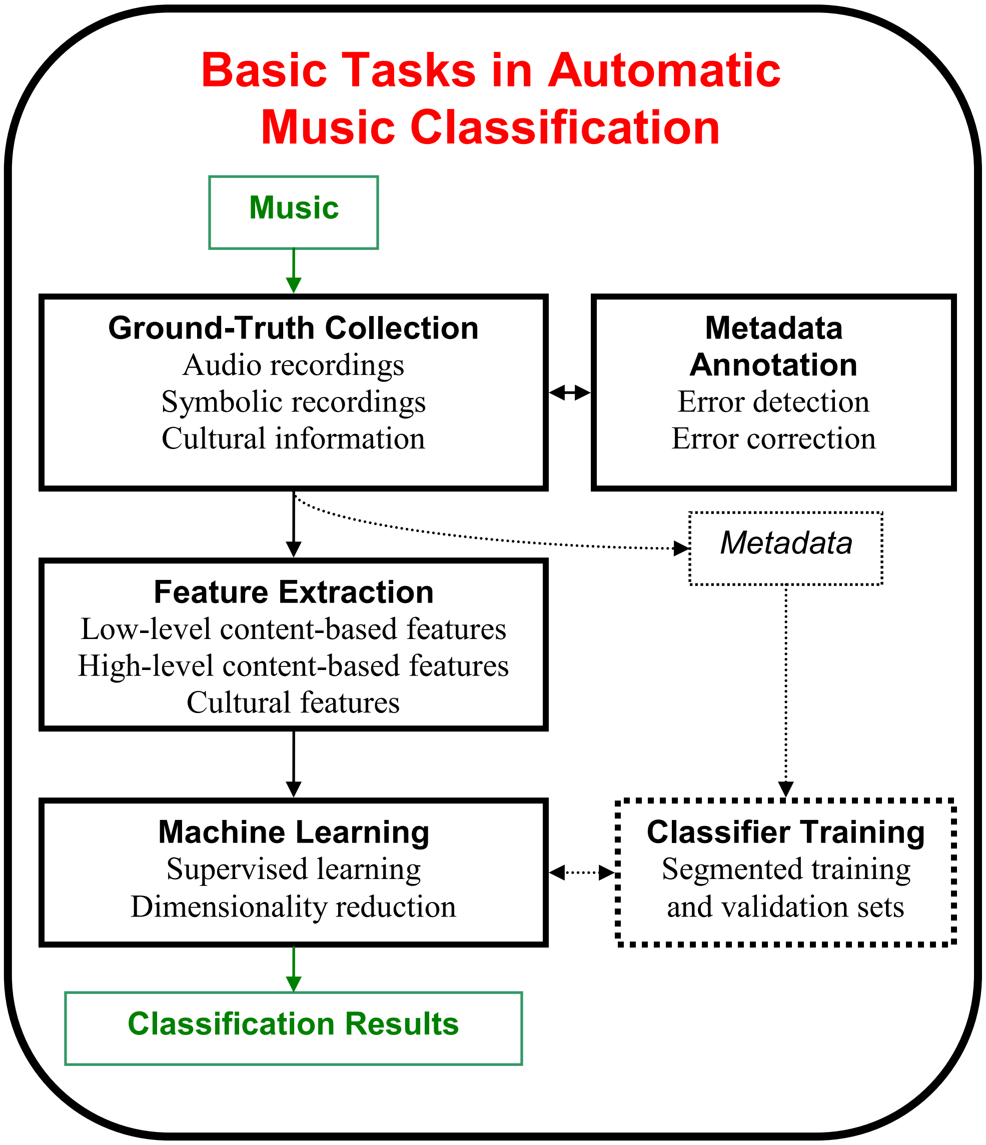


**Figure 1.1:** A workflow of the tasks that must be performed in a typical automatic music classification scenario. The first step involves collecting the ground-truth data. This collected musical data must then be labelled with metadata or, if the metadata is already present, then it must be validated. Features must then be extracted from the ground-truth data. These extracted features must be capable of sufficiently characterizing the differences between the classes that the ground-truth instances are to be classified into. The features are then input to the machine learning algorithm(s) that are to classify the music, typically using training and validation subsets of the ground-truth data. Once the classifiers are trained and validated, new unlabelled instances can then have their features input to the trained classification model, which in turn will output predicted class labels.

- **jAudio:** An audio feature extractor that includes implementations of 26 core features, including both features proven in MIR research and more experimental perceptually motivated features. jAudio places an even greater emphasis on extensibility than the other jMIR components, and includes implementations of *metafeatures* and *aggregators* that can be used to automatically generate many more features from core features (e.g., standard deviation, derivative, etc.). A number of tools to facilitate the development and testing of new features are also included, such as time and frequency domain visualization tools; audio recording functionality; test data synthesis; audio file format conversion; and MIDI to audio conversion functionality. Feature extraction parameters such as window size, overlap, downsampling and normalization can also be set by the user.

- **jSymbolic:** An application for extracting features from MIDI files. jSymbolic is packaged with a very large collection of 111 features, many of which are original. A further 42 features are proposed for implementation in the jSymbolic feature library. These features all fall into the broad categories of instrumentation, texture, rhythm, dynamics, pitch statistics, melody and chords.

- **jWebMiner:** A cultural feature extractor that, at its most basic level, operates by using web services to measure relative hit counts for sets of query strings submitted to search engines. jWebMiner then processes these hit counts to calculate feature values denoting classifications or similarity measurements based primarily on co-occurrence. Search results are processed statistically by jWebMiner in a variety of ways in order to improve results. Further processing options include the abilities to specify search string synonyms; to filter out sites containing specified strings; to require that sites contain certain strings in order to be counted; and to weight results from different on-line sources differently. jWebMiner can also automatically access a variety of sources in order to acquire the strings needed to build queries. Such sources include Apple iTunes XML files,[3] delimited text files, ACE XML files and Weka ARFF files (Witten and Frank 2005).

[3] support.apple.com/kb/HT1660

- **ACE:** Meta learning software for experimenting with, selecting and applying pattern recognition algorithms. Given a set of extracted features, ACE experiments with a variety of machine learning algorithms, algorithm parameters, classifier ensemble architectures and dimensionality reduction techniques in order to arrive at a good configuration for the problem at hand. This can be very helpful, as a particular algorithm can be more or less appropriate for a given problem in terms of considerations such as classification accuracy, training speed and classification speed. ACE can also be used simply to build and apply pre-selected classification models if desired. ACE is based upon the well-known Weka (Witten and Frank. 2005) Java pattern recognition library, so new classification algorithms implemented in Weka can be easily incorporated into ACE.

- **jMusicMetaMangaer:** A tool for statistically profiling large musical datasets as well as automatically detecting metadata errors, inconsistencies and redundancies. Such dataset analysis and metadata error checking are essential tasks, as the success of ground-truth training and evaluation data is contingent upon the quality of the musical datasets from which they are drawn and the accuracy of the associated metadata, especially class labels. jMusicMetaManager implements a wide variety of metadata error detection operations, including general edit-distance and word ordering/subset operations as well as customized operations designed to detect specific common errors. A total of 42 different HTML reports can be automatically generated to help profile and publish musical datasets. jMusicMetaManager can parse iTunes XML files and MP3 ID3 tags as well as ACE XML and Weka ARFF files in order to access the metadata that is to be analyzed.

- **Codaich, Bodhidharma MIDI and SAC:** Three different labelled datasets for use in debugging, validating and evaluating automatic music classification technologies and for performing general exploratory computational musical research.
    - Codaich is an audio dataset consisting of over 25,000 MP3 recordings belonging to 55 genres of music. The recordings in Codaich are labelled

    with a variety of metadata that has been carefully checked for errors and inconsistencies both manually and with jMusicMetaManager. Information about Codaich is available in the form of iTunes XML files and jMusicMetaManager reports.
    - The Bodhidharma MIDI dataset is a collection of 950 MIDI recordings belonging to 38 genres.
    - The SAC dataset consists of matched MP3 recordings, MIDI encodings and cultural data for 250 pieces of music belonging to 10 genres of music. This dataset is specifically designed for research comparing the relative utility of features extracted from audio, symbolic and cultural sources of information, both individually and in combination.

- **jMIRUtilities:** A set of tools for performing miscellaneous tasks associated with jMIR. These tools include a GUI for batch-associating class labels with instances, utilities for merging various kinds of information, and utilities for extracting useful information from iTunes XML files or delimited text files.

The jMIR components are designed so that they may be used either together, as an integrated whole, or separately as individual and isolated applications, as the user prefers. The relationships between and the various jMIR components are illustrated in Figure 1.2.

The jMIR software is all open-source and is implemented in Java in order to promote platform independence. Both the source code and compiled user versions of each of the jMIR components may be downloaded for free from jmir.sourceforge.net. This website also includes detailed manuals and other documentation. Further information on each of the jMIR components is also provided in the various chapters of this dissertation, as detailed below in Section 1.2. Each of these chapters also includes references to various jMIR-related publications, and there is also a paper by McKay and Fujinaga (2009b) that summarizes the jMIR software suite as a whole. Section 1.4 also includes an overview of the core design priorities and characteristics of the jMIR project as a whole.

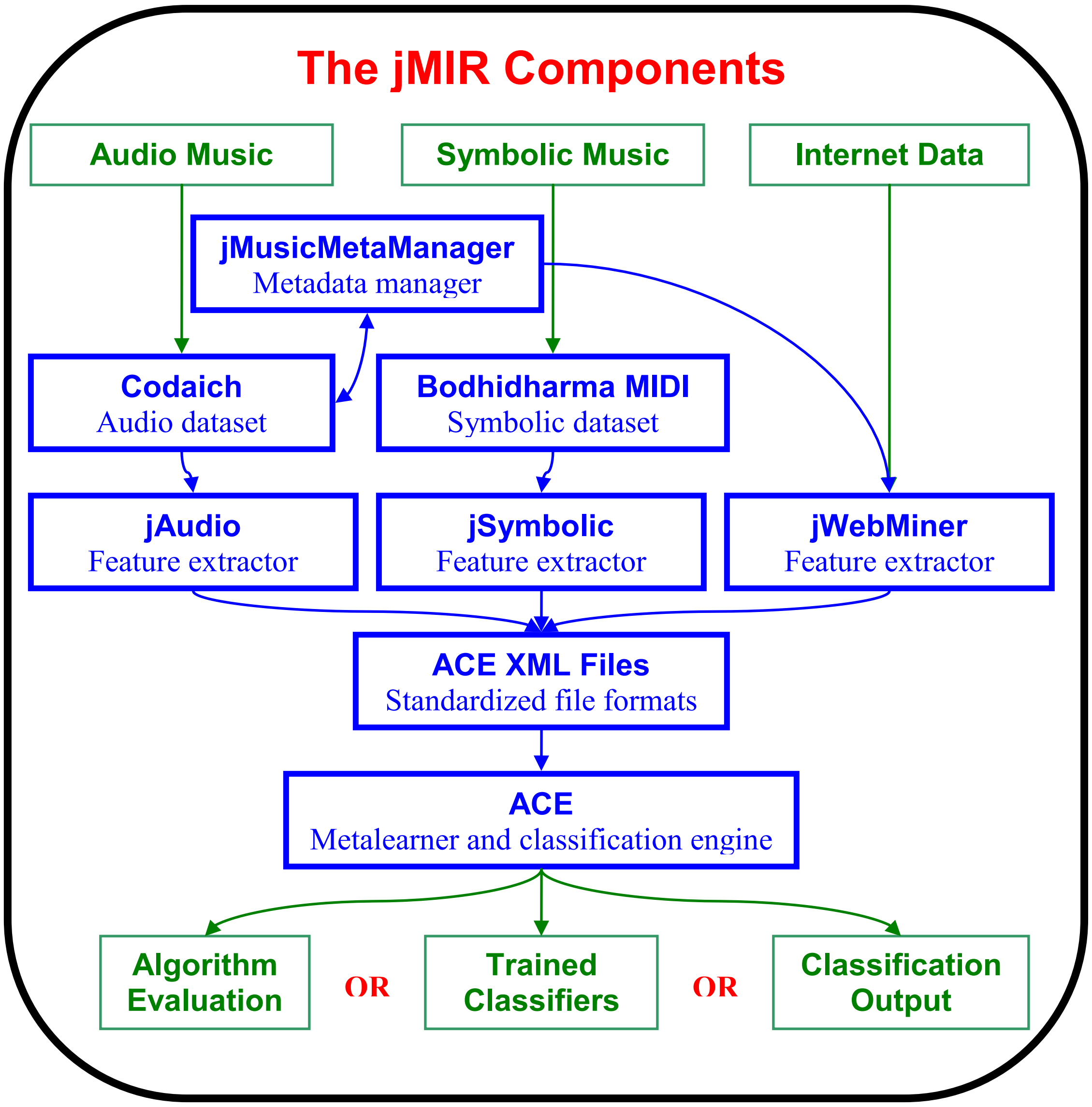


**Figure 1.2:** The relationships between the different components of the jMIR software suite. Audio recordings, symbolic recordings and/or cultural data available on the Internet can all provide input to jMIR. The various components of jMIR can be used to extract features from this data and check the metadata associated with it. Extracted features may also be processed such that the relative appropriateness of various pattern recognition algorithms to the problem are compared, a classification model is trained on the features or predicted class labels are output for instances. It is important to note that, for the sake of legibility and simplicity, this figure only demonstrates how ACE XML files can be used to store extracted feature values, when in fact other information, including feature metadata, instance metadata, class labels associated with instances and class ontologies can also all be stored in ACE XML files.

## *1.2 Structure of this dissertation*

This first chapter provides an overview of the jMIR project and places it in context. Section 1.3 provides relevant background on music information retrieval in general and automatic music classification in particular, as well as an overview of prominent software systems that have been designed to perform or facilitate automatic music classification. Section 1.4 then describes the primary design priorities and corresponding characteristics of jMIR, and Section 1.5 highlights some of the primary research contributions of this dissertation.

Chapter 2 provides insights from the fields of psychology, musicology and music theory relating to how humans classify music and measure musical similarity. Although automatic music classification systems can and have been implemented without explicitly taking such research into account, the position is taken here that an understanding of how humans classify music can help to more accurately simulate their behaviour using computers.

Chapter 3 presents jAudio, the jMIR application that extracts features from audio data. Aside from a detailed description of the software itself, this chapter also provides relatively substantial background information associated with digital audio and audio feature extraction, as one of the goals of jAudio is to encourage and facilitate the development of new and useful features that can be extracted from audio data.

Chapter 4 introduces jSymbolic, the jMIR application that extracts features from symbolic musical data. An essential aspect of jSymbolic is its extensive library of 111 implemented features and 42 further proposed features, as this is by far the most extensive library of symbolic classification features to date. Each of these 153 features is described in this chapter, along with guidelines for designing symbolic features and other associated background information.

Chapter 5 describes jWebMiner, the jMIR application that extracts cultural features from the Internet. Techniques that can be used to extract further cultural features from the Internet are also discussed, as are particular problematic issues to consider.

Chapter 6 presents ACE, the jMIR meta learning application that can be used to train classification models, perform classifications using trained models and perform experiments evaluating and facilitating the selection of classification algorithms that are well suited to individual problems. Although ACE is designed to make powerful machine learning algorithms available to users with little or no background in such techniques, it is also designed to be a tool for experts in machine learning and a platform for applying and evaluating new algorithms. This chapter therefore includes relatively extensive background information on machine learning problems, techniques and algorithms.

Chapter 7 introduces ACE XML, a set of standardized file formats for representing, storing and distributing feature values, feature metadata, instance class labels, instance metadata and class ontologies. ACE XML is the default format for communicating information between the jMIR components, and is also proposed as a standard for more general use. This chapter includes a critical review of alternative approaches to storing information important to music classification, and proposes a set of design principles that should be taken into account when designing or selecting particular file formats for use in MIR-related tasks.

Chapter 8 presents the three datasets that were assembled as part of the jMIR framework, namely the Codaich audio dataset, the Bodhidharma MIDI dataset and the SAC mixed symbolic, audio and cultural dataset. This chapter also introduces jMusicMetaManager, the jMIR application for statistically profiling and detecting metadata errors in large music collections, as well as jMIRUtilites, the jMIR application for performing various miscellaneous tasks. A discussion of approaches and priorities to consider when building, selecting and using ground-truth data collections and associated class ontologies is also included in this chapter.

Chapter 9 describes several experiments that have been performed using jMIR in order to validate it and demonstrate its usefulness. Particular emphasis is placed on two sets of experiments, one involving a winning entry of the jSymbolic feature library[4] in the MIREX Symbolic Genre Classification competition, and the other involving an experimental investigation of the benefits of combining features extracted from matching audio, symbolic and cultural data sources.

Chapter 10 provides a summary of the jMIR project. This chapter also outlines key areas of future development and research. Chapters 3 to 9 also include more detailed

[4] The jSymbolic feature library was submitted as part of the Bodhidharma symbolic genre classifier, the predecessor of jSymbolic and ACE.

ideas for future research that are more specifically related to each of their respective topics.

Finally, the bibliography for this dissertation is found in Chapter 11.

## 1.3 Context and background information

### 1.3.1 Music information retrieval (MIR)

*Music information retrieval,* or *MIR,* is a field of research associated with the computational extraction of information from music, and with making this information accessible to users of different kinds in ways that are useful to them. MIR overlaps with and is enriched by many other disciplines, including but not limited to machine learning, data mining, digital signal processing, music theory, musicology, music psychology, music education, human computer interaction and the library sciences. As is made clear below, not only is automatic music classification an essential component of many aspects of MIR, but it is the central goal of several of its sub-disciplines.

Some particularly common sub-disciplines of MIR, both past and present, include:

- **Score following:** Tracking the position in a known score of an audio rendition of the score. This is sometimes associated with automatic accompaniment systems.
- **Optical music recognition:** The generation of symbolic scores from optical scans of physical scores.
- **Automatic transcription:** Producing a symbolic representation of music from an audio signal. This can involve monophonic audio, which contains only one audio source and is typically relatively easy to transcribe automatically, or it can involve polyphonic audio, which contains more than one audio source and can be very difficult to transcribe.
- **Source segmentation:** Separating out the components of audio signals originating from different sources.
- **Onset detection:** Identifying the beginnings of individual notes.
- **Pitch tracking:** Identifying the pitch of notes.

- **Chord identification:** Segmenting and identifying harmonic units.
- **Key identification:** The identification of the key of a piece including, sometimes, modulations to other keys.
- **Tempo induction:** Identifying the tempo and, sometimes, the meter of an audio signal.
- **Beat tracking:** Identifying and following the beat of an audio signal.
- **Instrument identification:** Identifying the type of instrument or instruments contained in an audio signal.
- **Song segmentation:** The identification of the points in time where musical pieces start or end in a continual audio signal.
- **Structural analysis:** The breaking apart of a musical piece into structurally discrete segments. This can involve high-level structures (e.g., verses and choruses) as well as lower-level structures (e.g., melodic segmentation).
- **Automatic musical analysis:** The automatic analysis of music using high-level musical models.
- **Fingerprinting:** The identification of musical pieces by matching data extracted from unknown songs to data extracted from known songs. These extracted "fingerprints" should ideally be invariant to acoustic listening environment and audio compression.
- **Artist identification:** The identification of musical performers.
- **Composer identification:** The identification of composers.
- **Genre classification:** The classification of music based on musical genre, as well as the study of appropriate genre ontologies to use.
- **Mood classification:** The classification of music based on mood or emotion.
- **Tag prediction:** Automatic textual annotation of music with tags of various kinds that listeners would likely use or find helpful.

- **Digital Rights Management (DRM):** Various kinds of digital protection applied to audio files for the purpose of listeners' ability to copy and/or transmit the music. *Watermarking,* or the embedding of traceable perceptually hidden data in audio signals, has been a particular area of past MIR interest.
- **Cover song detection:** The identification of different versions of the same musical piece. This sub-discipline is sometimes also associated with investigations of potential copyright violations.
- **Hit prediction:** The identification of musical pieces that are likely to become commercially successful.
- **Music recommendation:** Automatic recommendation of music that individual listeners are likely to enjoy based on their personal tastes.
- **Playlist generation:** Automatic formulation of playlists consisting of pieces that are considered to be particularly suitable to be played together. Such playlists are often, but not always, associated with particular listening scenarios, such as while reading, working out, driving, performing chores, etc. The particular ordering of the pieces can be an important consideration, not just the selection of the pieces themselves.
- **Musical similarity:** Strongly related to a number of the tasks listed above, such as music recommendation, playlist generation and hit prediction, the measurement of musical similarity involves the study and prediction of perceived similarity between different pieces of music along various dimensions.
- **Feature extraction:** The design of features that can be extracted from musical data and gainfully used for various useful purposes, such as classification, similarity measurement or visualisation.
- **Internet-based data mining:** The extraction of useful music-related information from the Internet. There are many possible sources of data, with web services offered by various organizations serving as particularly useful resources.

- **Music archives, libraries and digital collections:** Issues relating to collecting, annotating, organizing, archiving and making musical data accessible.
- **Standards:** Standardized formats and methodologies for storing, annotating and making music-related data available.
- **Metadata collection and annotation:** Methodologies for acquiring reliable metadata, correcting faulty metadata, determining the types of metadata that are appropriate to use for different purposes and finding good ways to store metadata and make it available. There has been a particular focus on the semantic web in recent years.
- **Musical querying:** This can consist of traditional symbolic or textual queries, or it can involve query by humming or query by tapping, which involve, respectively, searching music collections based on sung or tapped queries. Automatic query correction is sometimes necessary.
- **Visualization and interfaces:** This can involve anything from forming visual representations of aspects of individual pieces to visualizing large music collections as a whole. There are many possible reasons for doing so, such as annotating music, editing music or facilitating music browsing.
- **Music perception and cognition:** Studies and models of how, for example, listeners evaluate musical similarity, classify music, organize music and perceive various musical parameters.
- **User behaviour:** Studies of how individuals consume and would like to consume music.
- **Sociology, economics and law of digital music:** Issues relating to the production, marketing, distribution and consumption of music. This is often associated with intellectual property rights.
- **MIR evaluation and information exchange:** Methodologies for comparatively evaluating the performance of different MIR systems and algorithms, as well as for sharing data and algorithms efficiently and effectively.

Foote (1999a), Byrd and Crawford (2001), Downie (2003), Fingerhut (2004) and Casey et al. (2008) have each written excellent overviews of music information retrieval research. The *International Society for Music Information Retrieval Conference (ISMIR),*[5] an annual conference that focuses specifically on MIR and publishes its proceedings online free of charge, is an excellent source of cutting edge MIR research. Also of particular interest, the *Music Information Retrieval Evaluation eXchange (MIREX)*[6] is an annual competition associated with ISMIR where various approaches and algorithms are independently compared using the same data. Due to its highly multidisciplinary character, MIR research is also published in a wide variety of other conferences and journals.

### 1.3.2 Advantages of automatic music classification

As noted above, automatic music classification involves using computers to automatically label musical instances with appropriate class labels of some kind. Automatic music classification has many advantages relative to manual human classification. To begin with, computers can perform classifications much faster than humans. A human must listen to music in real-time if she or he is to classify it, whereas a powerful computer can effectively "listen" to music by extracting features from it much more quickly. Once this is done, computers can process these features through a trained classification model and store classification results almost instantaneously in most cases, whereas humans need seconds or even minutes to arrive at a classification and then store this classification. The consequence of this is that computers can classify music many times faster than humans, something that is very significant when dealing with the potentially immense size of modern music collections.

Automatic classification is also much cheaper than manual classification. For example, consider the case of a large library that wishes to classify its music collection by genre. Accomplishing this task manually with any degree of reliability would require hiring several highly trained musical experts, each of whom must be remunerated appropriately based on their expertise. Furthermore, each recording should probably be classified by more than one human expert in order to minimize individual bias and

[5] www.ismir.net
[6] www.music-ir.org/mirex/2009/

increase consistency. These experts would also need to be trained to use the particular genre class ontology under consideration consistently, something that requires still further expenditure. Each of these experts would also require a computer and an appropriately quiet facility to work in, which also costs money.

Performing this task using an automatic classification system would require only one computer, although more could increase the speed even further, and could indeed be performed without purchasing new computers by using the same computers that the library provides to the public while the library is closed or when there is no demand for them. No human intervention is needed, other than beginning the process. Once provided with the data, the system could train itself within a matter of minutes to, at most, a week or two, and could then likely classify music at rates between several recordings a second to one recording every few minutes, per computer, depending on the algorithms used and the power of the computers.

An additional advantage of automatic classification is that classification models can be updated relatively easily. For example, to continue the library genre classification use case scenario, automatic classification models can be easily updated or retrained if a decision is made to add new genres to the class ontology. So, for instance, perhaps all Jazz music was simply labelled as *Jazz* in the original ontology. A decision is later made to also label it with sub-genres, such as *Dixieland, Swing, Bebop,* etc. If manual classification were being performed, then it would be necessary for new experts to be hired to go through all of the Jazz recordings. If automatic classification were being used, in contrast, then all that would be needed would be some ground-truth data to train a more detailed Jazz classification model, and then this could be used to quickly and economically update all of the appropriate labels automatically.

The techniques developed for a particular type of classification can also be relatively easily adapted to other types of classifications. So, for example, the same feature extraction and machine learning software used to classify music by genre could also likely be used to classify music by artist or mood, for example. The only modifications needed would be a new class ontology and ground-truth dataset to train the new classification model, with quite possibly no changes at all being necessary in the software itself or the computer(s) running it.

It is clear that automatic classification is dramatically cheaper and faster than manual classification. Indeed, it is cheap enough that it can be used for free or with minimal expenditure even by individuals who may have little or no music training on their own personal music collections for purely recreational purposes, something that is entirely out of the question for high-quality manual classification.

Of course, automatic classification is only useful if it can classify recordings at least as reliably as musical experts. In order to determine whether this is the case, it would be helpful to establish a baseline for human classification. Unfortunately, there is very little experimental evidence establishing such a baseline, and even if there were, it would be hard to generalize, since human performance is likely highly dependent on the type of classification, the particular class ontology, the particular instances to be classified and the expertise of the human classifiers.

Genre classification is one area where two relatively widely cited studies have been performed. The first found that a group of undergraduate students made classifications agreeing with those of record companies only 72% of the time when classifying among ten genres (Perrot and Gjerdingen 1999). Listeners in these experiments were only exposed to 300 ms of audio per recording, however, and higher agreement rates could quite possibly have been attained had longer listening intervals been used.

Another study involving longer thirty second listening intervals found inter-participant genre agreement rates of 76% (Lippens et al. 2004). However, one of the six categories used in this study was "Other," an ambiguity that could lead to substantial disagreement due to degree of membership and category coarseness issues rather than to entirely divergent classification perspectives.

So, although these two studies do provide useful insights, neither involved true musical experts, and there are some problematic experimental aspects with respect to determining an accurate classification baseline.[7] There is therefore clearly a need for more experimental evidence before definitive conclusions can be drawn regarding expert human genre classification performance. In any case, the best evidence we have to date seems to indicate that non-expert classification rates seem to lie roughly in the seventieth

[7] These problematic aspects are in no way an indication of poor research quality, as these studies are both of a very high quality. The issue is simply that the experiments were designed to address research questions that are only obliquely related to those being considered here.

percentile. Considering that this is undesirably low, it seems reasonable to hope that automatic classification systems can perform at least well as this, if not better. This is substantiated by the experimental results achieved with jMIR, as described in Section 9.4, where instances were classified automatically into a ten-class genre ontology with a success rate of 78.8%, a value slightly better than the human performances described above. Although this is still much lower than desirable for practical purposes, it does support the potential viability of automatic classification systems relative to manual classification in terms of classification reliability.

An additional advantage of automatic classification systems relative to manual classification systems is that they are arguably more likely to be able to perform classifications consistently. Multiple human classifiers, as are unavoidably necessary if large quantities of music must be classified manually in a reasonable amount of time, are each likely to have varying internal models of the class ontology under consideration, and as a result are each likely to vary somewhat in their classifications, even if they are all experts in the types of music in question. There is, for example, experimental evidence that even highly experienced adjudicators at music competitions can display a high degree of variability with one another, even when they are internally consistent (McWilliams 2005). Furthermore, individual classifiers are often not always even consistent with themselves, as their classifications can vary depending on their mood and other factors.

Computers, in contrast, will typically use the same consistent model once it is trained without variation, so a given trained model will output the same classification for a feature set corresponding to a given musical instance no matter when or by which computer it is classified. Although it is often possible to update the model if desired when new training data or features become available, this can be limited to circumstances where it is intended and desirable. Also, although one certainly wishes to have perfectly classified data, it is at least preferable to have misclassified data that is misclassified in similar ways when it is misclassified, as this makes it easier to detect and deal with the misclassifications. Automatic classification thus holds advantages over manual classification in terms of consistency, with respect to both correct classifications and incorrect classifications.

Automatic classification also has the advantage that computers can analyze and classify music in novel and non-intuitive ways that might not occur to human classifiers or researchers. Computer classifiers can thus avoid the preconceptions and biases that humans typically have, no matter how objective they try to be, which means that computer classifiers may find effective ways of classifying music that might not occur to human classifiers, and can also avoid contaminating experimental procedures and evaluations with such preconceptions. Furthermore, the development and application of automatic classification systems can result in insights and revelations that inspire new research ideas and directions that might not otherwise have been considered.

Of course, the labelling of the ground-truth data itself is still typically sensitive to human bias, which can, unfortunately, influence the quality of a model trained on it. It is therefore necessary to take special care when labelling ground-truth, and ideally multiple experts should label each instance and consult with each other in order to achieve as much consistency and correctness as possible. Although such a process can be expensive, labelling a small ground-truth dataset carefully is much less expensive than labelling an entire musical collection by hand.

Machine learning algorithms are also ultimately limited to the data provided to them, and human preconceptions can result in the failure to extract potentially useful features if care is not taken. This issue is one of the reasons why it is often best to extract many features and then automatically, and therefore objectively, reduce them to a manageable number using dimensionality reduction techniques, as described in Chapter 6.

Automatic music classification techniques can also have a number of important advantages with respect to musicological and music theoretical research. This is discussed in Section 1.3.4 below.

### 1.3.3 Applications of automatic music classification

Automatic music classification can be applied to a wide variety of tasks, both academic and commercial in nature. Correspondingly, there are many ways in which one can classify music, for many different purposes. For example, automatic music classification techniques can be of great use to: libraries and other institutions that archive music; composers and musicians who wish to make use of these technologies in their creative works; educational institutions that can use automatic classification techniques in pedagogically useful teaching software; courts making decisions on potential copyright violations; recording studios and record companies; music vendors; listeners who wish to improve and customize their listening experiences and personal music collections; and researchers in both music technology-oriented disciplines as well as in more traditional fields, such as musicology and music theory. This sub-section provides highlights of some of the ways in which automatic music classification can be of value to such users.

To begin with, automatic music classification is an essential part of many types of MIR research, as is made clear by an examination of the sub-disciplines of MIR outlined in Section 1.3.1. Indeed, many important areas of MIR research can be formulated directly as automatic music classification problems. Tasks such as genre, mood, artist and composer classification are all examples of this, as are tag prediction and classification by time period or geographical place of origin.

Many of the MIR research areas associated with musical similarity also use very similar techniques to those used in automatic music classification. For example, tasks such as playlist generation, music recommendation, cover song detection and hit prediction all typically involve many of the same features used in automatic music classification. Such tasks also typically require the collection and labelling of ground-truth data and application of machine learning algorithms, albeit sometimes using unsupervised rather than supervised approaches. The distinction between and relative advantages of classification-based and similarity-based systems are discussed below in Section 1.3.5.

Automatic music classification is also essential to many important sub-tasks associated with a variety of other MIR research areas. Many optical music recognition systems use classification-based approaches to identify qualities such as the pitch and rhythmic duration of notes, for example. Automatic music classification techniques are also often applied to the sub-tasks of automatic transcription, including areas such as pitch, chord, instrument, tempo, meter and key identification, as well as areas such as onset detection and segmentation.

Individuals can use a variety of tools based on automatic music classification to improve their personal musical experiences. For example: music recommendation systems can help them discover music that they might not otherwise know of; playlist generation software can customize their music consumption to particular listening scenarios and moods, as well as reacquaint them with portions of their music collections that they have unknowingly neglected; visualizations and music interfaces in general can help them browse their musical collections in new and interesting ways; sophisticated querying technologies can help them find music that they are looking for but having difficulty locating; and auto-tagging and classification systems can help them organize and annotate their music collections.

Musicians and composers can also use automatic classification technology in a variety of ways. Optical music recognition and automatic transcription software, for example, can reduce the work required to annotate, arrange, store and publish their compositions. Interactive accompaniment systems can also be used in concerts, and automatic classification techniques can open interesting opportunities in the implementation of artificial intelligence-based composition systems.

These technologies can also have significant pedagogical value when incorporated into music education software. Such software can detect and track errors in a student performer's pitch or rhythm, for example, and can automatically provide them with custom exercises to improve upon their particular weaknesses. Automatic analysis systems can also detect errors in student compositions or music theory exercises in order to help them improve. Score following and interactive accompaniment software can also accompany student musicians so that they can gain experience playing with other "musicians" when human musicians are unavailable to accompany them. Such systems can accompany a musician in a much more flexible and adaptive way than simply playing along with a recording. Automatic transcription systems can also help students annotate and learn music if they are unable to locate a score for it and have not yet acquired the skills or do not have the time to do so by ear.

Automatic music classification technologies can also be of great use to audio engineers and producers, both amateur and professional. The integration of functionality for tracking pitch and detecting note onsets and track segmentations, for example, have already significantly facilitated the use of digital editing software, and further improvements to related automatic transcription-related technologies could help to even further facilitate the process of editing and mixing music. Source separation functionality can also be extremely helpful when remixing music or making mashups when multitrack masters are not available.

Music vendors and producers have long recognized the potential of automatic music classification-related software, particularly with respect to similarity-related technologies, and have played an important role in funding much of the early research in this field. Software that can automatically label their music with various tags greatly decreases costs associated with hiring people to label their catalogues with metadata. These labels, along with visualisation technologies, can be essential in helping users search for and browse on-line music catalogues, particularly as these catalogues have grown in size. Fingerprinting technology has also been very useful in helping consumers identify music that they are interested in. Music recommendation technologies have drawn particular interest in recent years, as they can potentially be very useful in influencing customers to buy music that they might not otherwise have considered purchasing. There are also hopes that hit prediction technology can help record companies focus their resources on developing artists who are the most likely to be commercially successful.

There has been significant interest in similar technologies with respect to intellectual property rights. Cover song detection, artist identification, automatic music analysis and similarity analysis systems in general hold potential for use in, and in fact have been used, in legal cases attempting to determine if copyright infringements have occurred. Fingerprinting and watermarking technologies have also been proposed for use in detecting and tracking music piracy.

Libraries and other institutions that archive music have also long been interested in automatic classification technologies. Software that can automatically classify and label music with metadata quickly, accurately and consistently can be of particularly great value to them given their sometimes immense music collections and often limited budgets. Sophisticated musical querying functionality can also be of significant value, as it can permit queries based on sung pitches or tapped rhythms, for example, and also improve the results from queries entered symbolically, such as melodic themes or chord progressions. Optical music recognition and automatic transcription software can be especially useful, as it can be used to generate scores in digital form that can be easily accessed and distributed digitally.

### 1.3.4 Applying automatic music classification technologies to musicology and music theory

The use of automatic music classification and musical similarity measurement technologies in musicological and music theoretical research is something that warrants special attention. As convincingly argued by Huron (1999), musicological insight and scientific empiricism can greatly complement one another. It is particularly hoped that jMIR and other MIR-related technologies and software will help to bridge the gap between music information retrieval and the musicological and music theoretical research communities by placing powerful tools for performing large-scale automated feature extraction and machine learning at the latter's disposal. Musicologists and music theorists can in turn apply these tools to their research and provide feedback that can be used to enrich and improve these tools based on their musical expertise and experience.

An important factor contributing to the general relevance of MIR tools to music research in the humanities is the increasing availability of musical source materials in digital form. Libraries and archives are continually digitizing both scores and audio recordings, and are increasingly making the results and the related metadata available to researchers online. As noted by Huron (1999), the discipline is going from a "data-poor" field to a "data-rich" field. This is making wide-ranging empirical studies possible to an extent that was not previously feasible.

Although an expert human can certainly analyze one or several musical pieces with far more insight and understanding than a computer, such experts are limited in the number of pieces that they can analyze in a reasonable amount of time and in the range of musics that fall within the scope of their individual expertise. A computer, in contrast, can process huge quantities of diverse music many times faster than a human with perfect consistency. More specifically, feature extraction and pattern recognition techniques can successfully be applied to many varieties of art, popular and folk musics from various parts of the world, including those musics that do not have established theoretical frameworks.

The musical breadth permitted by research software that can very quickly process many thousands of musical works can reveal hidden musical insights and regularities that might not be apparent from studying just a few pieces, and can additionally allow one to empirically verify the validity of existing theoretical frameworks (e.g., Gingras and Knopke 2005). This can lead to important theoretical refinements and corrections, as well as inspire entirely new approaches and perspectives. Although traditional manual musicological or theoretical studies involving only a few pieces of music are certainly worthwhile and can be of value in increasing one's understanding of those specific pieces, the validity of generalized conclusions drawn from such research must of necessity remain in doubt until verified using much larger collections of music. The practice of making effectively unverified generalizations based on only a few musical examples was understandable in the past, given the lack of alternatives due to the relatively limited access to musical literature and the time constraints imposed by manual analysis. Computers and digitized music collections have now removed these constraints, however, making scientifically and statistically valid studies feasible and, one might argue, imperative to the advancement and validation of musicological and music theoretical scholarship.

Software utilizing sophisticated feature extraction and pattern recognition techniques and technologies can help to greatly expand upon the research benefits offered by more conventional music analysis software tools, such as Humdrum (Huron 2002). Analysis tools like Humdrum essentially serve as aids simply allowing theorists and musicologists to automate the types of tasks that they have traditionally performed manually, rather than allowing the application of fundamentally new approaches. Also, the application of traditional analysis software has typically incorporated assumptions that, whilst appropriate for the limited studies for which they were intended, nonetheless ultimately limit the types of music to which they can be applied. Of course, this is in no way meant to diminish the worth of such tools and approaches, as they are of proven value. They offer a number of benefits that pattern recognition tools do not, just as pattern recognition tools offer advantages that traditional analysis software tools do not.

Ideally, one would like to have a single software system that could be applied to classical music, jazz and a wide variety of popular and traditional musics, including cross-disciplinary studies that span diverse types of music. Furthermore, one would like to be able to use this software without needing to make any manual adjustments or adaptations in order to deal with different types of music. The types of feature extraction and pattern recognition techniques associated with automatic music classification make this possible. As noted above, musical research involving modern machine learning algorithms has the benefit of providing researchers with a fresh perspective on music, as models learned by such algorithms can avoid the potentially misleading ingrained assumptions and biases that humans invariably develop, despite their best efforts. Human researchers might unconsciously reject potentially valuable paths of inquiry because of such prejudices, whereas a computer would not.

In order to demonstrate the potential benefits of automatic music classification and similarity measurement-related technologies to musicological and music theoretical research, it is helpful to begin with a simple example. Consider a feature that measures the prevalence of parallel fifths in a piece. This feature could be input to a clustering algorithm, which would organize compositions into groups based on how often parallel fifths occur. One would expect the music of J. S. Bach, for example, to be clustered into a group with few or no parallel fifths, and the music of Green Day to be clustered into a group with many parallel fifths. If the clustering algorithm consistently segments musical examples along these lines, then this would confirm certain theoretical models concerning the differences between Baroque counterpoint and the types of chord progressions and voicings found in Punk music. If not, then this would lead one to question these same models. Additional types of music, such as Jungle electronic dance music, Romantic symphonies or Irish jigs, for example, could also be input to the model to see how they compare to other types of music with respect parallel fifths.

This example is, of course, very simplistic, and conventional analysis tools such as Humdrum could just as easily be used to perform equivalent tasks. However, the usefulness of such automatic music classification and similarity-based approaches becomes apparent when one considers less trivial examples where one considers hundreds of musical features and the potentially very complex ways in which they interrelate with one another, rather than just one feature. One will no longer have such clear expectations of how different types of music will be clustered or classified when many different features and types of music are under consideration, nor will it likely be practically feasible to manually formalize relationships between musically meaningful features and particular groups of music using conventional analysis techniques with any empirically verifiable validity.

It is clear that considerations relating to harmony, rhythm, melody, texture, dynamics, instrumentation and other factors are all essentially intertwined. A certain melody in a given harmonic context might be appropriate only when certain notes are played softly, for example, or perhaps only when certain notes fall on weak beats. A certain chord voicing might sound better using a particular instrumentation than another. Skilled musicians and composers clearly intuitively incorporate such knowledge into their work, but it is rarely clearly and unambiguously specified in formal theoretical models, largely because of the difficulties discussed above. Any attempt to isolate one set of musical parameters from all others, while traditionally necessary to make possible any meaningful analysis at all, nonetheless comprises the completeness, scope and validity of the resultant theoretical models due to the associated failure to consider music in its full holistic sense.

It is also important to consider the issue of how one might represent and analyze the output of a feature extraction system that includes dozens or hundreds of dimensions. Even traditional statistical co-occurrence and correlation-based analyses are limited in how well they can represent such data in ways that are musically meaningful when compared to what can be achieved by the more sophisticated statistical techniques associated with machine learning-related dimensionality reduction, classification and clustering. Such techniques allow high-dimensional information to be meaningfully and conveniently represented in low-dimensional spaces as self-organized clusters or specifically labelled categories. Some machine learning techniques, such as decision tree algorithms, also allow empirically learned rules and dependencies to be examined directly.

Of course, traditional manual and computer query-based analysis techniques have not typically involved hundreds of features such as this, perhaps precisely because of these problems. As a consequence, ultimately undesirable simplifications have often been unavoidable in the past with respect to the number of musical characteristics considered simultaneously. As noted above, however, this past necessity does not mean that such simplifications should be propagated now that more powerful alternatives are made available by pattern recognition-based technologies.

So, it is clear that approaches based on machine learning techniques have a number of important advantages over manual or conventional computer-aided analyses: the ability to consider many musical variables and the dependencies between them at once; the elimination of the necessity of explicitly formalizing, often subjectively, the types of relationships between musical characteristics that one wishes to compare ahead of time; the avoidance of potentially invalid theoretical assumptions and simplifications; the ability to very quickly study huge and diverse collections of music; and the ability to represent the results of sophisticated processing in relatively simple and easy to understand ways. It is particularly important to reemphasize that machine learning algorithms can help to avoid the biases that humans inevitably develop, despite their best efforts to remain objective. The fresh perspective offered by machine leaning-based processing can thus provide human researches with valuable ideas, insights and inspiration that they can then build on.

It is important to stress once again that it is recognized here that machine learning is a substitute neither for human researchers nor for traditional analysis software like Humdrum. It is highly improbable that a computer will independently evolve any perfect musical model on its own. The great amount of generalized musical knowledge, experience and understanding that human experts have available to them, as well as their human intuition, certainly allows them to achieve results in musicological and music theoretical studies that would be missed by computers. Furthermore, the human perception of music is really the only fundamental truth of significance when one is considering music, and this is ultimately what machine learning-based systems should model, even if indirectly.

The arguments above do, however, demonstrate the very significant value of machine learning-based musicological and music theoretical research as an important supplement to manual human research, particularly with respect to exploratory research and verification of theoretical models on larger and potentially broader musical datasets than could feasibly be studied by a human expert. Pattern recognition techniques can reveal insights that might otherwise be obscured, and can perturb human researchers out of ideological ruts that they may have fallen into.

It is also possible to use data generated by automatic classification systems to perform research into how we construct the notion of musical similarity and form musical groupings. Automatically generated feature weightings can be used as experimental data providing some indication of the relative importance of different features in distinguishing between particular classes. Genre classifiers, for example, can be used to gain experimental insights into what characteristics are important in distinguishing between particular genres and how these characteristics vary (McKay and Fujinaga 2005a). Classification results can be used to support sociological and psychological research into how humans construct the notion of musical similarity and form musical groupings, and how this compares to the "objective" truth produced by computer-based classifiers.

There are also a number of more specific ways in which automatic music classification-based technologies can be useful to music researchers. To provide a practical example, such technologies can provide more effective and sophisticated querying software that can be very useful in helping researchers find particularly pertinent musical material even when performing manual analyses or studies.

Optical music recognition and automatic transcription technologies also allow digital symbolic representations of music to be made available to them in cases where physical scores may be rare and hard to come by or, in the latter case, if only audio recordings are available. This can be particularly useful when studying types of music with no written tradition as well as for analyzing performance practices that are not typically encapsulated by traditional symbolic representations.

There are also a number of very direct applications of automatic music classification in musicology and music theory. One might, for example, train classifiers to recognize the work of particular composers, and then use the trained classifiers to attempt to determine if compositions whose authorship is unknown are in fact likely to belong to be the work of one of these composers. One could also use such systems to detect and classify particular musical devices and stylistic characteristics in association with particular composers, time periods, geographical locations, styles and so on.

### 1.3.5 Comparing musical classification and similarity measurement

Automatic music classification and musical similarity measurement are sometimes treated as entirely separate sub-disciplines in the music information retrieval literature, and are sometimes treated very similarly. This emphasizes the varying opinions as to the links between these two approaches and the relative appropriateness of each of them to various types of tasks. It is therefore useful to clarify the differences between them, as well as to discuss their relative strengths and advantages.

In the case of classification, the essential problem is to fit each instance into one or more appropriate classes found in a pre-existing class ontology. Similarity, in contrast, does not make any *a pirori* assumptions about the existence of such a class ontology, or even necessarily of classes at all. Similarity-based research attempts to arrive at some measurement of similarity between instances that indicates how strongly each instance is related to each other instance, without reference to an external structure.[8]

For example, genre and artist identification are clearly classification problems, as pieces of music are classified into pre-existing genre or artist classes, respectively. In order to perform such tasks, one must first have candidate genre or artist classes to which each instance can be mapped. Applications like music recommendation and playlist generation, in contrast, are typically posed as similarity problems. One must respectively recommend music to a listener based on its similarity to what he or she is known to like or choose music that would go well together because it is sufficiently similar or, in the case of playlists that emphasize variety, dissimilar. The decision of whether to recommend a particular piece or place it on a particular playlist is thus typically made based on its similarity with other pieces music, not on what classes it belongs to.

Having noted this, it is possible to pose certain problems as either classification or similarity problems. For example, mood extraction could be treated either as a classification problem, where the mood of a piece is associated with one or more candidate mood classes or, less commonly, as a similarity-based clustering problem, where features are chosen such that the music is grouped based on mood-related considerations. To give another example, even though playlist generation is typically

[8] It is certainly possible, however, to derive class or other structures from similarity measurements. For example, one can derive discriminants separating classes defined by groups of similar instances that are well clustered in similarity space.

posed as a similarity problem, as noted above, it could also be treated partially as a classification problem where piece is classified into listening scenario and/or mood classes. An appropriate listening scenario or mood could then be chosen for the playlist, and then only pieces with appropriate scenario or mood labels would be considered for inclusion on the playlist. Even music recommendation could theoretically be treated as a classification problem with two classes, where one class is *recommend* and the other is *not recommended.*

Indications have been found in some psychological research that there may be a very strong link between the processes used by humans to measure similarity and to classify instances. As noted in Chapter 2, exemplar-based models of human classification in particular are generally based upon the assumption that humans classify each instance that they observe by measuring its similarity to exemplars that they have stored in an internalized similarity space.

There is often a significant overlap in the techniques and technologies used in automatic classification and in similarity measurement. For example, similar features are often extracted and used for both. Machine learning techniques are often used for both as well, although classification research usually involves supervised learning and similarity is more likely to involve unsupervised learning.

Having noted there are certainly important links between automatic classification and similarity measurement, there can also be important differences between them. One may not have any good-quality class ontology, for example, or it may be impossible or inappropriate to apply one to a given problem, in which case a similarity-based approach is clearly appropriate. Alternatively, one might be interested in associating instances with particular labels, in which case a classification-based approach would be much better.

jMIR is designed specifically to perform automatic music classification. This emphasis on classification is a bit of a deviation from the general momentum of MIR research, which has in recent years tended to emphasize similarity research over classification research, although a significant amount of research on the latter is certainly still performed.

Commercial software has also tended to focus more on similarity in recent years, particularly as the industry has come to place more importance on tasks such as music recommendation, playlist generation and hit production over more traditional classification tasks. Part of this is because such similarity-related applications most strongly facilitate the ultimate goal of commercial software, which is to sell music; part of it is because some classification problems have essentially been solved, such as fingerprinting; part of it is due to the fact that that problems such as genre or artist classification have turned out to be more difficult to perform successfully and to scale than initially anticipated; and part of it is because there seems to be a common belief in industry that similarity-based music selection and browsing have a greater appeal to listeners than more classification-oriented querying approaches to retrieving music.

It is argued here that, contrary to this recent tendency to emphasize similarity over classification, classification does in fact have a number of important benefits over similarity with respect to a variety of important research tasks and domains. It is also, of course, recognized that similarity can also have its own advantages with respect to other tasks and research domains. In essence, research in both areas can be very helpful, and can each be more or less applicable to different problems. Since it has been proposed by some strong proponents of similarity research that classification research should be abandoned in favour of similarity, however, it is appropriate to briefly emphasize some of the relative strengths of classification.

Genre classification in particular serves as a good example to center this discussion around,[9] as it is one of the most difficult and inherently ill-defined kinds of classification, and thus particularly subject to criticism with respect to its usefulness. It has been suggested by some that genre is a hopelessly ambiguous and inconsistent way to organize and explore music, and that users' needs would be better addressed by abandoning it in favour of more general similarity-based approaches. Those adhering to this perspective generally hold that genre classification is only a subset of the broader and, to them, more meaningful similarity problem, and that genre classification is only worth pursuing as an initial stage of research where features and pattern recognition algorithms can be developed, compared and refined in preparation for a more general similarity-based approach.

[9] McKay and Fujinaga (2006) present a more detailed discussion of the relative strengths and weaknesses of genre classification and more similarity-based approaches.

Although it is certainly true that genre classification is related to similarity in a variety of ways, genre has a strong association with culturally determined and defined classes, as discussed in Chapter 2. This association to external classes suggests that there is more to genre than simple similarity. Even similarity measurements that involve cultural features such as playlist co-occurrence tend to be based on individual preferences rather than genre's more formal and general cultural characteristics. In essence, the query "find me something like this (relatively small) set of recordings" is intrinsically different from "find me something in this generally understood genre category," which could encompass a potentially huge set of recordings and that is based on culturally determined categories rather than more content-oriented or individually defined notions of similarity.

Furthermore, although browsing and searching by genre is certainly not perfect—and alternatives are always worth researching—end users are nonetheless already accustomed to browsing both physical and on-line music collections by genre, and this approach is proven to be at least reasonably effective. A relatively recent survey, for example, found that end users are more likely to browse and search by genre than by recommendation, artist similarity or music similarity, although these alternatives were each popular as well (Lee and Downie 2004). Resources such as the AllMusic Guide,[10] which use labelled fields such as genre, mood and style, are also commonly used, while alternative similarity-based interfaces have yet to be as widely adopted by the public, despite the significant research and commercial attention that they have been given. MIR researchers and software developers should avoid adopting a patronizing approach where they insist that end users abandon a form of music retrieval for which they have a demonstrated attachment and which they find to be useful.

Categories such as genre also have significant importance beyond simply their utility in organizing and exploring music, and should not be evaluated solely in terms of commercial viability. For example, classification-oriented labels such as genre and mood have the important advantage that they provide one with a vocabulary that can be used to easily, efficiently and effectively discuss musical categories. Conversations concerning more general notions of similarity, in contrast, are quickly limited by the lack of such a vocabulary and the necessity of making frequent references to musical examples in order

[10] www.allmusic.com

to make oneself understood. Moreover, such discussions can be unclear in terms of which dimensions of similarity are being considered with respect to the musical examples that are used.

Furthermore, many individuals actively identify culturally with certain genres of music, as can easily be observed in the divergent ways in which many fans of Death Metal, Classical Music or Rap dress and speak, for example. Genre is so important to listeners, in fact, that psychological research has found that the style of a piece can influence listeners' liking for it more than the piece itself (North and Hargreaves 1997). Additional psychological research has also indicated that categorization in general plays an essential role in both music appreciation and cognition (Tekman and Hortacsu 2002).

Research in classification can also provide valuable empirical contributions to the fields of musicology and music theory in ways that are distinct from the also valuable contributions offered by similarity research. Research that forms correlations between particular cultural and content-based characteristics and particular categories and ontological structures, for example can have important musicological and music theoretical significance.

Once one accepts the usefulness to listeners and music consumers of categories such as genres, then the advantages of automatically classifying recordings stored in large music databases becomes clear. This is particularly true given the difficulty and time requirements associated with manually labelling the huge and rapidly growing musical databases that are becoming increasingly common, as noted above.

One of the most common criticisms of genre classification systems, and classification systems in general, is that high-quality ground-truth can be difficult and expensive to label properly, with the consequence that the quality of trained classification models and the metrics used to evaluate them is compromised. However, it is important to point out that similarity measurement has many of its own problems related to ground-truth ambiguity and subjectivity, particularly when it comes to evaluating systems and comparing their performance. Indeed, problems in objectively and consistently evaluating the quality of similarity measurements are probably even greater than those associated with classification. It is therefore inconsistent to promote similarity as a superior alternative to categorical classification specifically because of problems relating to ground-truth.

Ultimately, research in both classification and similarity can be very useful, and neither should be neglected. Furthermore, as noted in above, there is a strong correspondence between the component tasks that must be addressed in order to perform either automatic music classification or similarity measurement. These include areas such as ground-truth collection and labelling; feature design and selection; and the application and training of machine learning algorithms, be they supervised or unsupervised.

The jMIR applications are thus very much applicable to certain aspects of similarity research as well as to classification research. For example, jAudio, jSymbolic and jWebMiner can be used to extract features from datasets such as Codaich, SAC and Bodhidharma MIDI, which have had their metadata validated by jMusicMetamanager, and these extracted features can serve as the basis for similarity research just as much as they can for classification research.

### 1.3.6 Existing non-academic systems

This section highlights prominent commercial and other non-academic software systems and services that utilize or provide technologies associated with automatic music classification. A particular emphasis is placed on systems providing APIs that can be used in research projects. A great many companies have designed systems associated with various aspects of similarity and automatic music classification, so for the sake of brevity this section focuses only on systems that are particularly well-established or have significant value with respect to performing original automatic music classification research. More details on other more specialized systems are presented in the chapters of this dissertation that are the most directly related to them.

As noted above, commercial systems in general have tended to focus more on similarity-based tasks than on more structured classification problems. This is likely because music recommendation in particular has long been seen as an area with significant potential profitability because of its ability to increase electronic music sales by making the long tail[11] accessible to consumers and to promote music in general. This

[11] In the context of music, the "long tail" refers to types of music that each have a small market share, but which are significant in aggregate (Anderson 2006).

has also helped to motivate work in other similarity-related areas, such as playlist generation, which can be useful in generating advertising revenue and subscription fees from personalized Internet radio stations. A number of companies have also used their services as a tool for collecting extensive valuable data associated with their users' listening and tagging behaviour.

Fortunately for the MIR research community, several commercial entities have been kind enough to make their data and/or tools available for free via web services.[12] Unfortunately, very few of these companies publish open-source code, with the result that developers may use their tools only as "black boxes." Although there are therefore no guarantees as to how long these resources will continue to be available, they have already been used to great benefit by many MIR researchers and music developers in general.

Amazon[13] was one of the first companies to provide automated recommendation services to the public at large. In addition to offering basic search functionality, this on-line retailer provides two primary tools that users can use to access music. The first involves browsing through hierarchically organized genre or "subject" categories, and the second involves recommendations based on music that other customers with similar buying histories have purchased. The former approach involves manually classified music, and is thus not directly relevant to automatic music classification. The latter approach essentially consists of collaborative filtering, a strategy that has a number of significant weaknesses, as discussed in Section 5.2.4. Nonetheless, the huge wealth of data available to Amazon has made this feature at least somewhat effective, and at the very least it is likely good at promoting impulse purchases. Amazon provides access to much of its data via web services.[14]

MoodLogic was one of the first recommendation engines dedicated specifically to music. Unfortunately, it has been inactive since 2003, although it has been absorbed by the Rovi Corporation,[15] who may integrate its technology into the All Music Guide.[16] The original MoodLogic approach consisted of building a database of song profiles based on

[12] "Web services" are resources available on-line that can be accessed by software via standardized interfaces.
[13] www.amazon.ca
[14] aws.amazon.com
[15] www.rovicorp.com
[16] www.allmusic.com

the ratings of users that could then be used to classify and recommend music. In particular, class types such as genre, mood and "perceived energy" played an important role. Fingerprinting was used to identify songs in users' collections so that metadata could be downloaded from the MoodLogic database. Each user was provided with a certain number of "credits" that they could use to access song profiles, and further credits could be obtained by profiling songs themselves or by paying.

Pandora[17] is a well-known Internet radio station that integrates automatic music recommendation technology. Unfortunately, Pandora no longer allows access to its services outside of the U.S.A. due to licensing constraints. Pandora is built upon the Music Genome Project (Joyce 2006), which was an effort to describe music with vectors consisting of hundreds of "genes" or "musical attributes" describing each song. Each gene describes some characteristic of the music, such as "gender of lead vocalist," "level of distortion on the electric guitar" or "type of background vocals." These genes can then be used to rank similarity using a distance function. Although the process of recommendation is fully automated, the manual annotation of each song can be a very expensive process.

Users of Pandora can either listen to general genre-based stations, or they can seed their own station with songs or artists of their choice. Listeners can respond to each song presented to them negatively or favourably, which helps refine their station and allows Pandora to collect additional information that can be used to integrate a collaborative filtering element into their recommendation algorithm.

Last.FM[18] is another high-profile Internet radio station that is integrated with a music recommendation engine. Unlike Pandora, Last.FM focuses on a collaborative filtering approach. Although this does suffer from the same problems of all collaborative filtering, Last.FM has the significant advantages over Amazon's approach of focusing on detailed listening behaviour rather than on purchasing behaviour; of focusing on song-by-song data rather than album-based data; and of using social networking functionality to facilitate the collection of valuable user data, such as personal recommendations and user tags. Last.FM has been particularly proactive and helpful in providing access to their data

[17] www.pandora.com
[18] www.last.fm

through a powerful web services API.[19] This API has been used by many MIR researchers in their own work.

Musicovery[20] is another interactive web radio station that also provides interesting visual musical maps. It allows users to search for stations by entering artist names as well as by entering filtering metadata relating to mood, tempo, danceability, genre and time period.

Apple incorporated an automatic playlist generator called the "Genius" into its iTunes media player,[21] iPods and iPhone in 2008. This feature received a great deal of publicity, although responses to it have been mixed. Although the details of the algorithm have not been publicly released, it appears that individual pieces in a user's music collection are identified based on audio fingerprints, and collaborative filtering oriented information such as user ratings and skip counts are then used to generate short playlists from the user's collection based on a single seed song provided by the listener. iTunes can also generate "Genius Recommendations" consisting of songs that are judged to be similar to a seed song by the Genius but are not in the user's music library.

Sun Microsystems has been involved in a number of research projects associated with music browsing and recommendation, namely the Search Inside the Music project (Lamere and Eck 2007) and the Music Explore FX.[22] Both of these allows users to explore music based on similarity, with the former emphasizing graphical browsing spaces and the latter presenting music in more conventional clusters that incorporate the notion of "hotness" and "familiarity."

The now inactive Sony Cuidado (Pachet et al. 2006) system also focused on providing users with convenient browsing interfaces, although with more of an emphasis on traditional metadata-based filtering than the Sun systems. Cuidado collected metadata from users and also extracted and predicted it using automatic music classification technology.

The above examples are just a few of the many commercial browsing, recommendation and playlist generation systems that have been released, some of which

[19] www.last.fm/api
[20] musicovery.com
[21] www.apple.com/itunes/
[22] blogs.citytechinc.com/sanderson/?p=105

are no longer active, and some of which have proven themselves to be commercially successful. Many of these are excellent systems, and most operate on some combination of collaborative filtering, manually edited metadata and content-based audio analysis. Most also operate using the same basic business models, which is to say they typically offer some combination of Internet radio, musical social networking and personal music collection filtering and organization to gain revenue from advertising, relayed music sales and data collection. Additional examples of such systems include Audiobaba,[23] Critical Metrics,[24] ExploreMusic,[25] The Filter,[26] Grooveshark,[27] MediaUnbound,[28] MusicStation[29] and Slacker Personal Radio.[30]

The types of software tools mentioned above present users with many new and powerful ways of accessing and discovering music. Although the public at large has not yet taken full advantage of such tools, they do have a great deal to offer and can potentially have a significant impact on listening behaviour and music consumption. It is worth noting that a number of scholars have provided important and insightful observations on related social, cultural and commercial issues, such as Anderson (2006), Jennings (2007) and Moscote Freire (2007).

As noted above, there are a number of commercial systems that make musical data or processing functionality available to developers and researchers via web services. Last.FM and Amazon have already been credited for the valuable services that they provide to the MIR community, and there are certainly others as well.

The Echo Nest[31] has been particularly proactive in providing powerful tools via their API.[32] Although the processing itself is closed source, as are almost all of the systems discussed in this section, it does provide extensive functionality for analyzing, retrieving metadata, and processing music. Several useful third-party libraries and tools have also been developed using the Echo Nest API.

[23] www.audiobaba.com
[24] criticalmetrics.com
[25] www.exploremusic.com
[26] www.thefilter.com
[27] listen.grooveshark.com
[28] www.mediaunbound.com
[29] www.omnifone.com
[30] www.slacker.com
[31] echonest.com
[32] developer.echonest.com

Yahoo! Music[33] provides access to a huge amount of musical information via their Music API.[34] Yahoo! also offers many other APIs[35] that may be used for musical data mining, including the Yahoo! Search Web Services,[36] which provides access to general Yahoo! search functionality, and Flickr Web Services,[37] which may be used to access images relating to music. Google[38] also offers several APIs[39] that may be adapted to musical applications, including a YouTube API.[40]

MusicBrainz[41] offers a fingerprinting service that allows users to access its huge recording metadata database using web services.[42] Of particular interest, MusicBrainz takes the rare step of making some of its developer tools open-source. MusicBrainz is associated with FreeDB,[43] a database of CD track listings, which also has its own API.[44] Gracenote[45] also offers access to a huge quantity of recording metadata via fingerprinting, but its developer API[46] offers limited benefits relative to MusicBrainz if one does not purchase a commercial licence from Gracenote.

There are also several sites that make lyrics available via web services. Of particular interest, Lyrcisfly[47] and LyricWiki[48] allow developers to download lyrics and associated metadata via, respectively, URL querying[49] and a SOAP-based API.[50]

Discogs[51] is a community-built database of musical metadata. Although the range of fields that it offers are somewhat limited, its data is extensive and consistently formatted.

[33] new.music.yahoo.com
[34] developer.yahoo.com/music/
[35] developer.yahoo.com/everything.html
[36] developer.yahoo.com/search/
[37] developer.yahoo.com/flickr/
[38] www.google.com
[39] code.google.com/more/
[40] code.google.com/apis/youtube/overview.html
[41] musicbrainz.org
[42] wiki.musicbrainz.org/Products
[43] www.freedb.org
[44] www.vbaccelerator.com/home/VB/code/vbMedia/Audio/CD_Tracklistings/article.asp
[45] www.gracenote.com
[46] doors.gracenote.com/developer/
[47] lyricsfly.com
[48] lyrics.wikia.com
[49] lyricsfly.com/api/
[50] lyrics.wikia.com/LyricWiki:SOAP
[51] www.discogs.com

More significantly from an MIR perspective, it also makes this data available via an API.[52]

Billboard[53] has made information associated with their historical sales charts available to developers as well.[54]

The BBC[55] has made a significant amount of musical information available, in their case via a RESTful framework.[56] Of particular interest, they have integrated their information with MusicBrainz and Wikipedia. In the U.S.A., National Public Radio[57] has made information associated with their radio stations available via an API.[58] YES.com[59] also tracks playlist information from commercial radio stations in the U.S.A. and makes this information via an API.[60]

Gigulate[61] is a music news aggregator, with a particular emphasis on tracking concert dates. This data is made available via an API.[62] Songkick[63] also has an API[64] that gives one access to information on live performances. Idiomag[65] provides access to musical articles, as well as multimedia content, via an API.[66]

There are also a number of more general-purpose sites that have useful APIs from the perspective of MIR. For example, Wikipedia has an API,[67] although it is relatively under-documented. Freebase[68] also makes its data available via web services.[69] Although the data available through Freebase is not as extensive as that accessible via Wikipedia directly, it is more structured and its API provides better access to it. DBPedia[70] also provides structured access to Wikipedia data.

[52] www.discogs.com/help/api
[53] www.billboard.com
[54] developer.billboard.com
[55] www.bbc.co.uk
[56] www.bbc.co.uk/music/developers
[57] www.npr.org
[58] www.npr.org/api/
[59] www.yes.com
[60] api.yes.com
[61] gigulate.com
[62] gigulate.com/api/
[63] www.songkick.com
[64] developer.songkick.com
[65] www.idiomag.com
[66] www.idiomag.com/api/
[67] en.wikipedia.org/w/api.php
[68] www.freebase.com
[69] www.freebase.com/docs/web_services
[70] dbpedia.org

Some general social networking sites also offer powerful APIs that can be used to access music-related data, even though they do not focus on musical applications in particular. Examples of such APIs include MySpace,[71] Twitter,[72] and Facebook,[73] with MySpace's API in particular offering some specialized musical functionality. iLike[74] also has a Facebook application for sharing musical information, and it has an API.[75]

BLIP.fm[76] is a more specialized social service that allows users to register as "DJs." Each DJ may submit songs packaged with short textual comment, known collectively as a "blips." DJs may also submit a limited number of "props" to other DJs who they feel have submitted good blips. Those DJs who receive many props gain prestige as well as the ability to issue additional props. From an MIR perspective, this process produces rich data that BLIP.fm has made available via an API.[77]

PeoplesMusicStore[78] is an on-line retailer that allows users to serve as "storekeepers" who each promote artists of their choosing via metadata entries and recommendations to others. Users can purchase music by discovering it through the promotional efforts of storekeepers, with the result that the recommending storekeeper receives points earning prestige and credit with which they can purchase music themselves. This generates useful data from an MIR perspective, which can be accessed via the PeoplesMusicStore API.[79] Although popular music is available at the PeoplesMusicStore, the focus is on lesser-known music. SoundCloud[80] and its API[81] also emphasize music that is not as well known. Music released under Creative Commons licensing can also be accessed from the Free Music Archive[82] and its API.[83]

[71] developer.myspace.com/community/
[72] apiwiki.twitter.com
[73] developers.facebook.com
[74] www.ilike.com
[75] www.ilike.com/developer/signup
[76] blip.fm
[77] api.blip.fm
[78] peoplesmusicstore.com
[79] wiki.github.com/peoplesmusicstore/api
[80] soundcloud.com
[81] soundcloud.com/api
[82] freemusicarchive.org
[83] freemusicarchive.org/api/docs/

Spotify[84] is a music streaming service that provides APIs[85] that allow developers to access both streamed audio and metadata. Listiply[86] also provides a way of accessing Spotify playlists. Rhapsody[87] is a subscription-based streaming on-demand music service that also offers an API[88] providing access to some of its data. 7digital[89] is another company that focuses on selling and delivering multimedia content, including music. It has an API[90] that can be used to access its catalogue, including tags, although it does not promote it strongly for public use.

Playdar[91] is a system allowing users to search various locations for music based on metadata tags. It has a simple HTTP API,[92] and it is possible to write plug-ins for it.

As noted above, only a few commercial systems have focused specifically on music classification, and when they do it tends to be only in combination with similarity-based methods, such as in the case of Cuidado. This is unfortunate, as many music consumers still do seek music using categories like genre and artist name (Lee and Downie 2004). As a consequence, music retailers continue to organize their music by category, whether they are brick and mortar stores like HMV or Walmart, or on-line stores like Amazon and iTunes. This requires a significant amount of overhead, due to the costs of manual classification and metadata entry. Similarity-oriented services based on manually edited data, such as Pandora, could also benefit from classification technology that could predict appropriate tag values automatically.

Fortunately, significant work has been done in academic contexts that is more directly related to music classification, and less exclusively focused on similarity-based tasks. Moreover, many academic systems have the significant advantage from a research perspective of being open-source.

[84] www.spotify.com
[85] developer.spotify.com
[86] www.listiply.com
[87] www.rhapsody.com
[88] webservices.rhapsody.com
[89] ca.7digital.com
[90] www.7digital.com/api
[91] www.playdar.org
[92] www.playdar.org/api.html

### 1.3.7 Existing academic systems

This section highlights existing academic software tools that have been developed for performing research in automatic music classification. There are a great many tools that have been developed for specialized types of music classification, or for performing specific sub-tasks associated with music classification, so this section focuses only on those tools that are intended for general-purpose music classification or are particularly well-established. More details on these systems and on more specialized systems that are omitted here can be found in the chapters of this dissertation that are most directly related to them.

Some of the earliest audio analysis systems were developed for application to speech rather than music. Nonetheless, some of these tools do have a number of analysis, visualization and statistical tools that can be usefully applied to certain MIR research tasks. WaveSurfer (Sjölander and Beskow 2000) is an open-source tool that can be used to visualize audio and extract simple features from it. Praat (Boersma and Weenink 2009) is another system originally intended for speech that allows one to analyze, synthesize and manipulate audio.

Marsyas (Tzanetakis and Cook 2000) is a pioneering C++ system that played an essential role in popularizing automatic music classification in general and audio analysis in particular in the MIR community. Marsyas emphasizes audio information extraction and processing, and also includes machine learning functionality.

Music-to-Knowledge, or M2K (Downie, Futrelle and Tcheng 2004), is a graphical patch-based system implemented in Java. It is designed to be applied to a broad range of applications, including machine learning and feature extraction, and is a powerful tool for prototyping new systems or integrating existing systems.

M2K makes use of the D2K distributed processing framework, something that is both a strength and a weakness. D2K allows M2K to process tasks using many computers simultaneously, and also provides good classification libraries and a flexible GUI framework. Unfortunately, D2K's licence can make it complicated for researchers outside of the U.S.A. to gain legal access to it, and D2K still has a number of unresolved bugs, something that can be particularly problematic since not all of D2K is open-source.

Fortunately, a descendant of M2K is currently being implemented using SEASR, or the Software Environment for the Advancement of Scholarly Research (Llorà 2008), as an alternative to D2K. SEASR offers more functionality, is more reliable and is less limiting than D2K. This is being done as part of the NEMA project.[93]

MIDIToolbox (Eerola and Toiviainen 2004) and MIRToolbox (Lartillot, Toiviainen and Eerola 2008) are powerful modular Matlab toolboxes for visualising, extracting information from and processing symbolic and audio data, respectively. These systems include cognitively inspired analytical tools relating to melodic contour, similarity, key-finding, meter-finding, segmentation and other high-level musical analysis. These tools can be very useful for rapidly prototyping new systems, but can pose a barrier to users without a strong signal processing and coding background, or who do not have access to Matlab.

CLAM (Amatrain, Arumi and Ramirez 2002; Amatriain and Pau 2005; Arumi and Amatriain. 2005) is a well-known audio analysis system implemented in C++. It includes extensive functionality for processing, synthesizing and visualizing audio, as well as for extracting certain features. Although the software can pose initial obstacles to non-specialist users who wishes to quickly extract features or design new features, its powerful functionality makes it worth the effort, and it can be an effective tool for application building.

Sonic Visualiser (Cannam et al. 2006) is a system for visualizing audio data in a variety of ways, and also allows one to extract certain features from the audio. This software can be applied to a wide variety of MIR tasks, including automatic music classification. Sonic Annotator[94] is an associated command-line program that can be used to batch extract features from audio files and publish them to the semantic web in RDF form. One of the important advantages of Sonic Visualiser and Sonic Annotator is their incorporation of Vamp[95] plug-in functionality. Vamp is an external API that can be used to add functionality to compatible software for processing audio in various ways. Sonic Visualiser, Sonaic Anotator and the Vamp plug-in format are all implemented using C++, although it is also possible to implement Vamp plug-ins in Python.

[93] nema.lis.uiuc.edu
[94] www.omras2.org/SonicAnnotator
[95] www.vamp-plugins.org

Work has also been done on adding embedded feature extraction and machine learning functionality to the ChucK audio programming language (Wang, Fiebrink and Cook 2007; Fiebrink, Wang and Cook 2008a). This has the potential to result in a very flexible environment for quickly implementing feature extraction and analysis tools that can be used in real-time, although the available functionality is still somewhat limited at the time of this writing.

The Humdrum toolkit (Huron 2002) is perhaps the best-known symbolic analysis toolkit, with a variety of query tools and specialized high-level musical representations. Although feature extraction is certainly not the software's primary intended purpose, Humdrum data has at times had features extracted from it, such as in the work of Sapp, Liu and Selfridge-Field (2004). Knopke (2008) has written new implementations of much of the Humdrum software that makes it more useful from an MIR perspective, including functionality such as MIDI-compatible file format translation.

Although it is neither a specialized MIR system nor an academic system, some researchers choose to perform research directly in MatLab.[96] MatLab is a powerful general framework with a variety of well-developed and well-tested machine learning and signal processing toolboxes. The disadvantages of MatLab are that it is a proprietary closed-source framework, and can also suffer from efficiency and extensibility issues. Nonetheless, it can be very useful for rapid prototyping.

Despite the many benefits of the systems described above, none of them are designed to deal with the full range of tasks associated with general-purpose automatic music classification. To the best of the authors' knowledge, jMIR is the only unified automatic music classification framework that addresses all, or even most of, the essential tasks of symbolic, audio and cultural feature extraction; meta learning-based machine learning; metadata management; dataset collection; and standardized file format design. Furthermore, most but not all of the systems described above suffer from limitations with respect to extensibility and usability. Most of these systems also focus on specific problems and limited types of data, and do not integrate easily with one another. These limitations, among others, have led to the design priorities underlying jMIR, as described below.

[96] www.mathworks.com

## *1.4 jMIR's core objectives and characteristics*

Although some very important gains have been achieved recently in automatic music classification-related technologies, it has become evident that the pace of progress has slowed in recent years. An examination of results of the annual MIREX[97] competition, for example, will demonstrate that relatively few improvements have been made in the best success rates from year to year in various classification-related evaluation tasks. Such limitations have been anticipated and observed for a number of years, such as by Berenzweig and his colleagues (2004) or by Aucouturier and Pachet (2004). There are a number of problems with the ways in which automatic music classification software and techniques have traditionally been designed, implemented and distributed that are likely contributing to the recent failure to achieve significant classification performance gains, or at least hampering the MIR community's ability to progress past this apparent "glass ceiling" on performance.

These problems served as motivators for the development of jMIR, and played an important role in guiding its design principles. This section describes the overall goals and design priorities of the jMIR project, and how these goals and priorities address some of the factors currently limiting automatic music classification systems in general. This section also describes the corresponding characteristics shared by all of the jMIR components. The particular implementation of jMIR's design objectives in any given individual jMIR component, however, is explained in that component's chapter, as are any additional design characteristics particular to that component.

### 1.4.1 Bridging the gaps between the MIR-related disciplines

Music is a multi-faceted area of inquiry, with the many factors that influence any individual's experiences of music leading to many different ways of considering music. Given the consequent diversity of approaches to studying music, researchers sometimes have a tendency to focus their attentions on limited types of music or on particular specialized approaches to studying music.

As noted in Section 1.3, MIR is greatly enriched by contributions from researchers from fields as diverse as machine learning, data mining, digital signal processing, music

[97] www.music-ir.org/mirex/2009/

theory, musicology, music psychology, music education, human computer interaction and the library sciences. Of course, no single individual can reasonably be expected to have true expertise in more than one or a few of these fields, so there is a need for ways to facilitate the integration of knowledge, insights and methodologies from each of these fields. For example, an electrical engineer might have invaluable expertise with respect to extracting information from digital audio signals, but might lack the music theoretical and psychological background to design features that would be the most musically meaningful or effective for a given task. To give another example, musicologists and music theorists have extremely valuable musical knowledge but do not necessarily have the background to fully take advantage of machine learning or digital signal processing technology in their research.

jMIR is designed specifically to help break down these kinds of barriers between research disciplines. jSymbolic, jAudio and jWebMiner, for example, allow even users with limited or no knowledge in, respectively, symbolic music, digital signal processing and web-based data mining to utilize powerful techniques from each of fields. Moreover, ACE may be used by researchers with little or no background in machine learning to automatically build powerful classification models using such features. Furthermore, ACE XML allows all of this information to be expressively and flexibly stored and communicated so that it can be used in combination with still other tools and techniques that may be of interest to individual researchers.

Another important advantage of jMIR is that it is designed to facilitate research using different types of music, something that is important in investigating the generality of any given research approach or perspective. For example, Codaich and the Bodhidharma MIDI dataset both contain examples of music from many different genres of music, thereby providing researchers with a diverse set of musical data that can be used to carry out or validate their research. Furthermore, jMusicMetaManager helps to manage and verify the metadata associated with these or other large collections of music. ACE and the jMIR feature extractors are also designed to extract useful information from as many different types of music as possible. All of this allows jMIR to aid researchers in performing research involving large and diverse collections of music, something that can help to overcome the scientific and statistical weaknesses of the approach that one sometimes encounters of investigating the applicability and validity of various approaches and perspectives on only a few relatively similar musical examples.

Finally, jMIR is not specifically tied to any particular kind of classification. This means that it can be applied agnostically to the diverse range of musical classification problems described in Section 1.3. The jMIR components can thus be used to accomplish goals in many different MIR sub-disciplines, whether these sub-disciplines have traditionally focused on automatic classification or not.

### 1.4.2 Facilitating the effective combination of different types of musical data

A cross-disciplinary approach not only has benefits with respect to taking advantage of diverse types of expertise, but also with respect to the types of data that can be studied. For example, an engineer at a commercial lab with access to a large audio research database, a computer scientist at an on-line social music service with access to valuable listener data and a musicologist associated with an academic library with a large collection of scores each have access to very different types of musical data.

Traditionally, individual music researchers have each tended to focus on only one type of musical data, depending on their particular backgrounds. Unfortunately, research with a limited scope such as this risks missing out on valuable complementary sources of information. Audio is clearly useful because it represents the essential way in which music is consumed; cultural data is well-known to be highly influential on our interpretation and experience of music; and symbolic data incorporates valuable high-level musical abstractions. Features extracted from each of these types of data sources have an increased likelihood of orthogonal independence, which can significantly increase classification performance compared to cases where one is limited to extracting information from only one of these types of data sources. The experiments described in Section 9.4, for example, provide empirical evidence supporting the benefits of combining different types of musical data in this way.

jMIR is designed not only to facilitate the integration of information extracted from symbolic, audio and cultural data sources, but also to encourage it. Firstly, it includes feature extractors that can extract features from all three types of data. Secondly, it allows these features to be combined cleanly, consistently and transparently via the ACE XML file formats. Thirdly, ACE can be used to process features in a way that is indifferent to the type of musical data from which they have been extracted. Finally, Codaich and the Bodhidharma MIDI dataset provide researchers with audio recordings, symbolic recordings and metadata that can be used to extract cultural features. In particular, the SAC dataset is specifically designed for carrying out research using all three types of musical data.

As a matter of related interest, it can be argued that it is particularly important at the moment to promote MIR research on symbolic music, a type of data that has become somewhat unfashionable amongst MIR researchers in recent years in comparison to audio and cultural data. The value of symbolic data is clear: it is relatively easy to extract features from symbolic data that incorporate high-level musical knowledge, something that can be particularly valuable in performing many kinds of musical classification. For example, it can be very difficult to detect cover songs if one does not have access to high-level musical features providing insights on information such as chord progressions or melodies, as lower-level features will likely obscure the essential musical invariances of different arrangements of the same piece.

Although there is currently relatively little commercial interest in studying or collecting and annotating symbolic data, an individual taking a long-term perspective will note that automatic transcription technology is continuing to move ever closer to being able to produce symbolic transcriptions from audio recordings that are of a sufficiently high quality for automatic classification purposes. Even if resultant transcriptions are too error-prone for use in performance contexts, for example, it has been found that classification systems in particular can be relatively robust to errors (e.g., Lidy et al. 2007).

Once transcription systems that are sufficiently accurate for feature extraction become available it will then become a simple matter to extract high-level features from the resultant intermediate symbolic representations. For example, one might implement an intermediate transcription layer to generate a MIDI file from audio analyzed by jAudio, which could in turn have features extracted from it by jSymbolic. The high-level features output by jSymbolic, as well as the original low-level jAudio features, could then all be processed together by ACE, without the need for any original musical data other than the audio itself. It is therefore useful to put effort into research on high-level features now so that it can be immediately taken advantage of as transcription technologies improve. Such research can easily be performed now using existing symbolic recordings. Furthermore, there is certainly a great deal of valuable music theoretical and psychological research that can be performed directly on symbolic musical representations, without any reference to audio recordings.

### 1.4.3 Accessibility and ease of use

Software that is overly technical or difficult to learn can pose significant and alienating barriers to potential users. This can be a particularly important issue with respect to software that is intended for users with backgrounds in diverse disciplines, as is the case in MIR, since such users share only a limited common knowledge base. If a designer wishes such users to adopt and gainfully use his or her software tools, then he or she must place a special emphasis on making the software widely accessible.

Unfortunately, much of the software produced for and by MIR research tends to emphasize essential functionality at the expense of usability. Although this is certainly understandable given researchers' limitations with respect to manpower and the pressure to meet publication deadlines before technologies become obsolete, it nonetheless has had the consequence of making much MIR software inaccessible to casual users and, in some cases, to effectively all but the developers of the software themselves. For example, the researchers running the annual MIREX[98] MIR competition have found that systems submitted to MIREX rarely work in their originally submitted form, even when required by the evaluators to meet set basic interface requirements, a minimal step towards usability that is itself not required of published systems in general.

jMIR, in contrast, is designed with the goal of making it as easy to use as possible for researchers from the full range of MIR-related disciplines, with a particular emphasis on making the software accessible to researchers with little or no training in areas directly related to engineering and computer science. As noted above, researchers such as psychologists, musicologists and music theorists have essential expertise and insights that are invaluable to MIR research, but not all such researchers have strong backgrounds in computer-related fields. Such researchers may thus be easily alienated by software

[98] www.music-ir.org/mirex/2009/

requiring extensive computer skills or backgrounds in areas such as machine learning and signal processing. Indeed, software with steep learning curves can be discouraging even to users with extensive backgrounds in such areas, so maximizing accessibility and usability are in fact important in increasing the likelihood that users of all backgrounds will take advantage of the software.

In order to accomplish this, it is necessary to at once limit the exposure of users to overly technical details and terminology in the software interface when such content is unlikely to be necessary, while at the same time allowing highly technical and detailed options to be available when the user is in fact interested in them. One way of accomplishing this is to give users a choice of interfaces. For example, users with only a limited technical background might be very uncomfortable using non-graphical interfaces, but other users might prefer to use a command-line interface in order to facilitate batch processing. Alternatively, users with computer programming backgrounds might be more interested in well-developed APIs that allow them to integrate the software's functionality into their own code.

The jMIR components were produced with this in mind. Each of the jMIR components therefore has a clear and easy to use API, and all of the components also include a GUI. ACE and jAudio also provide users with the option of accessing their functionality via command-line interfaces if desired.

The jMIR software also hides technical details from users when they do not need to be aware of them. For example, the jMIR feature extractors auto-schedule feature extraction based on feature dependencies, and automatically extract features that are not explicitly requested by users but are necessary for calculating other features that have been requested, all without requiring any knowledge on the part of the user of this processing or of the corresponding feature dependencies. These dependencies are made very clear in the code itself, however, so that this information is easily available to developers who do need to be aware of it.

Customizability is also an important priority to pursue when maximizing the usability of software, as different users will want to use software in different ways to perform different tasks under different conditions. The jMIR components are each designed with this principle in mind. To provide a few examples, the jMIR feature extractors allow users to select which specific features they want to extract; jAudio allows users to customize pre-processing options as well as individual feature parameters; jWebMiner and jMusicMetaManager allow users to customize the types of processing to be performed and the information to be included in generated reports; and ACE allows users to select different machine learning options, such as whether to order instances randomly during training, which of several variations on stratified or unstratified cross-validation to use and so on.

Documentation is also essential to making software accessible, with respect to both code documentation and user manuals. In terms of the former, all of the jMIR code is very well-documented with detailed Javadoc-formatted comments. In terms of the latter, ACE, jAudio, jWebMiner and jMusicMetaManager all have detailed HTML frames-based user manuals, and jSymbolic includes a helpful README document. Most of the jMIR components also include menu accessible help functionality.

It is also important to make software easy to install and use on the operating system of the user's choice. For example, requiring users to recompile code in order to install it, and potentially adjust *make files* and resolve linking problems when doing so, will immediately render the software completely inaccessible to users with limited computer backgrounds, and can very well irritate even expert computer programmers to the point that they choose not to use the software. jMIR is therefore implemented entirely in Java,[99] with the consequence that it benefits from Java's platform independence. All third-party libraries used by jMIR are also implemented in Java and are packaged and installed with jMIR, so there is no need to acquire or install any additional technologies or resources.

Although it can be very difficult to avoid all conceivable installation difficulties, jMIR is designed to make the process as simple as possible. Each of the jMIR components is contained in a simple and self-contained folder[100] that may be copied to the directory of the user's choice, and the software can be run directly from a Java JAR file. Each jMIR component also includes a detailed README file that provides easy-to-follow instructions clearly describing the simple installation process. For developers who

[99] java.sun.com
[100] jAudio is the exception to this, as decoding MP3s requires that a third-party file be placed in the Java extensions folder. However, the jAudio software is distributed with installation software to perform this task automatically for the user.

wish to extend jMIR, development versions of the jMIR components have been packaged as NetBeans[101] projects, and the source code may also be easily compiled outside of NetBeans if desired using standard Java development tools.

It is also ideally preferable to avoid requiring that users have access to proprietary software or libraries in order to use one's software, as this limits the accessibility of the software. jMIR is available free of charge, and none of its components require the use or installation of any closed software or libraries. jMIR and the third-party libraries used by it may also be freely redistributed. This means that both the jMIR software and future software utilizing its functionality will remain accessible even to users with very limited budgets.

### 1.4.4 Longevity and extensibility

There are many examples of excellent music software packages that were essentially abandoned after their initial development and publication. An unfortunate by-product of the academic process is that the publication of new research tends to be rewarded but the support, refinement and incremental improvement of existing software tends to be more difficult to use as a means of acquiring funding and publications. Furthermore, the graduate students who develop the majority of academic software systems typically move on to other pursuits after graduating, often without the opportunity or incentive to continue working on the software that they developed. The consequence of this is that even software with excellent potential often falls out of use as technologies progress and improve but the software is not kept up to date.

The solution to this is two-fold. Firstly, it is important for research labs, peer-review committees and granting agencies to invest in long-term software support so that good systems stay up to date and continue to improve. Secondly, it is essential to design software such that it is easily extensible both by the original developers and by others.

Although the first of these solutions is of course largely out of the control of individual developers, the second is certainly not. The original developers of a given software system will be more likely to support and expand the software themselves in the long-term if doing so is relatively easy to do. Even more importantly, other users will

[101] netbeans.org

themselves contribute improvements and updates to the software if they find this to be easy and useful to them. Emphasizing extensibility is therefore one of the most important contributions that developers can make towards improving the probable longevity of their software. Of course, this is easier said than done, as writing extensible code requires significantly more thought and time, but the long-term dividends make it worth the effort. This is demonstrated by the long-term success and growth of existing software projects such as Gamera (MacMillan, Droettboom and Fujinaga 2001), for example.

jMIR emphasizes extensibility in a variety of ways. Most essentially, it is entirely open-source, which means that developers can see exactly how the jMIR code works, without having to deal with any "black box" code. The code is also all clearly formatted and documented, as noted in Section 1.4.3. The prioritization of extensibility is also evident in the jMIR class structure and organization, which reflect both good object oriented design principles and a general cognizance of how future users are likely to want to use and expand the code.

An additional important point is that jMIR does not utilize any external technologies other than the very well established Java language. In addition to the advantages associated with Java's platform independence, this avoids the risk associated with the possibility that external technologies will become obsolete or difficult to find, thereby compromising the longevity of all software built using them. jMIR uses only a few third-party libraries, and they are all purely Java-based as well.

The ACE XML file formats are also designed to be extensible, especially the ACE XML 2.0 formats. These file formats provide a variety of optional and highly flexible modes of data expression, and are not specifically tied to any of the jMIR components.

The jMIR components are also themselves decoupled, so that they can be used either independently or together, as is convenient for any given user. For example, the jMIR audio feature extraction code is in no way dependent on machine learning code. This has important advantages, since users interested in particular jMIR components do not need to consider or even be aware of how the other jMIR components work if they do not wish to, thereby significantly reducing the learning curve associated with extending individual jMIR components. For those developers who do wish to extend multiple jMIR components, however, each of the components are designed using common conventions and design practices, which facilitates learning in their case as well.

There are a number of additional aspects of the jMIR code that favor extensibility, such as its use of portable modular components and of metafeatures and aggregators for auto-generating new functionality. Details are provided below in Section 1.4.5.

### 1.4.5 Providing a framework for developing new approaches

The jMIR components are designed so that they can be used directly as distributed, which is to say as ready-to-use MIR software applications allowing researchers to utilize automatic music classification technologies in their own work. This aspect of jMIR is essential to fulfilling the goals described in Sections 1.4.1, 1.4.2 and 1.4.3. This, however, only reflects one of the primary use cases for which jMIR is designed. The ability to use jMIR as a framework for developing new technologies and approaches also figures very prominently in jMIR's design.

The development, deployment and evaluation of new MIR research assets such as features, pattern recognition methodologies and metadata annotation strategies often requires a significant amount of infrastructural investment. Once a given lab has developed this infrastructure, the expenditures associated with its creation often make the lab reluctant to move away from it. The consequence of this is that time and resources are taken away from original research while the infrastructure is being implemented and, once that infrastructure is complete, it is typically not compatible with infrastructures used by other labs.

jMIR is designed to address this problem by, in addition to providing a set of ready-to-use applications, also providing a framework for developing, testing and distributing new techniques and algorithms. This allows jMIR to be used to implement and experiment with new technologies, not just apply existing ones. This can be very helpful in eliminating the need for each lab to develop its own infrastructure, as much of the infrastructure needed to perform MIR research involving automatic music classification is already implemented in jMIR. The extensive documentation and emphasis on extensibility associated with jMIR, as described in Sections 1.4.3 and 1.4.4, help to facilitate the accessibility of this infrastructure.

The jMIR components are each designed to have a generally modular structure overall. In particular, most of the jMIR components allow new functionality to be added via a simple and standardized plug-in architecture. For example, each of the features extracted by jAudio or jSymbolic are implemented as independent classes that extend a common superclass. New features can be added simply by extending this class and following its clear and simple conventions. A dependency on another feature, for instance, can be implemented simply by adding a reference to the name of that feature to one of the superclass' inherited fields. As noted above, the jMIR feature extractor will then automatically extract the value of the associated feature and provide it to the new feature at runtime. The implementation of a new feature therefore requires little more than the implementation of the processing associated with the new feature itself, since a complete extraction infrastructure is already provided by the jMIR feature extractor. Only a very minimal knowledge of the overall architecture of the jMIR feature extractor is necessary to add new features.

Other jMIR components also incorporate similar modular architectures. For example, ACE uses Weka (Witten and Frank. 2005) machine learning algorithm implementations. Weka is also open-source and Java-based, and it is supported by a very active development community. It is very easy to add additional Weka algorithm implementations to ACE so that they will be automatically used in meta learning trials. New machine learning algorithms can thus be implemented using the Weka framework, and then added to ACE, as can new algorithms added to the Weka distribution by others. The error-detection algorithms used by jMusicMetaManager are also relatively modular.

jAudio in particular goes especially far in facilitating the addition of new algorithms through metafeature and aggregator functionality. In essence, these automatically implement features based on other features, such as calculations of the variance of a feature or feature histograms. Metafeatures and aggregators are each described in more detail in Sections 3.4.7 and 3.4.8, respectively.

One of the essential advantages of jMIR's modular approach is that it facilitates the iterative development of new algorithms that build upon other algorithms, and that can then themselves be used as a building blocks for implementing still further algorithms. This makes it possible to utilize a design approach that incorporates ever increasing levels of abstraction. To provide a simple example, one could extract the RMS (Root-Mean-Square) feature from each of a series of analysis windows, perform an auto-correlation on this data to produce a beat histogram and then use this histogram to extract higher-level information on tempo and meter.

The widespread use of a common MIR development platform such as jMIR would not only allow individual researchers to build upon their own work in this manner, but also to add new jMIR algorithm implementations created by others to their own jMIR distributions and distribute their own new algorithms so that others may do the same with their work. This aspect of jMIR is discussed further in Section 1.4.6.

There is also further infrastructure built into jMIR that facilitates its use as a development platform. For example, ACE XML allows valuable data and metadata to be stored and communicated between applications in flexible ways. jMIRUtilities automates various useful administrative tasks. Also, the jMIR datasets are useful in providing data that can be used to test and validate new algorithms.

This last point emphasizes the particularly important advantage offered by jMIR of facilitating experimental comparisons of newly implemented algorithms with alternative pre-existing approaches. The performance of a new feature or machine learning strategy, for example, can be easily compared with other features or machine learning strategies already implemented in jMIR or previously added to it by using ACE's meta learning functionality to perform comparative experiments. This issue is also discussed in further detail in Section 1.4.6.

These characteristics of jMIR contrast with what one finds when using most alternative music research software. Most such software is poorly documented; difficult to install and use; difficult to extend; and not characterized by the modularity necessary to facilitate its use as a shared development platform. In particular, feature extraction code is often tightly coupled to pattern recognition and analysis code, something that hampers the reuse of the code in other contexts, especially in the case of researchers who wish to develop features and machine learning algorithms independently. Such software is also often tied in to specific types of applications and research goals, which limits its general usefulness. Finally, many music research systems suffer from dependencies on particular platforms or proprietary technologies, which limits their usability for other researchers who cannot or do not wish to develop code using these platforms or technologies.

### 1.4.6 Facilitating and promoting inter-institution collaboration

As noted above in Section 1.4.5, many MIR research groups have had a tendency to develop their own in-house software tools and musical datasets, an approach that has a number of important disadvantages in addition to those described above:

- The implementation of a full in-house MIR research framework requires a very substantial amount of effort. Since this framework itself is typically essentially secondary to the research that it is intended to support, researchers will have a tendency, and understandably so, to compromise functionality, documentation and testing of the framework when resources and time become scarce in favour of devoting them to the primary research. This results in support infrastructure that is likely not as well developed and tested as it would be if the infrastructure itself were the primary goal of those implementing it.
- Each research group tends to specialize in one or a few MIR sub-disciplines, yet the work required to develop a full in-house MIR research software framework often requires work associated with additional disciplines in which the lab members do not have expertise. For example, an MIR lab in an electrical engineering department that develops its own in-house system will likely need to collect and annotate ground-truth datasets, but will probably lack members with the musicological training to do so properly.
- Different implementations of the same algorithm may in fact differ subtly in ways that are not always apparent or well documented in publications, but that can impact results. For example, one implementation of genetic algorithms might use islands but another might not, or two different implementations of the Spectral Roll-off Point feature might use different values for the fractional parameter (e.g., 0.85 or 0.95).
- Each research group will typically develop their framework in a way that is convenient to their own available resources and research goals, with the result that

core infrastructure such as data structures and file formats, as well as the technologies used to implement them, are often incompatible with those used by other research groups.

The end result of this is that the quality of in-house MIR research frameworks tends to be compromised in certain respects and, at the very least, requires a substantial initial investment of time and resources that is often a redundant and wasteful duplication of work already done by others. Even more importantly, the lack of standardization makes it very difficult to share resources and data conveniently and efficiently between different research groups. This is a very serious problem, as the success of any field is dependent upon the ability of different researchers to evaluate the performance of their approaches relative to those developed by others, and their ability to incorporate successful approaches used by others into their own work.

The facilitation of and promotion of inter-institution collaboration is therefore one of the core goals of jMIR. This is tied in closely to the qualities of jMIR discussed in Sections 1.4.4 and 1.4.5, since the implementation of this goal is dependent on the ability of researchers to extend jMIR and use it as a development platform.

If researchers implement new features and pattern recognition algorithms in the jMIR framework, for example, this then means that they can immediately distribute their jMIR implementations to other researchers. These other researchers can then easily plug these implementations into their own jMIR installations and use them in their own research, or even extend them. This means that powerful new algorithms can immediately be used by the entire jMIR-using community in their own research without any difficulties or delays associated with resolving incompatibilities or implementation ambiguities. Even more importantly, it also greatly facilitates the iterative development of increasingly sophisticated approaches that build upon one another, not only by individual researchers, but by the entire jMIR-using community.

One particularly helpful step in facilitating such sharing of implementations would be to build a common repository of algorithms implemented under a standardized framework such as jMIR. Such a repository would have important advantages for developers, as it would allow them to easily distribute and publish their implementations, as well as for users, as it would provide them with an easily accessible and up to date set of tools that they could expect to operate consistently for all users who have added them to their jMIR distributions. This would also allow researchers from diverse backgrounds to easily take advantage of the work of researchers from other disciplines, an important advantage for the reasons discussed in Section 1.4.1.

The use of a single development environment like jMIR by a large segment of the MIR community would also have important advantages with respect to the validation and comparison of different approaches to solving MIR problems. Modern scientific research methods are predicated on the ability of different research groups to independently verify and evaluate each other's work. This is essential in ensuring research transparency and in allowing researchers to build upon each other's work. If software incompatibilities and inconsistencies make it difficult for one group to verify the results of another group or to fairly compare their approaches to solving a given problem with those used by another group, then the consequence is that the merits of one approach relative to another become ambiguous, and the progress of the field is slowed down by uncertainty as to which solutions are the best to use and build upon. Even when one approach can be demonstrated to be better than another, a given research group may still be reluctant to adopt it if its implementation in their own potentially incompatible research infrastructure would involve significant time and effort.

These problems would be eliminated if the competing algorithms were each implemented in a single framework, such as jMIR. Any researcher could then simply add the jMIR implementations of the various algorithms to their jMIR installation and perform evaluative experiments using them. The dimensionality reduction and meta learning implemented by ACE are particularly useful in this respect, as the performance of these tasks implicitly involves comparisons of different features and machine learning algorithms. One can also use jMIR to perform more traditional comparative experiments as well, of course. The jMIR datasets also provide a large, diverse and ready-to-use collection of ground-truth data that can be used as a basis for such experiments.

The annual *Music Information Retrieval Evaluation eXchange (MIREX)*[102] competition has played an invaluable role in facilitating the comparison of different approaches. Unfortunately, MIREX is only run once a year, and only systems that are

[102] www.music-ir.org/mirex/2009/

submitted to it by their authors can be evaluated. Furthermore, as indicated above, it has been noted by the MIREX organizers that submitted systems rarely work in their originally submitted form even when required by the organizers to meet specific basic interface requirements. This has necessitated a significant amount of work on the part of the organizers in order to perform the evaluations. A shared infrastructure would be very helpful in addressing such problems, since algorithms to be evaluated could simply be submitted as jMIR classes, for example, and installed without needing compatibility modifications on the part of either the submitters or the evaluators.

Such an approach is an important part of the Networked Environment for Music Analysis (NEMA)[103] project, which will include jMIR components in a distributed processing and data infrastructure framework that will allow users to automatically and independently submit algorithms and processing requests. A very significant advantage of shared infrastructures like jMIR and NEMA in general is that they allow individual researchers to run large-scale evaluations of their new approaches relative to other existing approaches at any time.

jMIR also has additional advantages with respect to the promotion of inter-institutional collaboration beyond its extensible plug-in architecture. The ACE XML file formats are particularly important in this respect, as they allow users to store extracted feature values, ground-truth labels and a diverse range of metadata in expressive, flexible and clear ways. This means that different research groups who do not have access to the same datasets or who are using different infrastructures can still share essential information using ACE XML as a standardized format.

As a final point, it is important to note that too wide a range of tools intended for general MIR research use can have the unintended consequence of placing barriers between different user groups, just as too few tools can have the same effect by encouraging the use of specialized in-house tools. For example, users who have adopted jMIR might be unwilling to take advantage of some of the useful tools described in Sections 1.3.6 and 1.3.7, and vice versa. Such a scenario would certainly be counterproductive from the perspective of promoting inter-institutional collaboration. Although it might be stated here that jMIR is the best choice and that it should be adopted

[103] nema.lis.uiuc.edu

by all MIR researchers, the developers of other tools might very well make the same claim, with the result that each researcher in the MIR community will make their own choices, and different tools will be adopted by different institutions.

This potential problem is one of the key motivators behind the ACE XML file formats, as they are designed to facilitate data communication across diverse toolsets. Another important step is for the developers of each toolset to provide simple and well-documented APIs to facilitate the porting of new algorithms to multiple toolsets. This has been another motivating factor for jMIR's emphasis on extensibility and documentation, as described in Sections 1.4.3 and 1.4.4, and it also provides a strong argument in favor of an overall distributed processing umbrella such as that proposed by NEMA.

## 1.5 Highlights of research contributions

Much of this dissertation places significant emphases on the engineering work associated with each of the jMIR components and on background knowledge that will be useful in helping music researchers new to the various fields touched on by jMIR understand them better. This is done in order to provide as much information as possible to readers about why and how they can use the software effectively to perform music research, and it is in keeping with the overall goal of the jMIR project of facilitating effective cross-disciplinary original research by others.

It is important to emphasize, however, that the jMIR project was not simply an engineering project, and that it also involved significant theoretical work and experimental research as well. A considerable amount of exploratory work was done during the design phase of each of the jMIR components, for example. It was not possible to include all of the associated details in this already lengthy document, however, due to space constraints. The purpose of this sub-section is to highlight those portions of this work that it was possible to include, and to summarize the results of some of the experiments that have been performed with jMIR.

One of the most important theoretical contributions of the jMIR project is the library of high-level features outlined in Section 4.5, many of which are original. Just as important is the theoretical discussion presented in Sections 4.1.2 and 4.4 of how one should approach the problem of designing and choosing high-level features. This is an important area that is rarely touched on in the literature, as those designing computational features rarely have extensive music theoretical or musicological knowledge, and music scholars very rarely consider music in terms of features intended for processing by pattern recognition algorithms.

Another area of significant theoretical interest is the discussion of fundamental issues associated with the collection and labelling of ground truth data, including concerns related to class ontologies. These issues are treated rather carelessly in too much MIR research, a problem that is explored in Section 8.2, along with proposed guidelines for addressing these problems in a responsible way. These guidelines played an important role in methodology used to assemble the jMIR datasets, as described in Sections 8.5.2 and 8.5.3 in relation to the Bodhidharma MIDI dataset, for example.

An additional important theoretical contribution is the discussion of the limitations of existing file formats associated with representing data relevant to automatic music classification, and the proposal of design priorities to consider when creating new formats, as discussed in Sections 7.3 and, especially, 7.4. This theoretical work was essential in the design of ACE XML and ACE XML 2.0, as described in Sections 7.5 to 7.11, which was itself an important research contribution.

This first chapter has also already presented an analysis of some of the current problems in the current state of automatic music classification research. Several general solutions have also been proposed in order to move past these problems.

A number of original algorithms have also been developed during the course of the jMIR project, such as the automatic feature extraction scheduling and dependency resolution algorithm used by jAudio and jSymbolic, as discussed in Section 3.4.6. The novel accessible approaches to metafeatures and feature aggregators, as described in Sections 3.4.7 and 3.4.8, are also helpful new contributions.[104] A series of new metadata error and redundancy detection algorithms were also developed as part of jMusicMetaManager, as described in Section 8.4.

A significant amount of experimental research has been performed during the course of the jMIR project. Space and time constraints made it impossible to include all of it in

[104] It is important to emphasize that the co-developer of jAudio, Daniel McEnnis, was chiefly responsible for the software's metafeatures and aggregators.

detail in this dissertation, which is why only brief overviews are presented of some of the results, such as the exploration of the kinds of metadata errors common to music found on-line (Section 8.6.4) or the types of high-level features that are most effective in classifying music by genre (referred to briefly in Section 4.5.1). Other experiments, such as those described in Section 9.2, or the MIREX 2005 winning performance of the jSymbolic features on symbolic genre classification described in Section 9.3, were more associated with evaluating system performance than with revealing musical insights.

An experiment that was particularly important in revealing meaningful information about music, however, was that described in Section 9.4. This experiment explored whether classification performance gains could be realized by combining multi-modal information, specifically features extracted, for each piece of music, from audio recordings, symbolic recordings and cultural data available on-line. The results indicate that there are indeed statistically significant performance gains when features extracted from two or three of these types of data are combined, compared to when only one type of data is used. It was also found that features extracted from cultural data are not only helpful in improving classification performance, but also in reducing the seriousness of those misclassifications that do occur.

More details on these and other theoretical and experimental contributions of this dissertation are provided in each of the remaining chapters. Sections 10.1 and 10.2 also provide more detailed general summaries than that provided here, albeit ones that include novel engineering contributions as well as research contributions. In addition, summaries of the particular contributions of Chapters 3 to 9 are presented near the end of each of these chapters.

# 2. Related research from psychology and the humanities

## *2.1 Chapter overview*

This dissertation as a whole focuses primarily on technical procedures and tools for automatically classifying music in various ways. Strictly speaking, it is not necessarily to imitate the processes used by humans in order to perform such classifications, as entirely different process can potentially be used by computers to simulate human classification behaviour. Computers process information in ways that are intrinsically different from the mechanisms used by the human brain, and the best approaches to modelling human behaviour with computers may well be very different than those actually used by humans themselves.

Having noted this, automatic music classification is, at least from the perspective of most currently conceptualized applications, still very much a problem of modelling human behaviour. The particular musical categories that humans use as well as the particular classifications that they make are often subjective from a purely data-driven perspective, and sometimes even irrational. Since most automatic classification applications require simulating the subjective and potentially irrational classification judgements that humans may make, it is therefore useful to gain as much of an understanding as possible about the mechanisms used by humans to make classifications. This can help in implementing automatic classification systems that anticipate and imitate human behaviour as well as possible, regardless of whether the systems actually implement similar mechanisms.

This chapter is therefore presented in order to provide a basic grounding in some of the research in psychology and in the humanities on human music classification and similarity judgement. Such research is rarely given the attention that it deserves in the music information retrieval literature, and it is presented here in order to help address this shortcoming. Research in psychology and the humanities can be essential not only in providing inspiration for techniques that are implemented in automatic music classification systems, such as the choice of features to extract, but can also be essential in designing as well as evaluating the relevance and significance of automatic music classification evaluation experiments.

The material in this chapter is intended only to present particularly significant highlights of the relevant literature in psychology and the humanities, however, and is certainly not a comprehensive survey. This chapter is intended primarily to provide context and background for the rest of this dissertation, which focuses more specifically on using computers to perform automatic music classification. More detailed coverage of research on human music classification is well beyond the scope of this dissertation and is outside of the author's area of expertise. The works of Lakoff (1987), Ashby (1992), Nosofsky and Zaki (2002) and Smith and Medin (1981) provide broad overviews of established research, and there are many other excellent sources as well, including many that highlight more recent approaches and results. Many more details are available in the works referred to in this chapter and in the psychological, musicological and music theoretical literature at large.

Section 2.2 describes the psychological models that have been developed with respect to human categorization and classification in general, without particular respect to music. This includes explanations of both classical models and exemplar-based models, which are the two most commonly espoused general models for human classification, as well as a few other models and miscellaneous relevant psychological insights.

Section 2.3 focuses on psychological research that has been done specifically on music classification. The topic of similarity is emphasized here as well because it is essential to most exemplar-based models of classification and because it has been a particular focus of research in music psychology.

Section 2.4 presents insights drawn from the musicological and music theoretical literature as a complement to the psychological research presented in the previous two sections. A particular, although certainly not exclusive, emphasis is placed on genre classification, as this is one of the hardest and most interesting kinds of music classification, and because it is particularly emphasized in the literature.

## 2.2 General psychological classification models and research

### 2.2.1 Classical classification theory

The oldest known theory of classification, known as the *classical theory,* dates back to Aristotle. It is based upon the idea that, in order to be considered a member of a particular category, an entity must fulfill each of a set of necessary and sufficient conditions required by the category in question. The details of the classical theory are well documented in the literature (e.g., Smith and Medin 1981).

To provide an example of how the classical theory works, one might consider an equilateral triangle. Such a triangle is defined as a closed figure with three sides of equal length, where each side must be a line segment. If all of these conditions are met, then an entity may be considered to be an equilateral triangle. Failure to meet any of these conditions, however, would mean that the entity would fail to qualify as an equilateral triangle.

There are a number of problems with the classical theory, as famously noted by Wittgenstein (1953) and verified both experimentally and theoretically by many others. For example, different entities belonging to the same category can have some fundamentally different key characteristics. Not all games can be said to meet the same requirements, for instance, as some involve competition and some do not.

An additional problem is that classifications can often be context-dependent, and can depend on one's current state of mind as well as background. The classical theory incorrectly assumes absolute rules in all cases.

Yet another problem is that numerous experiments have demonstrated that people often consider some entities to be better, or more typical, examples of a category than others. This is called *centrality,* and is associated with *membership gradience,* or the notion that categories can have degrees of membership as well as unclear boundaries. For example, a robin can be considered to be a better example of a bird than a peacock, or a chair can be considered a better example of furniture than a shoe rack.

There have been a number of attempts to modify the classical theory in order to account for such problems. One modification permits an entity to be considered a member of a category if it successfully meets some but not all of its membership rules (e.g., Wittgenstein 1953). This approach allows some entities to be better members of a category than others because they fulfill more rules than other lesser members. Different rules may also be given different weightings. Another modification allows categories to incorporate diversity by allowing them to be defined in terms of a union of other smaller categories each with their own rules which may differ from one another.

It is interesting to note that the classical model is in some ways similar to expert system approaches to computer-based classification (see Section 6.2.2).

### 2.2.2 Exemplar-based classification theory

The classical theory waned in popularity as evidence mounted against it. One of the most influential alternative theories to be proposed was based on the notion that categories, rather than being defined by rules, are defined based on *prototypical exemplars,* and classifications are made by comparing probe entities with these prototypical exemplars (Poser and Keele 1968, 1970; Reed 1972; Rosch 1973a, 1973b; Rosch, Simpson and Miller 1976). A probe entity can thus be considered, under many variants of such a model, to belong to the same category as that of the prototype to which it is the most perceptually similar. Members of a category may also be thought of as being connected by a series of overlapping similarities, something that is referred to in the literature as *family resemblance* (Wittgenstein 1953).

The prototype model overcomes many of the inconsistencies in the classical theory. For example, it is no longer necessary for each and every entity belonging to a category to share certain properties with all other entities belonging to the same category, as categories are not defined by such properties under the prototype model. Furthermore, entities may have differing amounts of typicality, based on their relative similarity to a prototypical exemplar, and boundaries between categories can be fuzzier and more flexible. The prototype theory can also help account for category ambiguity, such as encountered when subjects are asked to classify a tomato as a fruit or as a vegetable.

A number of experiments support this prototype-based approach. It was found, for example, that prototypes were the first exemplars learned by children (Mervis 1980; Rosch 1973b), that prototypes are the most likely to be named when subjects are asked to list all members of a category (Mervis, Catlin and Rosch 1976) and that, when asked to verify whether a probe entity is a member of a category, the fastest *yes* responses are to category prototypes (Rips, Shoben and Smith 1973; Rosch 1973b).

A vertical hierarchical structure of categories is typically associated with exemplar-based models. The *basic* categories are those categories at the level that is most commonly processed by humans, and consist of categories that have common attributes but are not variants of one another (e.g., skirt, coat, pants, etc.). Broader *superordinate* categories are above these basic categories, and are defined by their function (e.g., clothes cover people). More specific categories, called *subordinate,* are below basic categories, and consist of categories that are variants of the basic categories (e.g., parkas, rain coats, wind breakers, etc.). To provide an additional example, animal→bird→robin is an example of the superordinate→basic→subordinate structure. Exemplar-based models also typically have a horizontal aspect to them as well.

Basic categories are considered to be of particular importance, a notion referred to as *basic-level primacy*. As summarized by Lakoff (1987), they are:

- The highest level at which a single mental image can reflect the entire category.
- The highest level at which a person uses similar motor actions when interacting with category members.
- The level at which subjects are fastest at identifying category members.
- The level with the most commonly used labels for category members.
- The first level named and understood by children.
- The level at which most of our knowledge is organized.

The earliest exemplar-based models assumed that each category has only one prototype, and that each category is distributed normally around its prototype. Classifications can thus be made by calculating the distance between an entity and each prototype in psychological space and then choosing the category whose prototype is closest to the entity. The closer an entity is to a prototype, the better an example of the category it is under this model. Essentially, degree of category membership is evaluated based on the similarity of an entity to a category's prototype.

Experimental evidence soon began to demonstrate that this model was too simplistic, however (Smith and Medin 1981). For example, the assumption that similarity is inversely related to psychological distance is not always correct (Tversky and Gati 1982), and it has been found that categorization performance can be affected if exemplars possess correlated properties (Ashby and Maddox 1990). It has also been repeatedly demonstrated experimentally that internalized exemplars beyond just the prototype can have an effect on categorization (e.g., Brooks 1978).

Exemplar-based models were therefore modified to allow categories to be defined by multiple prototypes rather than just one (Rosch 1975; Rosch 1978). For example, the bird category might have both robins and sparrows serving as prototypical exemplars. Some versions of this theory hold that there is still a single prototype that is more influential than other exemplars, while other versions allow multiple exemplars to be equally important, with no single central prototype. In either case, classification of a probe entity still involves comparing it with all stored exemplars of all categories and choosing the category that has the prototype that is the most similar to the probe entity.

The *Generalized Context Model (GCM)* (Nosofsky 1986) provides a variant of the exemplar-based model that, as usual, assumes that classifications are made by making comparisons between probe exemplars and prototype exemplars. However, special attention is given to the role of selective attention in the GCM. The similarity of two stimuli is assumed to be probabilistically related to psychological distance between the stimuli in perceptual space, with the implication that a given subject may respond differently to the same stimulus at different times.

There has been a break between exemplar-based models that assign particular significance to prototypical exemplars and models that assume that humans store and use many exemplars in a more equitable fashion that does not give primacy to prototypes. The first approach assumes that classification decisions are made based on similarity comparisons between probe entities and the various prototypes. The second approach assumes that classification is performed based on similarity comparisons with all stored exemplars. In the literature, variants of the former approach are often referred to as *prototype-based models,* and variants of the latter as *exemplar-based models.* However, the term *exemplar-based models* is also sometimes used to refer to both approaches collectively, so there is some ambiguity in the terminology. Contemporary supporters of prototype-based models include Minda and Smith (2001), for example, and supporters of exemplar-based models include Nosofsky and Zaki (2002).

One problem with most exemplar-based models is that they typically assume that humans perform similarity comparisons between an entity and every exemplar of each category when we are making classifications. Ashby (1992) questions this because of the large computational load required.

There are also a number of questions with respect to exemplar-based models that remain to be definitively resolved. For example, for models that include prototypes, how exactly are these prototypes determined? More precisely, how does one find the prototypes of a category if it is assumed that categories are defined by the prototypes? Although there are certainly possible solutions to such problems of category formation, such as the potential for a bottom-up clustering method (e.g., Cambouropoulos 2001), such models can be difficult to verify.

It is interesting to note that exemplar-based models have similarities to nearest neighbour computer classification algorithms (see Section 6.2.6.4). The similarity measurements that are part of exemplar-based models can also be said to have resemblances to stochastic local search techniques that are used in machine learning to optimize feature selections and measure distances in feature space appropriately (see Section 6.2.5).

### 2.2.3 General recognition theory

*General recognition theory (GRT)* (Ashby 1989; Ashby 1992) is a modified version of exemplar-based models that does not require the large number of mental similarity comparisons required by most unmodified exemplar-based models. The essential idea behind GRT is that humans divide perceptual space into regions and associate a category label with each region. Each region is separated by a decision boundary given by some function, and each probe entity is assigned the category of the region into which it falls. Each classification therefore involves first mapping the probe entity to the appropriate region in perceptual space and then assigning the category corresponding to this region. The linking of categories to abstract regions of perceptual space makes it unnecessary to compute similarity with all exemplars every time that a probe entity is presented.

In GRT, the decision regions in perceptual space are formed based on the union of, typically, multivariate normal probability density regions surrounding each exemplar. Each category can therefore be represented perceptually as the probability mixture of its stored exemplars.

One implication of GRT is that the similarity of two categories can be viewed as the proportion of each category's decision region that falls in the other's decision region, something that fits well with a vertical approach to categorization. For example, apples and oranges have little or no intersection as they are unlikely to be confused with each other, but they both overlap with the fruit category's decision region. Apples and oranges can therefore be said to be more similar to fruit than to each other.

Proponents of GRT (e.g., Perrin 1992) note that GRT is particularly successful in uniting similarity and preference in a way that does not falsely assume that similarity is symmetrical. This approach therefore arguably allows a deeper exploration of the relationship between identification, similarity, preference and categorization, while at the same time avoiding some of the weaknesses of other approaches.

It is interesting to note that decision-boundary models such as GRT have similarities to discriminant-based classification algorithms (see Section 6.2.6.6).

### 2.2.4 Assumptions and evaluation of classification models

Ashby (1992) has usefully noted that classification models tend to make three types of assumptions:

- *Representation assumptions:* These assumptions describe how contrasting categories and the stimuli are represented internally. Most categorization models assume that a stimulus can be represented as a point in multidimensional space, but differ in other respects.
- *Retrieval assumptions:* These assumptions describe the information that must be collected before a response can be made. For example, exemplar-based models assume that subjects compute similarities between stimuli and exemplars.
- *Response assumptions:* These assumptions describe how subjects select the particular category or categories to assign to stimuli.

Ashby provides a detailed description of each of the assumptions made by the different models and notes that, in practice, it can be difficult to accurately and precisely evaluate different classification models and the assumptions that they make in experiments.

### 2.2.5 Additional issues in classification theory

The issue of determinism is an important area of continuing debate in classification theory. Deterministic models assume that a subject given the same stimulus multiple times will classify the stimulus in the same way each time, whereas probabilistic models permit variation based on factors such as perceptual noise, cognitive noise, attention and experience. Probabilistic models typically incorporate perceptual noise and/or cognitive noise and use probability density functions in order to evaluate similarity and make classifications.

In practice, it seems that classification is often probabilistic if inter-category similarity is high (Ashby 1990; Ashby 1992). This can be the result of both perceptual noise and noise in one's internal model, something called *criterial noise*. One's memory can vary with time, just as one's internalized model of categories can change as one is exposed to new stimuli over time.

An additional area of debate is whether or not entities can in fact be represented as points in a multidimensional space, as assumed by exemplar-based models and GRT. It has been suggested that such a numerical representation of entities may be inappropriate (Sattath and Tversky 1977; Tversky 1977). The assumption of a geometric model imposes boundaries on the number of nearest neighbours that any point may have, based on the dimensions of the model. For example, a point in a one-dimensional space may only have up to two nearest neighbours. Although this upper bound does not appear to be problematic in perceptual space, it may be significant in conceptual space (Tversky and Hutchinson 1986).

The possibility of category overlap is another essential issue. This overlap can manifest itself in situations where an entity can be said to truly be a member of multiple categories at once, as well as in situations where the entity is in fact only a member of a single category, but differentiation is difficult or impossible with the given set of percepts. As pointed out by Ashby (1992), it is very unlikely that an individual will confuse vegetables and cars, but it is much more likely that a non-wine drinker will confuse similar grapes. Any successful model of classification should therefore account for category overlap.

There have been a number of proposals on how to accommodate entities that belong to various categories to varying degrees. *Fuzzy set theory,* for example, assumes that entities can be given a number from 0 to 1 indicating how much they belong to each category.

There are many other variants of the models discussed above, as well as several entirely different classification models. There is still no absolute consensus as to precisely which model most accurately reflects human behaviour, although some experiments have found that a number of different models perform similarly when applied to classification problems (e.g., Cohen 1992).

## *2.3 Insights from music psychology*

The information presented above in Section 2.2 describes psychological models and research relating to classification and categorization in general. This section describes research that is specific to music classification and musical similarity.

The various classification models described above can certainly be applied to music classification. For example, an adherent to one of the variations of the classical model might argue that we compare musical properties that we perceive, such as rhythmic patterns or the particular instruments being played, with our existing knowledge, conscious or unconscious, of the characteristics of a particular musical category. An adherent of an exemplar-based model, in contrast, might argue that we perform classifications by measuring the musical similarity of the music that we hear with the prototypical exemplars of various musical categories. As one might expect, most approaches to music classification are based upon variants of these two dominant overall models.

### 2.3.1 Music classification

Deliege (1996; 2001a; 2001b), for example, approaches music classification from the perspective of exemplar-based models. She does emphasize certain differences between music classification and other types of classification, however, such as the particular importance of time in musical classification relative to various visual classification tasks. She suggests that we abstract *cues* from music, by which she means salient[105] properties that are useful for making classifications. Deliege emphasizes the role of abstracted cues in categorization, and the underlying role that categorization plays in the progressive development of a mental schema for a musical piece.

[105] The *salience* of a property can be understood as the relative importance and relevance of the property with respect to performing a particular type of classification.

Deliege also suggests that *imprints* occur when cues are repeated and stressed. These imprints serve as landmarks that help humans perceive structure in music, and are strengthened and refined after repeated listening. Since human memory cannot store all cues over a long piece, it is therefore necessary either to average some of them out or to choose to remember only the ones judged to be the most significant. Deliege argues that imprints are essential in this respect, and that imprints can serve as prototypes such that we compare the similarity of future cues to previously formed imprints in order to measure similarity and perform classifications.

Deliege notes that it is known that melodies retain a degree of similarity if contour and rhythmic organization remain unchanged, even if the tempo, key or size of the intervals are altered. Since melodies can still be recognized even after such transformations, it would seem that transformations that leave the abstracted cues that are relevant to a particular kind of classification intact do not tend to interfere with that particular kind of classification. However, if transformations are made that do significantly change abstracted cues that are relevant to a given type of classification, such as melodic contour or rhythmic organization in the case of melody recognition, then the ability to recognize the melody is impaired.

Deliege suggests that we automatically segment music by looking for changes in adjacent cues. Segmentation of sections is likely to occur when abstracted cues start to differ from what was just heard previously, and musical phrasing and rests may help this segmentation process.

Once this segmentation is complete, we can then perform similarity measurements by comparing the abstracted cues from different sections of the music in such a manner that both the similarity and the dissimilarity between different sections can reinforce or inhibit, respectively, the overall sense of similarity. This allows us to perform tasks such as thematic recognition.

Although music theorists often consider musical structure from a perspective that encompasses each musical piece in its entirety, Deliege argues that human memory is limited, and that it is unlikely that human classification in fact emphasizes musical properties that develop over the long term. Furthermore, humans have been shown experimentally to be able to segment and classify music after hearing only very short segments (Perrott and Gjerdingen 1999). It therefore seems much more likely that listeners form cue abstractions and groupings in the short term as they listen.

Lerdahl and Jackendoff (1982) have also formed a model of how musical groupings are made, based on gestalt principles of proximity, similarity, common fate, closure and good continuation. Other research emphasizes the role of rhythm (Cooper and Meyer 1960) and of melodic contour (Deutsch 1999) in forming groupings.

Research seems to indicate that different musical backgrounds can cause certain properties to be emphasized over others when forming groupings. Non-musicians, for example, tend to focus more on intensity and register, while musicians are more likely to focus on motivic aspects (Pollard-Gott 1983). Thorisson (1999) also found that novices base categorization of Romantic and Classic piano on texture more than chord progressions. Deliege and Thorisson also both note that perceived similarity and categorization may be dependent on culture and personal experience.

Overall, Deliege draws a parallel between Rosch's approach to categories and her own abstracted cues and imprints. Horizontality can be associated with the variations developed from a single cue, such as the variations of a motif, where these variations generate an imprint that is analogous to a prototype. Various types of cues, in turn, can be organized vertically into hierarchies.

Ockelford (2004) has modified and built upon Deliege's approach using an approach that is informed by both psychological research and music theory. He puts a particular emphasis on the notion of derivation and on the importance of mental links we form between chunks of music. He also notes that, when observing themes and variations, listeners must notice sufficient difference in order to tell differences in parts of piece, but there must still be enough similarity for the piece to have coherence.

Ockelford models music as being conceptually divided into *frames,* or slices of time, each with their own set of *perspects,* or properties. Any frame may be compared with any other frame using each of their associated perspects. We can also form mental relationships between different perspect values in different frames, something Ockelford calls *interperspective relationships*. If enough perspects match for any two frames, then they can be judged to be similar and, if all perspects deemed to be salient are the same, then the frames are perceptually identical. Ockelford notes that such judgments can vary with attention, context and experience.

Ockelford suggests modifying Deliege's model by introducing the notion that one musical element can be considered to derive from another element. If one perspective value is thought to be an imitation of another, then the second is considered to be a derivative of the first. Ockelford calls the interperspective relationships that can indicate derivation *zygonic relationships*. *Zygons* linking perspect values that are perceived to be identical are *perfect* and zygons linking similar values are *imperfect*. A sense of imitation can occur even if the derivative is only similar but not identical to the original. Under this system, the strength of the sense of derivation varies based on the perceptual salience of the values concerned.

Ockelford argues that such an understanding of derivation makes it possible to form larger mental models of structure, and that zygonic relationships can be used to describe how perceptual groupings relate to one another. He notes that, based on Ockham's razor, it may be best to assume that music is modeled in cognition using the simplest and fewest possible mental processes.

Cambouropoulos (2001) provides a good summary of categorization theory applied to music. In particular, he emphasizes the position that cue abstraction, similarity and categorization are inextricably bound together. He also stresses that issues of property salience are particularly important with respect to music classification. It is interesting to note that the salience of various properties with respect to different kinds of classification can also be an important issue in automatic computer-based classification systems. Unfortunately, the psychological study of salience is often complicated by context-dependence.

Cambouropoulos also points out that *expectation* can potentially play an important role in music classification. For example, if one is hearing music performed by a musician who is known for performing Classical music, one may be more likely to classify the music that is actually heard as Classical than if this *a priori* knowledge of the performer were absent.

Cambouropoulos also emphasizes a number of issues that further complicate matters. For example, although all members of a category may arguably be said to be similar, not all similar entities will necessarily share a category. It also seems likely that we constantly update our notions of similarity, our ontology of categories and the properties that are salient when performing classifications. Furthermore, the perceived similarity of entities can depend upon the contexts in which they are found, just as other aspects of similarity and classification can also be context-dependent.

Cambouropoulos proposes a model for category formation that is intended to address such issues. This model is based upon a clustering algorithm called *unscramble* that generates a set of categorizations given a set of entities. Two entities are defined as similar if the perceptual distance between them is below some threshold, and a category is defined as a set of entities where all entities are pair-wise similar. This approach permits overlapping classes, although it has the disadvantage that it does not account for categories that encompass several distant discrete clusters in perceptual space.

In order to address salience, the unscramble algorithm assigns weights representing salience to properties. Higher weights are also given to properties that are unique to a category, and lower weights are given to properties that are shared by multiple categories.

Prototypes are calculated once clusters are formed by the unscramble algorithm, and these prototypes are then used to perform classifications, as normally assumed by examplar-based approaches. Cambouropoulos reports that this algorithm performed well in simulating human behaviour, although he acknowledges that this does not necessarily prove that humans perform operations similar to the unscramble algorithm.

Although there is convincing evidence that music categorization cannot be explained by a purely classical model (e.g., Deliege 2001a.; Ziv 2007; Volk et al. 2008), it has been suggested by some (e.g., Lamontand and Dibben 2001) that perhaps some researchers have been a little too hasty to concentrate only on exemplar-based models of musical classification. It has been suggested that perhaps it would be best to consider both perceptual and conceptual elements and to use models of categorization that incorporate elements of both classical models and exemplar-based models.

For example, some cognitive psychologists argue that similarity may be too unconstrained to ground categories (Thorisson 1999), as is required by most purely exemplar-based models. Furthermore, similarity judgment and classification may each involve different properties and different weightings. Although similarity judgments based on differences in lists of properties likely do play a potentially important role in classification, it is also possible that classifications are performed partly based on higher-level analytical knowledge as well. Additionally, classification may also potentially affect similarity, as noted above, such as in the case of a listener who can identify the genre of a given piece and then use this information as part of future similarity judgments between this piece and others.

There is some empirical basis for such a hybrid approach, such as the experiment by Hortaçsu and Tekman (2002) that indicated both that style categories seem to be defined more by casual principles than by lists of properties, which supports a classical approach, and that style categories follow a hierarchical organization, which is more indicative of an exemplar-based approach. Another study found that listeners provided with some basic analytical knowledge were able to classify music into categories better than listeners without the same analytical knowledge (Thorisson 1999). So, even though it seems clear from the large body of experimental evidence that a purely classical model is unlikely, it is also seeming increasingly likely that a purely exemplar-based model may not be correct either.

### 2.3.2 Musical similarity

Exemplar-based classification models typically fundamentally assume that humans base classifications upon judgements of the similarity between probe entities and exemplars. As emphasized by McAdams and Matzkin (2003), similarity judgements also underlie other essential musical operations, including perceived invariance under musical transformation, music recognition and the perception of familiarity in music.

There are many perceptual cues that one could potentially use to judge musical similarity, including statistical properties extracted from the auditory data, structural information collected during the course of listening to a piece of music and judgments of how well the music matches one’s expectations of the grammar that the music is understood to have.

There is some debate on the relative importance of *surface properties* (e.g., melodic contour, statistical pitch distribution, dynamics, accents, note durations, rhythm, timbre and performance techniques such as trills, vibrato and portamento) and *deep properties* (e.g., harmonic structure, metric structure and thematic variations). Related to this, McAdams and Matzkin (2003) have postulated three types of potentially overlapping types of similarity:

- Statistical distribution of surface properties, their derivatives, the relations between them and, potentially, transitional probabilities.
- Figural similarity, which is to say patterns of properties that provide perceptual landmarks within long pieces of music. These landmarks will be perceived as invariant even after limited transformation of the audio signal. This idea is somewhat similar to Deliege's imprints.
- Structural similarity, which relies on the abstraction of structural invariants. This may be based on underlying harmonic and metric templates.

Although it is certainly possible that both surface properties and deep properties play some role, there is some evidence that seems to minimize the role of deep properties. For example, Bigand and Tillmann (1996) performed an experiment where recordings were divided into sections that were then played in reverse order. It was found that there was little perceived difference in expressivity or coherence by test subjects. In contrast, surface properties such as loudness, duration and pitch contour were found experimentally to have a significant role in influencing similarity judgments. In particular, chromatic pitch alterations and changes to note durations seem to be particularly important in creating a sense of dissimilarity (McAdams and Matzkin 2003). Eerola and his colleagues (2001) found that properties based on the frequency of musical events played a moderate role in influencing similarity judgements, and descriptive variables such as the number of tones, rhythmic variability and melodic predictability were more significant.

It has generally been noted that transposition and tempo changes have relatively little effect on perceived similarity (McAdams and Matzkin 2003). Even a few small interval changes do not interfere too much with the perception of similarity if certain prominent intervals that maintain melodic contour are maintained (McAdams and Matzkin 2003). Having noted this, such changes can still play some role in perceived similarity, which has led Hoffman-Engl (2001) to suggest the use of *melotons,* which are experimentally determined by studying pitches perceived by subjects, rather than pitches defined by physical frequencies when studying melodic similarity.

It also seems that the more changes to salient properties are made, the more dissimilar two musical segments will sound. Experiments have found that a change to both pitch and rhythm creates a greater sense of dissimilarity than a change to just one or the other (McAdams and Matzkin 2003).

Potentially related to this, issues relating to auditory streaming (Bregman 1990) can also be very important. If a segment of music is altered enough so that it is broken perceptually into multiple streams, such as by varying the timbre of every note, then the perceived dissimilarity is likely to increase significantly (McAdams and Matzkin 2003).

Another important issue is that the perception of similarity between two musical excerpts can depend on what is heard between them. If intervening audio information that is similar itself to the first excerpt is played in between the two excerpts then this may degrade the perception of similarity between the first and final excerpts. This effect appears to be modular in that, even if some dimensions of the intervening material are similar, if other dimensions are different then the perception of similarity between the first and final excerpt will not be degraded as significantly as would be the case if all dimensions of the intervening material were similar (McAdams and Matzkin 2003).

This issue of modularity (Fodor 1983), which is to say whether or not the dimensions of perception are processed independently, is itself an important issue in general with respect to music. According to Fodor's modularity thesis, the cognitive system contains multiple specialized subsystems. The central processing system is specialized in association, symbolization and metaphorization and other subsystems specialize in the detection of invariants and the formation of percepts and gestalts. Although it is not entirely clear whether music perception follows this model (Leman 1995), the results referred to by McAdams and Matzkin do seem to indicate some level of modularity with respect to pitch and duration processing, albeit not complete independence.

Several experiments have been performed relating to comparisons of the similarity judgments made by musicians and non-musicians, just as experiments have been performed on their respective categorization behaviours, as described in Section 2.3.1. Several of these similarity experiments have indicated that listeners of both types tend to make similar similarity judgements (e.g., Lamont and Dibben 2001), although sometimes for different reported reasons. There is some experimental evidence that musicians tend to consider structural properties such as motivic and harmonic relationships more than non-musicians, whereas non-musicians focused more on surface properties such as dynamics, articulation, texture and contour (Pollard-Gott 1983; Lamont and Dibben 2001). This may indicate that there are parallel approaches using different properties that can be used to arrive at the same similarity judgments.

Somewhat different results have been found in a sequence of experiments with musician and non-musician children between ten and eleven years old (Koniari, Predazzer and Mélen 2001). It was found that musician children performed better classifications than non-musician children on one piece but not on another piece. It was further found that similarity evaluations were made based on both surface and deep properties, and that different children segmented differently from each other, although each child tended to segment music in a way consistent with him or herself, particularly in the case of the musician children. The fact that even the musicians in this group were still relatively young children may account with the contrast with other studies.

It can be difficult to determine which properties are being used by a particular listener at a particular time, as these properties can be dependent on the listener's experience, the type of music that he or she is listening to, his or her mood, the situation that he or she is in and where his or her attention is focused. Salience is a core issue here, as different listeners may judge different properties to be salient at different times. Personal preference must also be considered, as a study of Turkish undergrads found that subjects tended to find different pieces more similar if they liked them than if they did not like them (Hortaçsu and Tekman 2002).

Lamont and Dibben (2001) have noted that experiments have revealed four ways in which context can affect how listeners perceive similarity and perform classifications:

- *Experience:* Children focus on factors such as loudness, whereas adults focus more on melody-based properties. Sensitivity to rhythm also increases with age.

- *Familiarity:* Surface properties such as melodic contour, loudness and texture dominate when a listener is unfamiliar with a piece. Structure is more important when users are more familiar with a piece, however.
- *Complexity:* The types of properties that are salient depend on the complexity of the music. Structure is more easily perceivable for simple music, for example.
- *Task complexity:* Thematic relationships are more easily extracted in less cognitively demanding situations

A further complication that must be recognized by a successful model is that similarity is not symmetric (Tversky 1977). For example, a contemporary composer might decide to write a piece that can be said to be very much like Beethoven's *Symphony No. 5 in C minor*, for example, but Beethoven's symphony would not be said to be like the new composition to the same degree. In addition, musical similarity can violate the triangle equality (Tversky 1977). For example, Dave Grohl is like Nirvana and is also like the Foo Fighters, but Nirvana and the Foo Fighters are not as similar with one another. Tversky suggests that such asymmetries may be addressed by using a measure of the similarity between two entities based on a function of their shared properties minus the properties that are distinctive to either of them.

One can also take the approach that similarity can be viewed as the distance between two entities in perceptual space, which ties in with exemplar-based classification models. Perceptual salience can be accounted for by weighting different properties differently when making similarity judgments. If two entities have a perceptual distance between them under some threshold, then they can be said to be similar. If all salient properties are the same, then they can be judged to be identical. As noted by Krumhansl (1978), issues of asymmetry can be addressed if one sees similarity from this perspective and takes the density of entities in one's cognitive space into account. Two points in a dense region of entities could be said to have a smaller similarity than two points equally far apart in a less dense region of cognitive space.

As a final note, it is to important to note that the vast majority of the psychological experiments upon which the models discussed above are based have been performed on North American or European listeners who are accustomed to listening primarily to

Western music. It may well be that the properties that have been found to be dominant in judging musical similarity are influenced by musical culture and training, and that listeners trained primarily in non-Western musical traditions may emphasize different properties in their similarity judgements and classification decisions.

## 2.4 Musicological and music theoretical insights

The research and models described in Sections 2.2 and 2.3 are largely based upon experimental data and relatively low-level models that are supported by this data. Musicological and music theoretical research, in contrast, tends to take a higher-level approach that is less based on controlled experiments with human subjects and more based on direct studies of music itself and on more general sociological research, such as the reception of music. Although there has been relatively little musicological and music theoretical research performed directly on the mechanisms used by humans to classify music, at least compared to the scope of the psychological literature, there is nonetheless some very useful work that touches on the subject, and it useful to highlight some of it here.

Another important distinction between the psychological research and research in the humanities is that a significant proportion of psychological research has focused, often of necessity, on classification behaviour and similarity judgements in response to small and carefully controlled groups of audio signals. There has been much less psychological research on musical classification behaviour and similarity judgements on a greater scale and with a greater scope. For example, how do humans classify and measure the similarity between different performers' playing styles, between different albums by the same performer or between different genres of music? Such judgments each require decisions that span multiple pieces of music, and thus may potentially operate on entirely different levels and using different mechanisms than comparisons between just a few well-controlled audio signals. Internalized abstracted symbolic and cultural representations and models, for example, may be just as significant here as relatively short-tem perceptual and cognitive representations of audio signals. Fortunately, there is some musicological and music theoretical research that has been performed that may help to fill in the gaps.

Lerdahl and Jackendoff (1982) suggest that listeners perform a reduction of music based on their *well-formedness* rules. It might be argued that this reduction serves as the mechanism by which we store information about music, since there is too much information in the raw audio signal for it to be stored in its entirety over the long term. This reduction could then provide the information about music that is used for performing classifications and similarity comparisons, whether one prefers classical or exemplar-based models.

Temperley (2001) has proposed *preference rules* that are in some sense an expansion on Lerdahl and Jackendoff's well-formedness rules. Temperley proposes these preference rules as a way of reducing representations of music and as a way of evaluating the goodness of various possible analyses of music. He further suggests that perhaps humans have internalized such rules and can judge the similarity between pieces based on how well they each meet the requirements of various preference rules. Different genres of music will have different preference rules, and how well the particular preference rules for a given genre are fulfilled by a given musical piece could provide a basis for classifying the piece. Although this approach does seem intuitively reasonable, and could certainly be experimented with as a tool for computer-based classification, Temperley acknowledges that there is only limited experimental support for this model.

Related to this, Fabbri (1981) has suggested that we form *hyper rules* to create different hierarchies of importance associated with different discriminants that depend on the particular characteristics of the music under consideration. For example, if one recognizes that a certain song is by a certain performer who is associated with certain styles and genres, this may affect subsequent similarity and classification judgments. This ties in with the importance of expectation, as emphasized by Cambouropoulos (2001) and others, as described in the sections above.

This reemphasizes the importance of not focusing exclusively on sonic cues when considering how humans classify music. Cultural and social characteristics of music beyond the scope of audio signals can also be of as much or more importance.

Musical genre represents one of the most fundamental ways in which humans categorize music, and serves as in interesting case study. Genre is strongly linked to the social, economic and cultural backgrounds of both musicians and listeners. Both of these groups tend to identify with and associate themselves with certain genres, with the result that their behaviour is influenced by their preferences. Or, viewed from a different perspective, many people have strong identifications with social and cultural groups that are associated with certain musical genres. In either case, there is often a strong correlation between musical genre preferences and personal appearance and behaviour. One need only see a photo or watch an interview with a musician, without ever having heard his or her music, to be almost certain whether the musician plays rap, heavy metal or classical music, for example.

The style of the album art, web pages and music videos of musicians all provide non-sonic cues that humans can use to classify music. Similarly, as noted above, a performer's appearance and actions on stage (facial expressions, ritual gestures, types of dancing, etc.) provide further cues, as do an audience's demographics, dress and behaviour (clapping, shouting, sitting quietly, dancing, etc.). The fine musical distinction between some sub-categories may well be related to such social and cultural properties more than musical content itself.

The writings of Franco Fabbri are particularly influential with respect to these types of issues, particularly in relation to musical genre. Fabbri discusses the links between genre and social, economic and cultural factors and how genres come into being in one of his early papers (Fabbri 1981). Fabbri continues this discussion in a slightly later paper (Fabbri 1982). He also presents a discussion of the issues related to musical categories, how the mind processes them and their importance in general in a more recent paper (Fabbri 1999). Of particular interest, Fabbri (1981) argues that musical genres, and by extension, perhaps, musical categories in general, can be characterized using the following types of rules, of which only the first is related strictly to musical content:

- *Formal and technical:* Content-based practices that are typically followed by members of the genre.
- *Semiotic:* Abstract concepts that are communicated (e.g., emotions or political messages). These can also relate to information conveyed by ways of dressing and acting and what they symbolize.
- *Behaviour:* How composers, performers and audiences appear and behave.

- *Social and ideological:* The links between musical categories and demographics such as age, race, sex and political viewpoints.
- *Economical and juridical:* The laws and economic systems supporting a musical category, such as record contracts or performance locales (e.g., cafés or auditoriums).

In particular, Fabbri suggests that if one views categorization according to the classical theory then it can be said that musical discriminants are learned as part of enculturation and education. If one espouses exemplar-based models, then one might say that individuals also learn prototypical exemplars via supervised learning provided, again, via enculturation and education. So, Fabbri suggests, in either case each culture provides its members with the general bases for performing musical classification, although there can be some variance in rules or exemplars from individual to individual. This means that one should not see categories and the rules or exemplars that define them as static and absolute, but rather as constantly shifting products of culture. In addition, as we collectively adapt our conceptions of categories based on our cultural surroundings, our society also collectively updates itself based on changes in individuals' conceptions, a bidirectional feedback process (Fabbri 1982; Fabbri 1999). Fabbri also suggests that new musical genres are originally formed as rules of existing genres begin to become consistently broken in consistent ways over time (Fabbri 1981).

Fabbri builds upon Umberto Eco's contention that categories do not tell us what a thing is, but how to put it into a system of concepts. Our awareness of the kind of music that we are listening to thus influences how we listen to it, something that underlines the importance of musical classification to musical experience in general (Fabbri 1999).

Frith (1996) explores related topics by presenting a discussion of the types of social and cultural factors that can affect how category distinctions are formulated and what their meaning is. Toynbee (2000) provides an interesting discussion of how genres inform musicians, of the influences of identifications with different communities and of the music industry.

Brackett has also done some very interesting work on musical genre, including a discussion of how the ways in which particular genres are constructed and grouped can vary in various charts, radio formats and media fan groups, and of issues relating to recordings crossing over from one set of groupings to another (Brackett 2002). Brackett has also written a good resource for those trying to deal with the task of characterizing genres (Brackett 1995).

It is important to emphasize that a great deal of theoretical work has been done on genre in fields other than musicology. Much of this work provides insights that are not only applicable to musical genre, but to musical classification in general. The collections of works edited by Duff (2000) and Grant (2003), for example, are excellent resources on literary genre and film genre, respectively.

Lyrics are another type of cue that can be particularly important in classifying music. Although lyrics are of course contained in sonic audio signals, they are often treated separately from more "musical" cues in the literature, and are often given relatively little attention by musicologists. Nonetheless, lyrics can be very important to many listeners. Content (e.g., love, political messages, etc.), rhyming scheme, vocabulary, use of clichéd phrases and use of characteristic slang, for example, all likely provide useful indications of genre.

It is for the reasons outlined above that jWebMiner (see Chapter 5) is included in jMIR. It is able to extract cultural information not stored in sonic cues, and complements the more traditional types of features extracted by jAudio (see Chapter 3) and jSymbolic (see Chapter 4). There are also plans in the future to write further jMIR tools to access even more of the information that can be useful for classification, such as a jLyrics feature extractor that will extract features from textual lyrics, and a jAlbumArt, which will extract features from images of album art.

There is a significant amount of additional relevant musicological and music theoretical research that is presented in other chapters of this dissertation. An area of particular interest, from both theoretical and practical perspectives, is the ways in which humans form, interrelate and organize categories. These issues are discussed in Section 8.2, as they are particularly relevant to the formulation and organization of musical collections.

The particular sonic cues that humans use to classify music have also been considered in the humanities as well as in psychological research. From the point of view of automatic music classification, this is something that is directly pertinent to feature extraction. These issues are consequently discussed in Section 4.4.

# 3. jAudio: Extracting features from audio

## *3.1 Overview of audio feature extraction and jAudio*

### 3.1.1 Introduction

Audio feature extraction is the process of automatically extracting characteristic information about audio files which can then be used for purposes such as classifying and analyzing audio. As discussed in the various sub-sections of Section 3.2, this involves a number of steps, including parsing and potentially decompressing the audio samples from the file in which they are stored, pre-processing the samples so that they are appropriately prepared for the particular features to be extracted and, finally, extracting the features themselves, which may be calculated based on the basic samples, other features that have already been extracted, or both.

jAudio is the jMIR component devoted to extracting features from audio data, and it performs all of these steps. It is designed to be used directly as a simple audio feature extraction software application as well as a platform for iteratively developing new features that can then be shared amongst researchers.

Audio feature extraction has to date commanded significantly more attention in the MIR community than either symbolic or cultural feature extraction. As a consequence, there are already several well-established audio feature extraction software systems designed for use in MIR research, as is discussed in Section 3.3.2. Reasons for this emphasis on audio feature extraction include, among other factors, strong commercial interest in technology for processing digital audio and, as discussed in Section 3.3.1, a large body of existing related research in speech recognition.

MIR researchers have been quick to make use of existing and easily accessible applied work in non-music related areas of audio research, something that has in many ways benefited MIR research. Unfortunately, this has, in some cases, been done without a full awareness of the relevant digital signal processing theory. Although quick and encouraging early results have been achieved in many cases, using approaches that are not fully informed can have the ultimate consequence of limiting long-term performance.

To give one example, many MIR researchers have been quick to use Hamming windows (see Section 3.2.3) when extracting frequency domain information from audio signals, likely because this is a common approach in speech analysis applications. However, although appropriate when dealing with signals such telephone speech, for example, this may not always be the best approach when dealing with musical audio. To give another example, relatively little attention has been paid to the potentially critical issue of feature dependence on audio encoding schemes, as discussed in Section 3.2.5.

This chapter therefore places a greater emphasis on background information than some of the other chapters. Although naïve approaches to signal processing can produce acceptable results in some cases, performance will ultimately be improved if one has the necessary background knowledge to properly parameterize feature extractions and design effective new features.

The wealth of existing work on audio feature extraction has also influenced the design priorities of jAudio. Much of the work in producing jWebMiner and jSymbolic was devoted to developing a basic framework for analyzing data and designing and implementing specific features.

The existing research and software relating to audio feature extraction made it possible to more fully develop jAudio than some of the other jMIR components. In particular, this involved adding functionality for testing new features, including basic recording, sound synthesis and visualization tools, as discussed in Section 3.4.9. It also made it possible to put an even greater emphasis on making it easy for developers to design and implement features within the jAudio framework, as discussed in Sections 3.4.5 to 3.4.8. jAudio also provides users with a particularly broad range of interface choices for extracting features, including a simple and easy-to-use GUI, a command-line interface for batch processing and an API for embedding jAudio's functionality in other applications. This GUI is described in Section 3.5.

### 3.1.2 Iterative feature development

Designing new features to extract from audio signals can be a particularly problematic task. A typical, or even musically trained, person might have difficulty expressing a precise list of characteristics when asked to distinguish between two different sounds, even if he or she can easily differentiate between the sounds. Even when one is able to describe audio characteristics, these features are likely to be abstractions that are difficult to quantify and scientifically extract from audio signals. For example, a musician asked to describe the differences between two genres of music is likely to use terms that require the ability to extract information based on pitches, rhythms and timbres.

Unfortunately, high-level information such as this is currently difficult or impossible to reliably extract from general music signals. This difficulty means that one must at least start with low-level signal processing-oriented features. Determining which such low-level features are best-suited for any particular task can be difficult, as humans do not tend to think about sound in terms that are meaningful in a low-level signal processing sense.

Fortunately, there are successful approaches to dealing with this problem. One can take an iterative approach to feature extraction, where low-level features derived directly from audio signals are used to derive mid-level representations, which can in turn be used to derive increasingly high-level features that are musically meaningful to humans. For example, basic spectral and amplitude features can currently be used to track note onsets and pitch in the special case of monophonic music, which can in turn be used to generate MIDI transcriptions, which can then be used to generate high-level features relating to rhythmic patterns and melodies. It is important to note that the degree of accuracy needed for automatic transcription is not necessarily required for features intended for classification. Individual incorrect notes, for example, can be averaged out through the construction of intermediate data structures such as beat histograms or pitch histograms. Furthermore, although low-level features are not usually intuitive to humans directly, an individual well-trained in signal processing and in auditory perception can use his or her expertise to gain insights into when certain low-level features can be useful even on their own.

An additional important point is that, even though it might be the case that some low-level feature is not generally used by humans to perform a given type of classification, this does not necessarily mean that this feature cannot effectively be used by a computer to perform the same type of classification. Feature selection techniques, which are statistical procedures for eliminating unpromising features for the purpose of improving classifier speed and accuracy, can be used to experimentally determine which features are useful and which are not in a given context.

Such an explorative approach is particularly important with respect to music classification. Although a great deal of research has been done on features that are useful for speech recognition, music has received much less attention. Experimental research on which features work well with respect to different kinds of music in different contexts could be of theoretical interest in and of itself, in addition to improvements in classification success rates that might also result.

In any case, it is clear that the acquisition of a large number of low-level features can be very useful, for the purposes of theoretical research, direct applied classification and constructing more abstract higher-level features. Software that can easily do this is therefore essential for audio classification, as is a good platform for iteratively designing new features.

A consequence of the past emphasis on low-level features in audio feature extraction is that tasks such as tempo extraction or chord identification are often treated as discrete MIR goals. In actuality, it can be appropriate to also think of the results of such processing simply as features that can themselves be used for other tasks, such as song identification or playlist generation, once again with the goal of iteratively developing increasingly high-level features. Indeed, although one certainly prefers perfectly reliable feature values, many machine learning-based approaches can deal with a certain amount of input noise, so such tasks/features can potentially be used effectively in higher-level tasks even before they are fully perfected.

A key design goal behind jAudio is to provide an infrastructure for facilitating these kinds of approaches to iterative feature development and reuse. An approach developed specifically for note onset detection, for example, could also be distributed and used by other researchers interested in audio transcription. So, while the majority of the features packaged with jAudio are in fact low-level features, it is hoped that they will be used to develop other more high-level features.

### 3.1.3 Contributions to jAudio

The work in both designing and implementing jAudio was shared with Daniel McEnnis. jAudio originated as two independent projects, one by the author of this thesis and the other by McEnnis. These two projects were merged into a collaboration prior to either of the original publications on jAudio (McEnnis, McKay, Fujinaga and Depalle 2005; McEnnis, McKay and Fujinaga 2006a).

McEnnis has been responsible for the significant majority of the work in improving, maintaining and supporting jAudio since these initial publications.

Most of the audio processing performed by jAudio is custom-implemented. However, jAudio does make use of four existing freely distributable third-party libraries to perform certain tasks, namely audio file parsing, sample rate conversion, calculation of MFCCs, calculation of LPCs and XML file parsing. These libraries, some of which are used to perform more than one of these tasks, are the Tritonus Open Source Java Sound library,[106] Ocvolume,[107] Sun's Java MP3 Plugin[108] and the Xerces XML Parser.[109]

### 3.1.4 Downloading jAudio

The jAudio Java bytecode and the associated source files are freely available at jaudio.sourceforge.net and at jmir.sourceforge.net/index_jAudio.html. Note that a number of independent changes may have been made by Daniel McEnnis to the most recent versions of jAudio that are not incorporated into this chapter.

The original jAudio prototype that was designed independently by the author of this thesis before it was merged with McEnnis' work is also available for historical reasons, although it lacks much of the functionality described in this chapter.

## *3.2 Background information*

This section is intended to provide background information on digital audio, digital signal processing (DSP) and specific features that are often extracted from audio files. Readers who are already knowledgeable in DSP may wish to skip directly to sub-sections 3.2.6 to 3.2.8, which deal directly with features that can be extracted from audio.

For readers less familiar with DSP, sub-section 3.2.1 provides a quick basic introduction to digital audio. Sub-section 3.2.2 then proceeds to describe Fourier analysis and the frequency domain, which is to say that it discusses digital audio from the

[106] tritonus.org
[107]ocvolume.sourceforge.net/ocvolume.php
[108] java.sun.com/products/java-media/jmf/mp3/download.html
[109] xerces.apache.org

perspective of its frequency content. Sub-section 3.2.3 discusses windowing functions, which are an important tool in performing effective Fourier analysis, and sub-section 3.2.4 covers other types of pre-processing that can be usefully applied to audio before extracting features. Sub-section 3.2.5 then provides a brief overview of some different audio formats and compression schemes, and discusses related problems relevant to feature extraction.

### 3.2.1 Digital audio

For the purposes of MIR research, features are typically extracted from audio in digital form rather than in analog form, as the vast majority of music consumed in the West is distributed in digital formats. It is therefore useful to briefly provide some pertinent background on digital audio.

A continuous analog signal can be represented in digital form by measuring the signal level of the analog signal at regular time intervals. This results in a sequence of discrete numbers that approximate the original analog signal, as shown in Figure 3.1. Each of these discrete numbers is known as a *sample,* and the value of each sample effectively represents the average of the signal level over the duration of the sample.

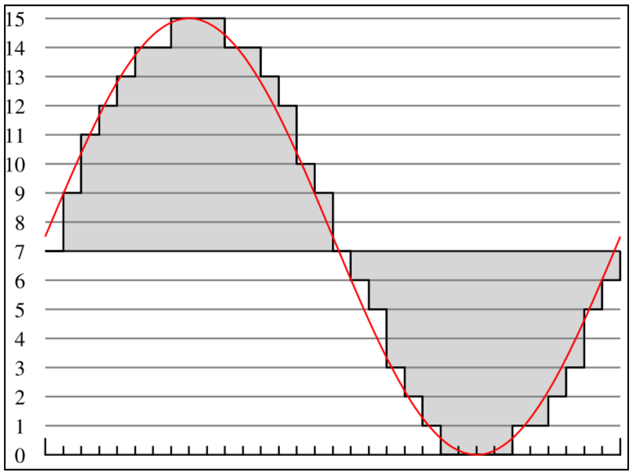


**Figure 3.1**: Sampling of one period of an analog sine wave (the continuous curve) into a digital signal (the step function).[110]

[110] This image belongs to the Wiki Commons and is included here under the GNU Free Document License and Creative Commons Attribution ShareAlike License. It was obtained from http://en.wikipedia.org/wiki/File:Pcm.svg.

It is appropriate to clarify the notation used in this chapter. The instantaneous signal strength of an analog signal is notated as a function where the input variable of time, *t*, is enclosed in parentheses (e.g., *f(t)*). Digital signals, in contrast, are represented by enclosing the input variable (be it *t* or the sample number, *x*) with square brackets (e.g., *f[t]* or *f[x]*).

Sampling involves a number of parameters, two of the most significant of which are the *sampling rate* and the *bit depth.* The sampling rate refers to how many samples are measured per second during digitization. The bit depth indicates how many bits are used to represent each sample. Higher sampling rates and bit depths generally result in better quality audio, but also result in the need for more sensitive instruments and greater storage space. To provide context, CD audio has a sampling rate of 44.1 kHz and a bit depth of 16 bits.

Based on what is known as the *Nyquist Theorem,* a digital signal can only represent frequencies that are no larger than one half of the sampling rate. Digitizing a signal with frequencies higher than half of the sampling rate results not only in the failure to encode these frequencies, but also in the addition of undesirable artefacts of the sampling process, referred to as *aliasing*. Low-pass filters are therefore typically applied before sampling occurs in order to remove any problematic high frequencies.

It is often assumed that sampling rates as low as 44.1 kHz produce acceptable quality audio because frequencies above approximately 20 kHz are outside of the range of human hearing. Some, however, have argued that higher sampling rates nonetheless result in perceptibly improved audio quality.

The choice of bit depth influences the dynamic range of a digital audio signal as well as the amount of signal distortion, known as *quantization error.* The greater the number of bits, the greater the number of signal levels that can be represented per sample. For example, a bit depth of 3 bits only allows $2^3=8$ different signal levels to be represented, whereas bit depths of 16 bits and 24 bits (DVD quality audio) permit 65,536 and 16,777,216 values, respectively. These discrete signal levels must be spread over a dynamic range defined by each particular audio format.

Any input signal that is stronger than the largest value that can be represented results in a form of signal distortion known as *clipping*, something that provides reason for choosing a high dynamic range. However, a larger dynamic range with a fixed bit depth results in larger spacing between each of the representation levels, with a corresponding larger average rounding error when analog levels are translated to digital levels, the essence of quantization error. Many audio formats use non-linear mappings to optimize perceptual audio quality with a given bit rate.

### 3.2.2 Fourier analysis

With the exception of artificially generated sound waves, all audio waves contain a *spectrum* of many different frequencies, each with its own amplitude and phase. It is possible to use a process known as *Fourier analysis* (which uses the *Fourier transform*) to decompose a wave into its component frequencies and phases. This allows signals to be considered in the *frequency domain*. A wave represented in the frequency domain can also be transformed into the *time domain* (via a process known as the *inverse Fourier transform*), which is the more generally familiar representation of a wave's instantaneous signal strength as a function of time.

Fourier analysis makes it possible to extract a great many useful features, and is therefore an essential part of any audio feature extraction system. There is no loss of data or signal corruption when the Fourier transforms or inverse Fourier transforms are applied.

It is possible to apply Fourier transforms to waves represented digitally via a process known as the *discrete Fourier transform (DFT).* A DFT results in a frequency histogram and a phase histogram. The phase histogram is most often not given significant attention in the extraction of audio features, but is essential for reconstructing the time domain representation from the frequency domain representation. The frequency histogram essentially consists of a set of bins, each corresponding to a different range of frequencies. The value corresponding to each histogram bin corresponds to either the combined magnitudes or the combined powers of all sinusoidal components of the wave in question with frequencies that fall within the bin's limits. In essence, a DFT outputs a vector of relative frequency strengths that is the same as a vector output by a filter bank consisting of a set of bandpass filters with regularly increasing center frequencies.

DFT and inverse DFT operations can be expressed mathematically as follows. Consider a set of *N* complex numbers, $x_0, \ldots, x_{N-1}$, that are to be transformed into a sequence of complex numbers, $X_0, \ldots, X_{N-1}$. The DFT is then:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-\frac{i2\pi}{N}kn} \qquad k=0, \ldots, N-1 \tag{3.1}$$

The inverse DFT is:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\frac{i2\pi}{N}kn} \qquad n=0, \ldots, N-1 \tag{3.2}$$

The complex numbers $X_k$ each represent the amplitude and phase of the binned frequency components of the input signal represented by the samples $x_n$. This can be seen by writing $X_k$ in polar form, where the binned sinusoidal amplitudes are $| X_k |$ and the phases correspond to the complex argument. The vector of calculated relative amplitudes of each range of sinusoidal components output by the FFT is referred to as the *magnitude spectrum,* or, alternatively, the *power spectrum,* which is the square of the magnitude spectrum.

In practice, when extracting features from audio, one must deal with *nonstationary signals*, which is to say signals that change over time. A DFT taken over 5 seconds of audio during which 12 different pitches are played, for example, will provide vectors of amplitudes and phases that are the averages of all spectral content over the entire 5 seconds. This means that the frequency contents of all 12 notes are smudged together, making it impossible to determine the spectral characteristics of the notes individually. It therefore seems reasonable to take DFTs of very short segments of the signal, called *windows* or *frames,* so that details of each of the 12 notes, and how they each evolve spectrally, can be captured.

Unfortunately, taking the DFT of shorter segments of sound comes with a mathematical consequence, namely the loss of frequency resolution. In other words, the fewer the number of samples used to calculate the DFT (and, correspondingly, the shorter the amount of time under consideration), the broader the frequency range of each bin output by the DFT. This means that each bin spans a wider range of frequencies, and one therefore loses frequency precision as one reduces the number of samples processed by the DFT. One must therefore balance the competing needs of time localization and spectral resolution, a trade-off that is physically analogous to the Heisenberg Uncertainty Principle.

It is useful to note that a consequence of the tradeoff between time localization and spectral resolution is that audio with a high sampling rate can be better analyzed spectrally than audio sampled at a lower sampling rate, because there are more samples to work with per unit time. Additionally, it is common to use overlapping windows in order to, in a sense, increase the time localization. The distance between the starts of overlapping windows is sometimes referred to as the *hop size.*

In practice, 256-sample, 512-sample, 1024-sample or 2048-sample windows are most often used for audio applications, depending on the time vs. frequency resolution priorities of the application and the sampling rate. In terms of time duration, windows of 10 ms to 30 ms are usually used, with overlap step sizes ranging from 5 ms to 20 ms. Much longer windows can be appropriate if trying to gather information on large-scale structure, for example. Window sample sizes are typically powers of 2, for reasons of computational efficiency.

Graphs of the magnitude spectrum or power spectrum vs. frequency bin are often used to provide visual representations of the frequency content of a signal over the duration of the samples processed by a single DFT window. This visual representation has the disadvantage, as might be expected, that it provides no indication of how the spectrum evolves over time. *Spectrograms* provide an alternative for viewing this evolution by graphing the results of a sequence of DFTs consecutively. Spectrograms thus graph frequency bin vs. time, with colour used to represent magnitude or power. Figures 3.2 and 3.3 respectively provide examples of graphs of the power spectrum and the spectrogram of the same signal.

Optimized versions of the DFT, running in O(N log N) time, as opposed to the O($N^2$) performance of the basic DFT, are known as *Fast Fourier Transforms,* or *FFTs*. There are a number of such algorithms, the most common of which is known as the *Cooley-Tukey algorithm.*

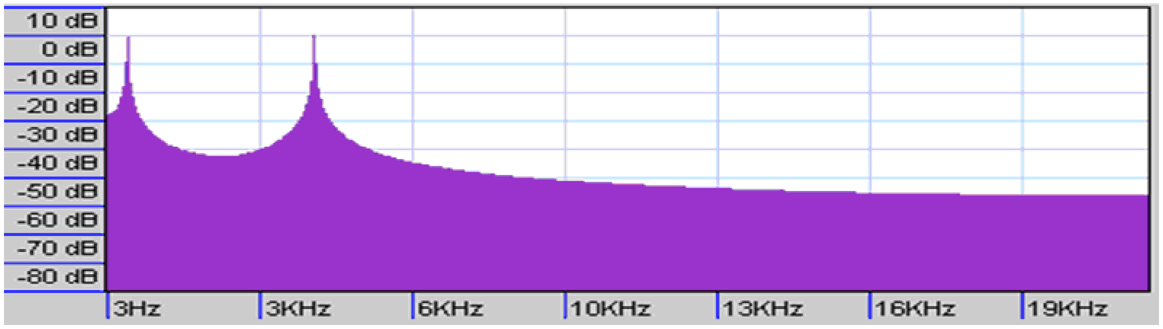


**Figure 3.2:** Power spectrum of a 440 Hz sinusoid lasting 0.5 seconds, followed by a 4400 Hz sinusoid lasting 0.5 seconds. The sampling rate was 44.1 kHz and Hann windows of 16,384 samples were used. Note that there is no way to tell when each tone occurred in time. Note also that, although the peaks corresponding to the two frequencies are clear, there is still low-level spectral leakage at other frequencies.

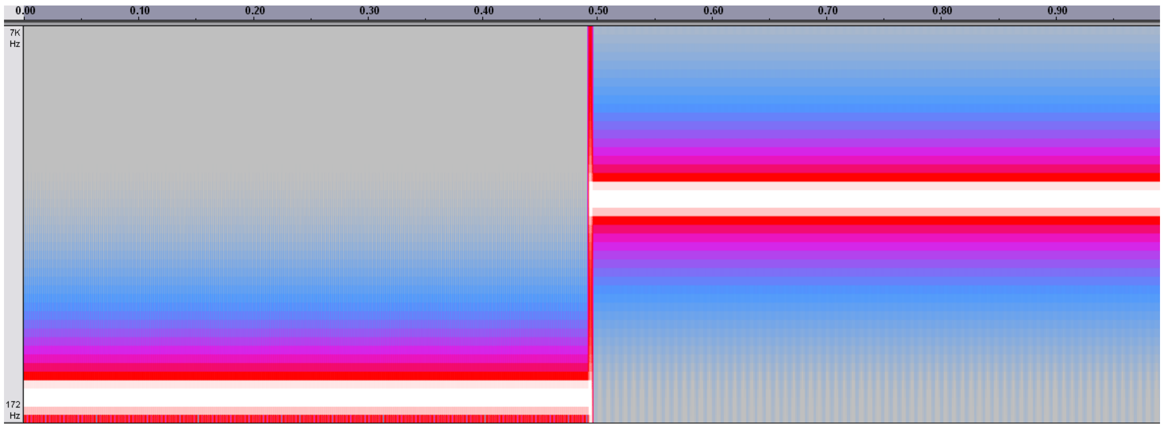


**Figure 3.3:** Spectrogram of the same signal analyzed in Figure 3.2. The vertical axis represents frequency, the horizontal axis represents time and the colour represents power. Note that the two frequencies of the sinusoids are still clear, and that the location in time of the two sine tones is now also clear as well. There is still spectral leakage, however.

As a side note, it is useful to note that an alternative to the DFT exists, namely the *discrete wavelet transform (DWT)*. From the perspective of audio feature extraction, the essential difference between the two relates to frequency and time resolution. The DFT provides uniform time resolution for all frequencies. The DWT, in contrast, provides higher time resolution and lower frequency resolution for high frequencies, and lower time and higher frequency resolution for low frequencies. This is closer to the way that human perception operates than the uniform resolution of the DFT.

### 3.2.3 Windowing functions

As mentioned in Section 3.2.2, breaking audio samples into shorter windows is a common technique for obtaining a degree of time localization when performing Fourier analyses. Although it is possible to use simple rectangular windows (also called *Dirichlet* windows), where each sample in a window is left unmodified, better results can be achieved in some cases by using *windowing functions* that give higher weightings to samples in the center of windows. In particular, alternative windowing functions can be used to reduce the problem known as *spectral leakage,* which refers to power that is assigned to frequency components that are not actually in the signal being analyzed.

Although simple rectangular windows do have good resolution characteristics when applied to signals containing spectral components of comparable strength, they perform poorly when there are spectral components with disparate amplitudes (a problem known as *low dynamic range*). At the other extreme, windowing functions with high dynamic range, such as *Blackman-Harris windows,* tend to result in lower frequency resolution and tend to be poorer with respect to *sensitivity,* which is to say that weak spectral components can be overwhelmed by random noise. In general, high dynamic range windowing functions are preferable when dealing with signals expected to contain spectral components of significantly varying strengths, and low dynamic range windowing functions are preferable when dealing with signals where components are expected to have similar amplitudes. Harris (1978) provides an excellent overview of windowing functions applied to the DFT, and only some of the most pertinent details are described below.

Two of the most commonly used windowing functions are *Hamming windows* and *Hann windows,* which are both moderate with respect to dynamic range. Their popularity is due to their applicability to a wide variety of wave types. They are particularly appropriate for applications associated with moderate bandwidth signals, such as telephone channels. There are also many other windowing functions, each with its own strengths and weaknesses.

The following equations and their associated figures describe each of the windowing functions discussed above. The variable *w* represents the windowing function, *N* represents the width in samples of one window and *n* represents the sample index. Each of these equations is applied to all samples in a frame, and it is the output that is processed by a DFT.

$$w_{rectangular}(n) = 1 \tag{3.3}$$

$$w_{Blackman-Harris}(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

$$a_0 = 0.35875; a_1 = 0.48829; a_2 = 0.14128; a_3 = 0.01168 \tag{3.4}$$

$$w_{Hamming}(n) = 0.53836 - 0.46164 \cos\left(\frac{2\pi n}{N-1}\right) \tag{3.5}$$

$$w_{Hann}(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{3.6}$$

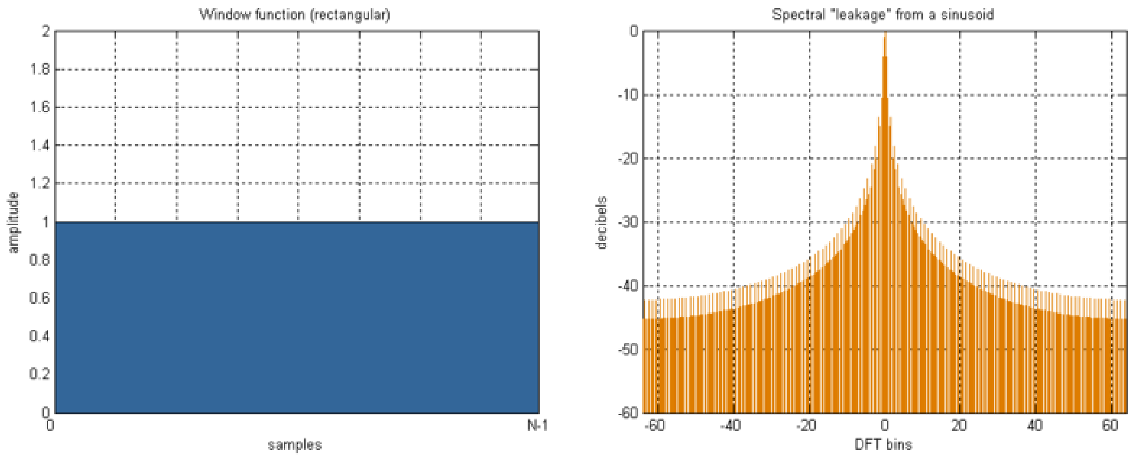


**Figure 3.4:** Rectangular, or Dirichlet, windowing function.[111] This window leaves the original samples in the window unchanged. The image on the left corresponds to the shape of the windowing function, as described in Equation 3.3, and the image on the right demonstrates the type of spectral leakage that results when a window of a simple sinusoidal signal is processed by a DFT after being transformed this windowing function.

[111] This image has been released into the public domain by its creator. It was obtained from http://en.wikipedia.org/wiki/File:Window_function_%28rectangular%29.png.

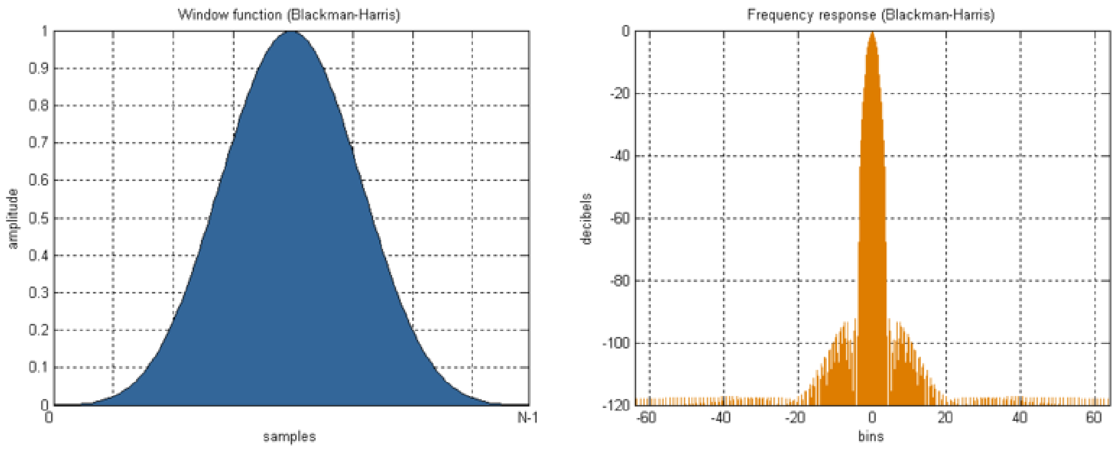


**Figure 3.5:** Blackman-Harris windowing function.[112] The image on the left corresponds to the shape of the windowing function, as described in Equation 3.4, and the image on the right demonstrates the type of spectral leakage that results when a window of a simple sinusoidal signal is processed by a DFT after being transformed this windowing function.

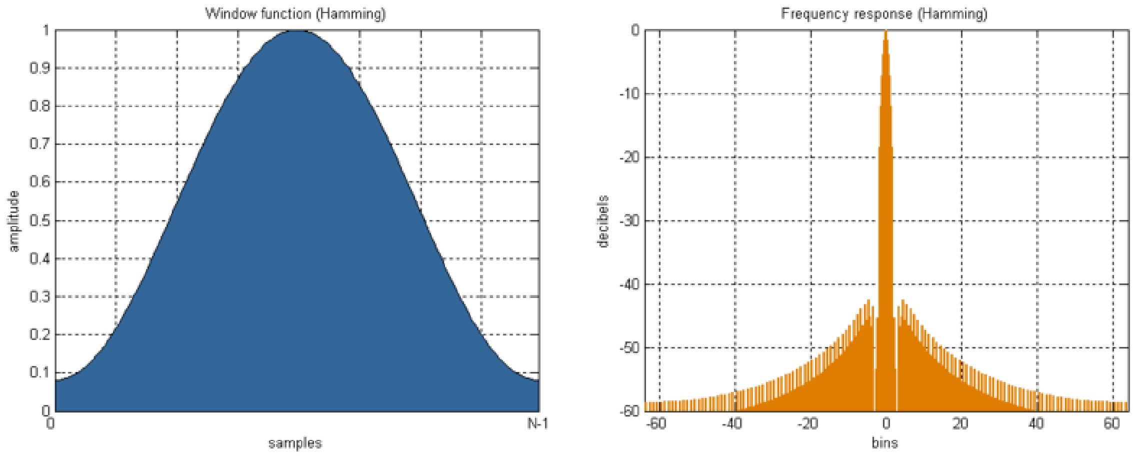


**Figure 3.6:** Hamming windowing function.[113] The image on the left corresponds to the shape of the windowing function, as described in Equation 3.5, and the image on the right demonstrates the type of spectral leakage that results when a window of a simple sinusoidal signal is processed by a DFT after being transformed this windowing function.

[112]This image has been released into the public domain by its creator. It was obtained from http://en.wikipedia.org/wiki/File:Window_function_%28blackman-harris%29.png.
[113] This image has been released into the public domain by its creator. It was obtained from http://en.wikipedia.org/wiki/File:Window_function_%28hamming%29.png.

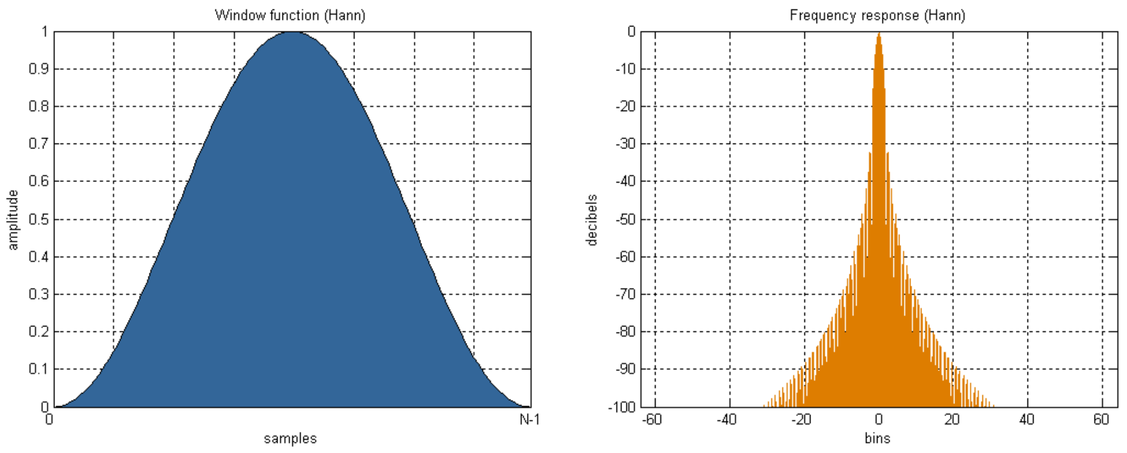


**Figure 3.7:** Hann windowing function.[114] The image on the left corresponds to the shape of the windowing function, as described in Equation 3.6, and the image on the right demonstrates the type of spectral leakage that results when a window of a simple sinusoidal signal is processed by a DFT after being transformed this windowing function.

### 3.2.4 Pre-processing

Audio samples are often pre-processed in a variety of ways before features are extracted from them. This is done to improve efficacy when features are processed by machine learning algorithms, among other reasons.

One common technique is to *normalize* samples based on amplitude. There are a variety of approaches that can be used to normalize signals, but one simple approach is to take the highest absolute sample value in a signal and then multiply all samples in the signal by the highest representable signal value (as determined by the bit depth) divided by the highest sample value present. In effect, this amplifies the signal such that it is as strong as it can be without clipping. If the original signal is clipped, then normalization simply leaves the signal unchanged by effectively multiplying all samples by 1.

Such normalization can be beneficial if multiple signals (or, equivalently, multiple audio files) are to be processed, as it causes all signals to take advantage of the full available dynamic range. This can be helpful when applying machine learning algorithms, since it helps to remove noise for the learners by controlling for variability in recording

[114] This image has been released into the public domain by its creator. It was obtained from http://en.wikipedia.org/wiki/File:Window_function_%28hann%29.png.

levels unrelated to the patterns being studied. However, normalization can in some cases destroy useful information, such as in cases where variability in recording levels is in fact a useful indicator in the application under consideration. So, for example, normalization might be a good idea if features are to be extracted for the purpose of genre classification applied over a diverse corpus of musical recordings. Using normalization might be a poor choice if one is interested in classifying the work of different sound engineers, however, as recording levels might be indicative of their styles.

Another common pre-processing technique is known as *downsampling,* where the sampling rate of a digital signal is reduced. This can be useful in cases where it is important to reduce data size, which could, for example, speed up feature extraction and machine learning.

Downsampling can also be useful for the purpose of making feature vectors for different audio files correspond to the same bandwidth. Consider the case of one file sampled at 44 kHz and another sampled at 11 kHz. If the power spectrum is extracted for the two signals, the 44 kHz signal will contain frequencies up to 22 kHz, but the 11 kHz signal will only contain frequencies up to 5.5 kHz. It would be difficult to train machine learning algorithms on such non-matched power spectra, as the frequency bins would cover different domains. Downsampling the 44 kHz signal to 11 kHz would have the benefit of making the domains of the power spectra of the two signals match, and therefore be easier to process.

Of course, downsampling a signal involves throwing away potentially useful information. Since files are typically all downsampled to the sampling rate of the file with the lowest sampling rate, this means that audio quality is reduced to that of the lowest quality audio file, at least in terms of sampling rate. Although there is the alternative of *upsampling* the files with the lower sampling rates, this does not add any useful information to the files that had lower sampling rates originally, but it does increase their size and corresponding required processing times.

The ideal is to start off with files that are all encoded using the same sampling rate, but this is a luxury that one does not always have. In any case, it is generally believed that the most important information in musical signals tends to be at lower frequencies, so sampling rates as low as 22 kHz or even 11 kHz can preserve much of the information useful for MIR tasks, despite the loss of audio quality for listening purposes when files are downsampled to such sampling rates.

*Channel merging* is also sometimes performed, which is to say that all channels in stereo or multi-channel audio are combined into one track (and usually attenuated and normalized to avoid clipping that could result from simple signal addition). Although panning information can be very useful for some MIR-oriented tasks, such as producer identification, it is generally felt that for most MIR tasks it is not significant, and that the data reduction achieved by channel merging is worth the loss of this data. It should be noted, however, that merging channels can introduce problems related to phase interference, in which case it might be better to simply choose one channel if channel reduction is necessary for the purpose of data reduction.

Signals are also sometimes *rectified* in order to remove the negative components of signals. This can facilitate some types of feature calculations. Two types of rectification can be applied to a signal *x[n],* namely *full wave rectification* and *half wave rectification,* as described by the following two equations:

$$x_{fullwave}[n] = |x[n]| \tag{3.7}$$

$$x_{halfwave}[n] = x[n] \; if \; x[n] \geq 0, \; else \; x_{halfwave}[n] = 0 \tag{3.8}$$

### 3.2.5 File formats and associated problems

As discussed in Section 3.2.1, different audio files can be encoded with different parameters, such as sampling rates and bit depths. Such inconsistencies can cause problems when features are extracted, by influencing feature values in ways that are unrelated to the essential musical content being analyzed. It is possible to at least partially address some of these inconsistencies using pre-processing techniques such as downsampling, but there are, unfortunately, a variety of differences in encoding strategies and parameters that can vary across file formats for which such relatively easy solutions are not available. Before proceeding to discuss these, it is useful to first provide some brief background on audio compression and some of the most common digital audio file formats.

A driving force behind much of the past two decades of research in digital audio files has been to reduce audio size. Raw, uncompressed audio has a size corresponding to the duration of the audio multiplied by the number of channels, the sampling rate and the bit depth, which can result in very large files. This can be inconvenient due to the need for correspondingly large storage space and network bandwidth, although these resources are now much less expensive than they were in the past. A variety of methods have been developed for reducing file size, including both *lossless* techniques to *lossy* techniques.

Lossless techniques take advantage in data redundancies to reduce the space needed to represent audio samples, without losing the ability to exactly reproduce original audio samples. Lossy techniques, in contrast, eliminate some information in ways such that it cannot be recovered, in the hopes that the lost information will not significantly affect perceptible audio quality. Downsampling is a simple example of lossy compression, although most lossy encoding schemes use much subtler and more sophisticated techniques. *Perceptual encoding compression* techniques take advantage of human physiological and psychological auditory limitations in order to remove only that data, it is hoped, whose absence will not have an important perceptual impact on listeners. Lossy techniques have in the past decade become very popular because of the much larger compression ratios than they can achieve relative to lossless compression.

Any software application using file formats associated with either lossless or lossy compression requires software components known as *codecs* to read or write audio in the appropriate formats. Codecs can be either embedded in the software or distributed as plug-ins. Sometimes a single format can have multiple codecs.

There are a variety of different file formats for losslessly encoding audio, some of which also allow the option of applying lossy compression as well. Below are some of the most common lossless formats:

- **FLAC:** A format widely supported by the open-source community.
- **WAVE:** The standard format promoted for use with Windows PCs.
- **AIFF:** The standard format promoted for use with Macintosh computers.
- **AU/SND:** Developed for UNIX computers.

There are also a variety of formats that emphasize lossy compression. Most of these formats allow music to be encoded using different *bitrates* (degrees of compression), and there are also multiple implementations of some of their compression algorithms. Further complicating matters, *variable bitrate* encoding is also sometimes used to dynamically vary the bitrate in a single encoding to optimize both audio quality and space efficiency. Below are some of the most commonly used lossy formats:

- **MP3:** The most popular lossy format, currently. The compression algorithm is not included in the MP3 standard, and there are a wide variety of proprietary algorithms that are used, each of which applies different perceptual encoding techniques, and each of which can therefore have different effects on feature values. There are therefore multiple different encoding codecs, each with their own strengths and weaknesses, although the output of any of these can be successfully parsed by any decoding codec.
- **OGG:** An open-source container that supports a variety of codecs. The most popular of these is Vorbis, which is often said to perform better than MP3 codecs, but is nonetheless much less popular.
- **AAC/MP4:** A format based on the MPEG2 and MPEG4 standards that is considered an improvement over MP3s. Apple uses a proprietary version that can include built-in digital rights management (DRM).
- **WMA:** Microsoft's Windows Media Audio format that emphasizes DRM.
- **RA:** The Real Audio format, originally designed for streaming audio over the Internet.

Sound files that use lossless compression pose no special challenge to feature extraction systems, as the original pre-compression audio signal can be reconstructed exactly. Lossy algorithms can be much more problematic, however, as the extent to which a given compression algorithms affects various features can vary, and depends largely on the particular compression algorithms and the particular features.

Some features are *invariant* to certain compression schemes, which is to say that feature values are not significantly affected by whether or not the compression is applied. Many features are not invariant in this way, however, which is problematic, as in general it is desirable to have features that are dependent on the music itself, not on the particularities of how it was encoded. This can pose serious problems in real-world situations, such as users extracting features from their libraries of MP3s, which could be encoded using a variety of encoders and bitrates.

The problems of determining which combinations of algorithms and features are vulnerable and of developing corrective processing are exacerbated by the fact that there can be many different compression algorithms even for a single file format, and a variety of compression parameters can be used even for a single algorithm. Furthermore, these algorithms are often proprietary, so it can be very difficult to determine exactly how they operate.

Lossy compression algorithms are developed with the intent of making them as invariant as possible to human hearing. If the features being extracted are similar to the features effectively being extracted by humans and if the models developed by machine learning systems based on these features operate similarly to human processing, then perhaps the features themselves could be argued to be relatively invariant to compression, something that would be good news for feature extractors. However, there is no guarantee that this is in fact the case, since models built by machine learning algorithms can well operate in entirely different ways than the human auditory system.

The dependence of feature values on compression schemes is in many ways an elephant in the room of MIR audio feature extraction research. Most published MIR research either assumes, whether correctly or not, that the features extracted are invariant to compression algorithms, or sidesteps the problem entirely by using only losslessly compressed audio or audio compressed using only one algorithm and bitrate. This latter approach is not an accurate simulation of how audio is consumed in the real world.

In one of the few studies to methodically study these issues with respect to music, Sigurdsson, Petersen and Lehn-Schioler (2006) found that, at least for the special case of MFCCs (see Section 3.2.6) extracted from MP3s encoded using different parameters, MFCCs are robust at bitrates equal to or higher than 128 kbits/s, but not at lower bit rates. This is at least somewhat encouraging from a practical perspective, given that most MP3s are encoded at 128 kbits/s or above. However, it must be kept in mind that these results only apply to one specific type of feature, and that all of the MP3s that were experimented with were encoded with the same codec, namely the LAME 3.96.1 encoder.

In any case, these results are discouraging from a general perspective, since they show that there can be significant feature dependence on encoding methodology at least in special cases.

There is a pressing need for further study on the vulnerability of each feature to each compression algorithm, but for the moment this remains an unresolved issue. There is no immediate and easy solution in sight, even if comprehensive data on feature/compression variance were to become available.

### 3.2.6 Common low-level features extracted using the DFT

This section briefly presents some of the low-level spectral features that have often been used in MIR research. Note that the descriptions below have in many cases been expressed in terms of either the power spectrum or the magnitude only as a matter of convention, and many of these features can be calculated in terms of either. $M_t[n]$ refers to the magnitude spectrum at bin $n$, out of $N$ bins, for the Fourier analysis frame corresponding to index $t$. Similarly, $P_t[n]$ refers to the power spectrum.

- **Power Spectrum:** A histogram derived directly from a DFT indicating the relative amounts of energy contained in various regions of the frequency spectrum over a window of time, as discussed in Section 3.2.2. The power spectrum is calculated by, for each bin, summing the square of the imaginary output of a DFT with the square of the corresponding real output of the DFT. The power spectrum and magnitude spectrum are rarely used directly as features, since they are often considered to contain too much raw information to be effectively processed directly by machine learning algorithms. Many spectral features are derived from either the power spectrum or the magnitude spectrum, however.

- **Magnitude Spectrum:** A histogram where the value for each bin is the square root of the value for the corresponding bin of the power spectrum. The magnitude spectrum is often used rather than the power spectrum when one wishes to pay closer attention to the lower energy spectral activity, and the power spectrum is used when it is desirable to emphasize the strong peaks.

- **Strongest Partial:** The center frequency of the bin of the magnitude or power spectrum with the greatest strength. Can provide a primitive form of pitch tracking.

- **Spectral Variability:** The standard deviation of the bin values of the magnitude spectrum. Provides an indication of how flat the spectrum is and if some frequency regions are much more prominent than others.

- **Spectral Centroid:** The centre of mass of the power spectrum. Perceptually, this feature gives an indication of how "dark" or "bright" a sound is, roughly speaking. The spectral centroid, *SC,* is calculated for a window as follows:

$$SC_t = \frac{\sum_{n=1}^{N} P_t[n] * n}{\sum_{n=1}^{N} P_t[n]} \tag{3.9}$$

- **Partial-Based Spectral Centroid:** Based on a variation of the spectral centroid proposed where the spectral centroid is calculated only using bins of the power spectrum that are peaks rather than all bins in the power spectrum, as is done in the traditional spectral centroid. A related feature is used in MPEG-7, where it is referred to as the *harmonic spectral centroid.*

- **Partial-Based Spectral Smoothness:** Based on McAdams' (1999) spectral smoothness algorithm, this algorithm operates on a set of *M* spectral peaks, $T_t[m]$, found by some peak picking algorithm applied to the power spectrum corresponding to the analysis frame with index *t*. The peak-based spectral smoothness, *SS,* is calculated as follows:

$$SS = \sum_{m=2}^{M-1} \left( \log(T_t[m]) - \frac{\log(T_t[m-1]) + \log(T_t[m]) + \log(T_t[m+1])}{3} \right) \tag{3.10}$$

  Note that if M < 3 then this feature does not generate a value. This feature gives an indication of how smooth the power spectrum is.

- **Compactness:** This feature is also based on McAdams' (1999) spectral smoothness algorithm. The difference between the compactness and the partial-based spectral smoothness is that the compactness is calculated based on the magnitude spectrum in general instead of a vector of selected partials. Compactnes, *C,* is calculated as follows:

$$C = \sum_{n=2}^{N-1} \left( \log(M_t[n]) - \frac{\log(M_t[n-1]) + \log(M_t[n]) + \log(M_t[n+1])}{3} \right) \qquad (3.11)$$

- **Spectral Roll-off Point:** The frequency below which some fraction, *k* (typically 0.85 or 0.95), of the cumulative spectral power resides. This provides a measure of the skewness of the power spectrum, and provides an indication of how much of the energy of the signal is in the lower frequencies. This feature is especially useful in speech analysis for differentiating between voiced and unvoiced speech. The spectral rolloff, $SR_t$, is defined as follows:

$$\sum_{n=1}^{SR_t} P_t[n] = k \sum_{n=1}^{N} P_t[n] \qquad (3.12)$$

- **Spectral Flux:** A measure of the amount of spectral change in a signal from frame to frame. The spectral flux, *SF,* is found by calculating the change in the normalized magnitude spectrum, $N_t[n]$, from frame to frame.

$$SF_t = \sum_{n=1}^{N} \left( N_t[n] - N_{t-1}[n] \right)^2 \qquad (3.13)$$

- **Partial-Based Spectral Flux:** Similar to the spectral flux, but calculated using a vector of peaks found in the magnitude spectrum that are tracked across frames rather than simply using the magnitude spectrum as a whole.

- **Method of Moments:** A feature vector consisting of the first five statistical moments of the magnitude spectrum. These consist of the area (0th order), mean (1st order), spectral density (2nd order), skew (3rd order), and kurtosis or "peakedness" (4th order). Although some of these components are individually redundant with some of the other features described in this section, as a unified feature vector the method of moments can provide a compact statistical representation of the magnitude spectrum.

- **Area Method of Moments:** Similar to the simple method of moments feature described above, but this feature analyzes a series of frames of spectral data rather than only one frame. A matrix is constructed consisting of the magnitude spectrum in one dimension and the frame number in the other, which is then analyzed using the two-dimensional method of moments (Fujinaga 1997). This provides a relatively compact statistical representation of a spectrograph encompassing a brief period of time.
- **Mel-Frequency Cepstral Coefficients (MFCC):** A feature vector that is analogous to the magnitude transformed into perceptually motivated bins. This feature vector is calculated by taking the log-amplitude of the magnitude spectrum and then grouping and smoothing the bins based on the perceptually motivated Mel-frequency scale. A discrete cosine transformation is applied. Traditionally, 13 coefficients have been used in much speech-oriented research, but other dimensionalities can certainly be experimented with. Rabiner and Juang (1993) provide more details.

### 3.2.7 Common low-level features extracted in the time domain

Features calculated from the time domain are typically calculated directly from the sequence of samples representing the digital signal. These features are typically extracted over windows of samples of a fixed width, called *analysis windows,* ranging from a few milliseconds to seconds or even minutes. For the purpose of symmetry, it is common to choose analysis windows spanning the same number of samples as the frames used to calculate discrete Fourier transforms. Although this is convenient for combining features from both the time and frequency domains, it is not compulsory.

Some features extracted in the time domain are calculated from collections of consecutive analysis windows in order to capture an indication of how a signal changes over time. In the feature formulae described below, $x[n]$ refers to value of the signal $x$ at sample $n$, for an analysis window consisting of $N$ samples.

- **Zero Crossings:** The number of times that a signal passes the 0 midpoint of the signal range. "Crossing zero" is defined as ($x[n-1] < 0$ and $x[n] > 0$), ($x[n-1] > 0$ and $x[n] < 0$) or ($x[n-1] \neq 0$ and $x[n] = 0$). This feature provides an indication of signal noisiness, as noisy signals with no DC component will have a tendency to cross the midpoint often. There is also a correlation between zero crossings and the spectral centroid of clean (non-noisy) signals.

- **Strongest Frequency via Zero Crossings:** A primitive measure of dominant frequency based on zero crossings, *ZC,* and sampling rate, *SR*, as follows.

  $$SF_{ZC} = \frac{(ZC)(SR)}{2N} \tag{3.14}$$

  This feature can provide a primitive form of pitch tracking for non-noisy monophonic signals.

- **Root Mean Square (RMS):** A measure of the average energy of a signal calculated over an analysis window:

  $$RMS = \sqrt{\frac{1}{N}\sum_{n=1}^{N} x[n]^2} \tag{3.15}$$

- **Relative Difference Function:** A measure of the amount of change in a signal relative to its signal level. This feature, proposed by Klapuri (1999), can be useful in detecting significant changes in a signal, such as note onsets. The relative difference function, *RDF,* is calculated as follows:

  $$RDF = \frac{\frac{d}{dt}\left(A[n]\right)}{A[n]} \tag{3.16}$$

  *A[n]* is an amplitude envelope function of the input signal, *x[n]*. RDF is set to 0 if *x[n]* falls below the audible threshold.

- **Fraction of Low Energy Frames:** The fraction of analysis window within a set of consecutive windows that have an RMS below some threshold. A common variation is to set this threshold dynamically to be the average RMS for the set of frames under consideration. This feature gives an indication of the proportion of silence or near silence in the portion of a signal under consideration.

- **Linear Predictive Coding (LPC):** A methodology developed for analyzing speech signals. LPC can be used to analyze a speech signal by estimating *formants* (spectral bands corresponding to resonant frequencies in the vocal tract), filtering them out, and estimating the intensity and frequency of the residual "buzz" that is assumed to be the original excitation signal. The result is a vector of values that can be used to estimate the formants and the residue. This is often used as an efficient way of representing a speech signal, as the vector can be used to create a source signal based on the residue that can then be filtered based on the formants. With respect to music, LPC can be useful in that the relationships between the formants and the nature of the residue can be helpful in identifying instrument types, for example. There are a variety of ways of implementing LPC. These are in general based on the idea of using a difference equation to represent each sample of a signal as a linear combination of previous samples. The coefficients of the difference equation are typically calculated by minimizing the mean-square error between the actual signal and the predicted signal, which can be done my computing a matrix of coefficient values and finding a solution to the corresponding set of linear equations. Approaches such as autocorrelation, covariance and recursive lattice formulation have been used to efficiently converge to a unique solution. Whatever the implementation, the resulting coefficients can be used as an audio feature vector.

### 3.2.8 High-level information that can be usefully extracted from audio signals

One would ideally like to be able to reliably extract high-level features such as pitches present, chords present, precise rhythmic patterns, etc. Unfortunately, it is not currently known how to reliably automatically extract such high-level information from general signals. Nonetheless, some useful high-level information can still sometimes be extracted, if one operates under the assumption that enough imperfections in the extracted information will be averaged out in broad high-level representations. So, for example, while it may not currently be possible to reliably calculate the precise timing of all note onsets in a complex signal, it still might be possible to construct a reasonably accurate (on average) histogram of time intervals between note onsets that gives some overall information about the rhythmic patterns in the signal as a whole. This section reviews some of the most common ways of representing such high-level information as usable features. Tzanetakis and Cook's work (2002) has been particularly influential in promoting this approach, and Brown's (1993) proposed use of autocorrelation to calculate rhythmic information about music has been widely adopted.

Autocorrelation is a technique that involves comparing a signal with versions of itself delayed by successive intervals, which yields the relative strength of different periodicities within the signal. In terms of musical data, autocorrelation allows one to find the relative strength of different rhythmic pulses. In the case of audio, this can be calculated based on some representation of the strength of a signal consisting of *N* samples, *Y[n],* at various points in time:

$$autocorrelation[lag] = \frac{1}{N}\sum_{n=1}^{N} Y[n]Y[n-lag] \qquad (3.17)$$

where *lag* is the number of samples of delay. Autocorrelation is calculated for all integer values of *lag,* subject to $0 \leq lag < N$.

Note that the function *Y,* may be something as simple as the RMS, or may involve more complex processing, such as a sophisticated onset detection algorithm. *Y* usually involves pre-processing such as full wave rectification, low-pass filtering, downsampling and removal of any DC components to center the signal at zero before autocorrelation is applied.

The calculation of autocorrelation results in a histogram where each bin corresponds to a different lag time. Since the sampling rate is known, such a histogram provides an indication of the relative importance of the time intervals that pass between strong peaks, which is to say probable note onsets. The histogram can thus reasonably be called a *beat histogram.*

Simple statistics can be calculated from beat histograms in order to extract useful high-level rhythmic information, particularly after the application of peak picking algorithms. The periods of the highest peaks, for example, provide good candidates for the tempo of the signal. The ratios between the highest peaks, in terms of both amplitude and period, can give metrical insights and an indication as to whether a signal is likely polyrhythmic or not. The proportional collective strength of low-level bins can give an indication of degree of rubato or rhythmic looseness, as can the width of peaks. The sum of the histogram as a whole can give an indication of beat strength. The number of strong peaks can provide some measure of rhythmic sophistication. And so on.

Histograms can also used to represent useful pitch information. Although there is currently no general polyphonic pitch tracker reliable enough for performing audio to symbolic transcription, there are a wide variety of algorithms that can be used to provide somewhat noisy pitch information that can nonetheless be sufficiently accurate on average for the purpose of providing useful features when the output is collected into a histogram over a period of time.

A variety of different pitch histograms can be constructed, as proposed by Tzanetakis and Cook (2002). Examples include large simple pitch histograms with one bin for each possible (usually discrete) pitch, smaller histograms where pitches separated by integer octave intervals are collected so that there is one bin for each pitch class and folded pitch classes, where there is also one bin for each pitch class, but consecutive bins are separated by perfect fifths rather than semitones.

Once again, there are many useful features that can be calculated from such histograms. The difference between the lowest pitch and the highest pitch in a pitch histogram can indicate range, for example. The bin label of the pitch class histogram with the highest amplitude may indicate the primary key of the piece, or perhaps its dominant. The interval between the two strongest pitches of the folded pitch class histogram can give an indication of the centrality of tonality in the piece. And so on.

As the reliability of pitch detection technology improves, additional histograms of greater complexity could be constructed. Examples include melodic or harmonic histograms.

## *3.3 Previous music information retrieval research*

### 3.3.1 The development and cataloguing of features

Many of the audio features that have been used in MIR research were originally developed for use in areas of digital signal processing not directly related to music. Research in speech processing and recognition in particular had a strong influence on much of the early audio MIR research, with the result that many of the audio features still used today in MIR date back directly to decades-old speech research.

The influence of speech recognition research certainly had a positive initial influence in providing an excellent starting point for MIR audio research, both in terms of a well-developed and well-tested set of features known to be effective at least in the realm of speech processing, and in terms of a set of pattern recognition-oriented methodologies that could be applied to these audio features. The excellent work of Scheirer and Slaney (1997) and of Cary, Parris and Lloyd-Thomas (1999) was particularly important in pioneering the application of speech-oriented approaches to music, and the very influential work of Tzanetakis and Cook (2002) played an essential role in bringing related approaches to the mainstream of the MIR community.

The unfortunate consequence of the easy availability of features originally developed for application to speech is that many researchers have come to rely on them rather than concentrate on developing new features specifically suited to music. Although excellent results can and have been achieved in some cases using primarily speech-oriented features, such features nonetheless fail to fully explore the full range of information available in audio signals that can be particularly pertinent to music. One of the primary goals of jAudio is to provide a platform for easily developing new music-specific features.

Having said this, there is still certainly much to be learned from speech and other audio processing technologies that can be applied to music. Rabiner and Juang (1993) and Gold and Morgan (2000) provide excellent summaries of the early work on speech analysis and processing, in terms of both history and techniques. McClellan, Schafer and Yoder (1999) and Oppenheim, Schafer and Buck (1999) also both provide good backgrounds on digital signal processing in general. The work of Mierswa and Morik (2005) is particularly helpful in discussing general issues and strategies relating to extracting features for use in classifying large collections of audio data in a relatively formalized fashion.

There has also been some important work done with music specifically in mind. Researchers in sound source and instrument identification have made important contributions (e.g., Fujinaga 1998; Kotek 1998; Martin and Yim 1998; Kashino and Murase 1999; Eronen 2001; Herrera, Peeters and Dubnov 2003; Essid, Richard & David 2004). Musical genre classification research has also often emphasized audio features, particularly in the research inspired by the work of Tzanetakis (Tzanetaks, Essl and Cook 2001; Tzanetakis 2002; Tzanetakis and Cook 2002). There has also been important work done on comparing different feature sets and audio parameterizations, such as that by McKinney and Breebaart (2003) and West and Cox (2004). Further relevant general research has been published by Jensen (1999), Park (2000), Verfaille (2003) and by Pope, Holm and Kouznetsov (2004). Works such as these include references to several features that hold potential but have yet to be widely adopted by the MIR community.

Although most of the aforementioned works each present the reader with background on a significant number of features, there has been relatively little published work on collecting features into a unified library to avoid duplication of effort and undocumented discrepancies in the ways that features are implemented. One of the goals of jAudio is to provide a framework for developing and distributing such a collection of features so that they can be shared between researchers as new features are developed.

Lerch, Eisenberg and Tanghe (2005) have begun some promising work on developing the FEAPI API for feature extraction that could be standardized and shared between researchers, serving a role similar to that served by VST in the field of audio effects processing. Unfortunately, no work has since been published on widely adopting or further developing this API. It is hoped that this will change, as the API is well thought out and could facilitate cooperative work and comparative testing between research groups. Some very promising work has also been done by Pope and Kouznetsov (2004) on the FASTLab Music Analysis Kernel Library (FMAK), but the details of this work remain proprietary and unpublished in the academic literature.

### 3.3.2 Alternative MIR audio feature extractor applications

There are a number of existing MIR-oriented standardized audio feature extractors that present alternatives to jAudio. Perhaps the best known of these is Marsyas (Tzanetakis and Cook 2000), a pioneering system that has strongly influenced much of the MIR audio classification research since its release and which is still widely used today. Marysyas is an open-source system that is implemented in C++ and which can save extracted features in Weka's ARFF format, a de facto MIR standard. Although there were some initial hurdles with respect to usability, Marsyas has since been updated so that it can be configured using a scripting language that makes it relatively easy to control which features are selected for extraction, and it also now supports distributed feature extraction (Bray and Tzanetakis 2005). Despite these improvements, there can still be something of a learning curve for new users of Marsyas, particularly with respect to the implementation of new features.

CLAM (Amatrain, Arumi and Ramirez 2002; Amatriain and Pau 2005; Arumi and Amatriain. 2005) is another well-known system that, like Marsyas, is implemented in C++. Although CLAM certainly can be used to extract audio features, much of its emphasis is on signal processing and synthesis functionality. The consequence is that the software's interface and development framework can pose initial obstacles to the non-specialist user who wishes to quickly extract features or design new features.

M2K (Downie, Futrelle and Tcheng 2004) is a patch-based system implemented in Java. This system provides a powerful tool for prototyping new approaches or integrating different systems. Like CLAM, it is designed to be applied to a broad range of applications, and is not specifically designed for feature extraction, although it certainly does provide relatively easy access to feature extraction functionality. It also provides a good environment for developing new features.

M2K makes use of the D2K distributed processing framework, something that is both a strength and a weakness. D2K allows M2K to process tasks using several computers simultaneously, and also provides good classification libraries and a powerful GUI framework. The most significant problems associated to D2K are related to its legal status. D2K's licence not only sometimes makes it complicated for researchers outside the U.S.A. to gain legal access to the software, but forbids its use in commercial applications. This means that any system that uses D2K cannot itself be used for any non-research-based tasks. In addition, D2K itself still has a number of unresolved bugs, something that can be particularly problematic since not all of D2K is open-source.

Maaate (Pfeiffer and Parker 2001) was produced by the Commonwealth Scientific and Industrial Research Organization. While it has a GUI front end, this GUI is geared towards visualization rather than controlling the feature extraction process. Also, maintenance and system development appear to have been inactive for several years.

MIRtoolbox (Lartillot and Toiviainen 2007) is a set of modular Matlab functions for extracting audio features. This system can be very useful for rapidly prototyping new systems, but poses a significant barrier to users without a strong signal processing and coding background or who do not have access to Matlab.

Sonic Visualiser (Cannam et al. 2006) is a system for visualizing audio data in a variety of ways, and also allows one to extract certain features from audio. Sonic Annotator[115] is an associated command-line program that can be used to batch extract features. One of the important advantages of Sonic Visualiser and Sonic Annotator is their incorporation of Vamp[116] plug-in functionality. Vamp is an external API that can be used to add functionality to compatible software for processing audio in various ways. Sonic Visualiser, Sonaic Anotator and the Vamp plug-in format are all implemented using C++, although it is also possible to implement Vamp plug-ins in Python.

Work has also been done on adding embedded feature extraction and machine learning functionality to the ChucK audio programming language (Fiebrink, Wang and Cook 2008a). This has the potential to result in a very powerful environment for implementing feature extraction tools that can be used in real-time, although only a few rudimentary features are available to date.

The Chroma Toolbox[117] extracts features specifically designed for deriving information associated with pitch. Although these features are a little too specialized for general-purpose automatic music classification, they are an excellent contribution to pitch-oriented MIR research, and can provide a useful supplement to other more general feature libraries.

## *3.4 jAudio's functionality*

### 3.4.1 Fundamental functionality

The essential functionality offered by jAudio is the ability to extract features from audio files. jAudio can currently extract features from MP3, WAVE, AIFF, AIFC, AU and SND files. Extracted features can be saved in either Weka ARFF files or as ACE XML files. The latter formats can also include feature metadata (via ACE XML Feature

[115] www.omras2.org/SonicAnnotator
[116] www.vamp-plugins.org
[117] www.mpi-inf.mpg.de/~mmueller/chromatoolbox/

Description files) as well as the actual feature values themselves (via an ACE XML Feature Value files).

Users may choose to save feature values for each individual window, or may choose to save only the mean value and standard deviation over all windows for each feature. The former option is useful if one wishes to retain information on how a feature changes over time or wishes to only look at particular moments in time. The latter option is preferable if one wishes to reduce the feature data that needs to be stored and processed or is only interested in acquiring an overview of an audio file as a whole. Users may also choose to save both types of information if they prefer.

As discussed previously in this chapter, jAudio places an even greater emphasis on providing a platform for the development of new features than do the other jMIR components. The details are described in Sections 3.4.5 to 3.4.8.

Another important aspect of jAudio is that it is integrated with McEnnis' OMEN system (McEnnis 2006; McEnnis, McKay, and Fujinaga 2006b). As described in Section 8.6.3, OMEN allows the sharing of extracted audio feature values among different research groups without violating copyright laws by performing feature extractions on-site and only distributing feature values, not audio samples themselves. A centralized network interface is used to coordinate extraction requests and feature distribution, with. jAudio as the default tool used by OMEN to extract the features on-site.

### 3.4.2 Pre-processing options

jAudio allows users to choose among the following settings prior to feature extraction:

- Whether or not to apply amplitude normalization
- Downsampling or upsampling all audio to a specified sampling rate
- Window size
- Window overlap

Many of the features implemented in jAudio have their own specialized parametrizations, each of which can be individually set in jAudio. jAudio automatically converts multi-channel audio to mono.

### 3.4.3 Implemented features

All together, jAudio is packaged with 26 core implemented features, which are listed below. The details of these features are described in Sections 3.2.6 to 3.2.8. When metafeatures (see Section 3.4.7) are taken into account, jAudio is able to extract a total of 136 features. Even more features can be extracted when jAudio's already implemented aggregators (see Section 3.4.8) are applied to any of the existing features.

There are also two additional "features" that can be extracted by jAudio, but that are in fact only useful as intermediates for calculating other features, not as features themselves. These are FFT Bin Frequency Labels and Beat Histogram Bin Labels. So, while technically jAudio extracts 28 core features, for the purposes of practical usability only 26 are typically used.

The core features implemented and distributed with jAudio are:

- Power Spectrum
- Magnitude Spectrum
- Magnitude Spectrum Peaks
- Spectral Variability
- Spectral Centroid
- Partial-Based Spectral Centroid
- Partial-Based Spectral Smoothness
- Compactness
- Spectral Roll-off Point
- Spectral Flux
- Partial-Based Spectral Flux
- Method of Moments
- Area of Moments
- MFCC

- Zero Crossings
- RMS
- Relative Difference Function
- Fraction of Low-Energy Frames
- LPC
- Beat Histogram

The following features based on the beat histogram feature are also implemented and packaged with jAudio:

- Strongest Beat
- Beat Sum
- Strength of Strongest Beat

Finally, the following rudimentary features for estimating fundamental frequency are also included:

- Strongest Frequency via Zero Crossings
- Strongest Frequency via Spectral Centroid
- Strongest Frequency via FFT Maximum

### 3.4.4 Multidimensional features

It should be noted that many of the features referred to in Section 3.4.3 are multidimensional features, which is to say that each feature "value" in fact consists of a vector of associated values rather than a single value. MFCCs or a beat histogram, for example, are multidimensional features, while RMS and spectral centroid are one-dimensional features.

Some multidimensional features, such as the magnitude spectrum, have dimensionalities that vary and are automatically calculated by jAudio based on factors such as sampling rate and window length. Others, such as MFCC and LPC, have dimensionalities that can be selected by the user. Others, such as the method of moments, have fixed dimensionalities.

jAudio allows the components of multidimensional features to be grouped together into logically related vectors, unlike most alternative audio feature extraction systems, which typically treat each dimension simply as a separate one-dimensional feature when it is saved. The logical grouping functionality offered by jAudio only works when features are saved to ACE XML files, however, as the Weka ARFF format does not permit this type of representation.

A separation between single and multi-dimensional features can be useful because it makes it possible to use classifier ensembles that capitalize on the particular relatedness of the components of multi-dimensional features. An example of this would be an ensemble constructed by training a separate classifier on each multi-dimensional feature, an approach that was experimentally found to be effective for genre classification (McKay 2004).

The maintenance of a clear and distinct logical relationship between the components of multidimensional features is also important when applying metafeatures and aggregators, as it allows dimensional consistency. It is also useful in facilitating the implementation of new features based on existing multidimensional features.

### 3.4.5 Platform for developing new features

The iterative approach to feature development emphasized in the design of all jMIR feature extractors is particularly important with respect to audio features, where low-level features can be combined to build increasingly high-level and musically meaningful features. jAudio therefore takes special steps to facilitate the iterative development and implementation of new features.

Features in jAudio are each implemented as a separate Java class. The class for each feature extends the existing *FeatureExtractor* superclass, which provides a simple, convenient and consistent framework for implementing new features. The new feature can specify any other features that it needs in order to be calculated (e.g., spectral centroid is calculated based on the power spectrum), any time offsets for each of these dependencies (in case the feature depends on past or future windows) and metadata describing the feature and its dimensionality. It is then only necessary to implement a method for calculating the feature value(s), called *extractFeature,* which will be called by jAudio during feature extraction. jAudio automatically passes this method an array of sample values, the sampling rate and any other feature values that were specified as needed by the new feature.

This approach has many advantages with respect to minimizing the learning curve and effort needed to implement new features. An additional advantage is that it does not require any knowledge of the details of how existing features are implemented in order to use them in calculating a new feature. For example, a new feature could output the pitch of detected note onsets based on multiple other features, such as MFCCs and onset detection features, which could in turn be based on features such as power spectrum and RMS. In other words, it is possible to implement increasingly high-level features using lower-level features while maintaining full abstraction on the part of the implementer if desired.

Individuals implementing new features in jAudio do not need to do anything beyond the steps described above. It is not necessary to make any changes anywhere in jAudio's code, as the feature will automatically be detected, called and provided with any data that it needs by jAudio. It is also not necessary to take any action to ensure that a new feature's extraction will be scheduled after that of any other features that it needs, as such scheduling is performed automatically by jAudio, as described in Section 3.4.6.

It is not even necessary for the implementer of a new feature to recompile jAudio after adding a new feature. The new feature class file just needs to be added to a plug-in directory and a reference added to it in jAudio's XML configuration file. This not only simplifies the process of implementation, but is extremely helpful in enabling new features to be distributed to other users of jAudio, who will not need to recompile their jAudio installations. This is especially useful for making new features available to users with limited technical proficiency and for facilitating the use of new features in the context of OMEN. It is even possible to remotely access new features by placing a URL pointing to a feature's class file in the XML configuration file. Features are also automatically detected by jAudio if they are in the Java Classpath.

jAudio also greatly simplifies the process of dealing with audio for those implementing new classes. By automatically interpreting and pre-processing audio files (see Section 3.4.2), jAudio automatically eliminates a great deal of legwork on the part of implementers by simply providing features with an already processed array of samples.

The default Java Core Libraries provide only very basic audio functionality, and jAudio helpfully abstracts away many low-level implementation details for those wishing to implement new features so that they do not need to concern themselves directly with issues such as buffering and format conversions. jAudio is packaged with a number of custom-implemented low-level audio libraries, which users are free to use for performing audio processing tasks, either inside or outside the jAudio framework.

### 3.4.6 Automatic feature extraction dependency resolution and scheduling

A difficulty encountered by any platform designed for iteratively building features that can be calculated based on other feature values is that extraction must be scheduled in the proper order so that feature calculation dependencies are properly resolved. If not treated with care, this difficulty could lead to significant difficulties as users select at runtime which features to extract for a particular task.

jAudio therefore includes functionality for automatically dynamically scheduling feature extraction at runtime such that features that are needed to calculate any other feature will automatically be extracted first. In other words, just before extraction begins, jAudio dynamically orders the execution of feature calculations based on the features selected by the user such that every feature's calculation is executed only after all of the features upon which it depends have been calculated.

Because jAudio calculates feature dependencies and orderings specifically at runtime, it becomes possible to efficiently perform custom extractions that save only a desired subset of all available features. jAudio's scheduling algorithm automatically notes if there are any features that have not been requested for extraction by the user but are in fact needed to calculate one or more other features that have been requested. In such a case, jAudio will automatically extract values for the intermediate features, use them in the appropriate calculations, but not save the intermediate feature values unless specifically requested to do so. Users might thus choose to extract many features calculated from the power spectrum, for example, but not to save the power spectrum itself. They would not need to know that these features depend on the power spectrum or intervene to have it extracted, since jAudio takes care of these details automatically. So, in addition to its benefits with respect to increasing the ease of adding new features to jAudio, jAudio's automatic scheduling also reduces the knowledge needed on the part of users of jAudio.

jAudio's scheduling algorithm also has advantages with respect to efficiency, both by enabling users to extract only those features that they are interested in, and by avoiding the recalculation of intermediate features each time that they are needed for a new feature. This is done by reusing previously calculated values while at the same time saving space by only saving those values that are explicitly needed.

The dependency resolution algorithm has two distinct steps, namely determining which features to extract and determining the order in which they are to be extracted. jAudio begins by making a list, *A,* of all features whose values are to be saved. Another list, *B,* is then built that will eventually contain a list of all features to be extracted. These two lists can differ, as a user may not wish to save a given feature, but this feature may nonetheless be needed in order to calculate another feature that the user does in fact want to save.

Initially *B* is set to be identical to *A*. jAudio then loops through each feature in *B* and adds each feature's dependencies to the end of *B* if they are not already in *B*. The loop terminates when an iteration is completed without adding any new features.

Once the features to be extracted have been identified, jAudio orders these features to ensure that every feature is calculated only after its dependencies have been calculated. To accomplish this, jAudio creates an ordered list of features to extract, *C*. jAudio then cycles through all the features in *B*. If all of the dependencies of a given feature in *B* are in *C*, then this feature is added to the end of *C*. This loop terminates once all features in *B* have been added to *C*.

### 3.4.7 Metafeatures

It can often be useful to calculate certain types of statistical information about a feature. For example, one might wish to calculate the change of a feature's value from window to window, or one might wish to calculate its standard deviation over a series of windows. Such information can often be useful with respect to a wide range of features.

Having to implement such statistical information as a separate individual feature for each relevant feature would be needlessly complex and time consuming. For example, if one is interested in finding the standard deviation over a series of windows for 20 different features, such as spectral flux, spectral centroid, RMS, etc., one would not want to have to implement 20 new functions, namely spectral flux standard deviation, spectral centroid standard deviation, RMS standard deviation, etc.

*Metafeatures* offer a convenient solution to this problem. Metafeatures are feature templates that can be applied to all features to automatically create new derivative features. So, for example, an implementation of a standard deviation metafeature could be used to automatically extract the standard deviation of any other feature, including new features that have yet to be implemented, without needing to explicitly implement it for each feature. Metafeatures are output by jAudio exactly like any other features.

jAudio automatically gives users the option at runtime of extracting each metafeature for every feature that is present in the jAudio feature catalogue. The option of applying existing metafeatures to new features is automatically added to the jAudio interface once a new feature is added to jAudio, and the option of applying new metafeatures to all existing features is automatically given to the user once the new metafeature has been added to jAudio.

jAudio is packaged with implementations of three basic metafeatures:

- **Derivative:** The change in the value of a feature from window to window.
- **Running Mean:** The mean of a feature across a set of adjacent windows. This can be useful in smoothing noisy features, assuming that sufficient time resolution is available.
- **Standard Deviation:** The standard deviation of a feature across a set of adjacent windows. Standard deviations have been found to be particularly useful in a number of informal experiments, as they provide a relatively compact representation of how audio signals change with time.

jAudio also makes it possible to chain metafeatures together. jAudio includes implementations of two such chained metafeatures:

- **Derivative of Running Mean:** The change in the running mean metafeature from one set of windows to another. This can be useful in tracking overall trends in noisy data.

- **Derivative of Standard Deviation:** The change in the standard deviation metafeature from one set of windows to another. This can be useful in tracking changes in the variability of a feature.

jAudio therefore includes a total of five metafeatures. The software is designed to make it simple to implement and add further metafeatures in the future, as it is only necessary to extend the MetaFeatureFactory class to implement a new metafeature. Adding new metafeatures to jAudio is much like adding new features, as described in Section 3.4.5.

To further illustrate how metafeatures work, consider the example of a researcher who has implemented an imagined feature named tonal energy and has added it to jAudio. Users would then automatically have the option at runtime of whether or not to extract each metafeature (i.e., derivative, running mean, standard deviation, etc.) for this new feature, without the implementer of tonal energy needing to implement any code for calculating these quantities. Figure 3.8 provides another illustration of how metafeatures work.

It should be noted that the metafeature aspect of jAudio was designed, implemented and tested entirely by Daniel McEnnis.

### 3.4.8 Feature aggregators

A *feature aggregator* is a jAudio function for collapsing some sequence of feature vectors into a smaller sequence of vectors or into a single vector. This idea is very similar to the *aggregate features* proposed by Bergstra et al. (2006). Aggregators in jAudio take as input the output of a selected subset of the available features over all windows in an audio file and produce a single output for the entire file. In essence, aggregators collapse information on how a feature or set of features vary with time into a single representation. Aggregators can be very helpful in attempting to come to terms with the *curse of dimensionality* when dealing with potentially huge amounts of feature data.

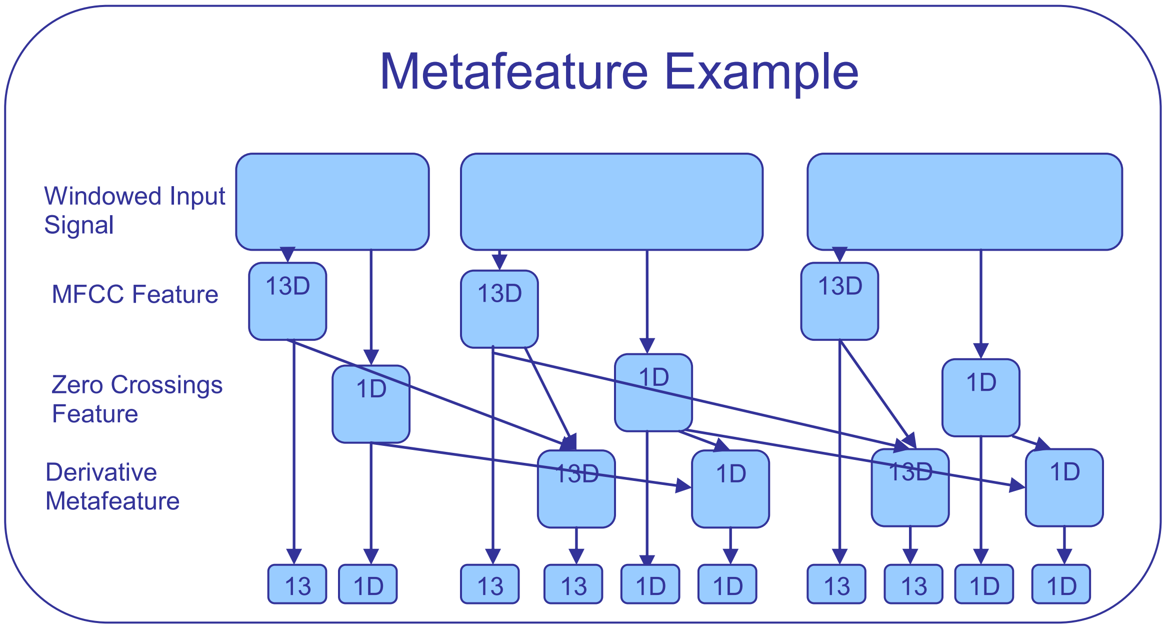


**Figure 3.8:** An illustration of how metafeatures work. This example shows three windows of an audio signal. The MFCC and zero crossings features are extracted from these windows, with dimensionalities of 13 and 1, respectively. The derivative metafeature is then calculated for each window and for each feature, with dimensionalities of 13 and 1, corresponding to the dimensionalities of the MFCC and zero crossings features respectively. This graphic is taken from the poster presented by McEnnis at ISMIR 2006 (McEnnis, McKay and Fujinaga 2006a).

Aggregators in jAudio come in two varieties, namely *general aggregators* and *specific aggregators.* General aggregators are functions that, if they are selected by the user, will automatically be applied individually to every feature extracted. For each feature, per-window feature output is collapsed into a single vector. If a feature that a general aggregator is applied to is multidimensional, then the aggregator function will be applied to each dimension separately. jAudio includes implementations for the following general aggregators:

- **Mean:** Calculates the mean value of a feature over all windows in a file.
- **Standard Deviation:** Calculates the standard deviation of a feature over all windows in a file.

- **MFCC:** This aggregator treats the value of the feature that it is applied to for each window as if it were a sample in a digital signal. The MFCC is then calculated for this "signal."

Specific aggregators are functions that target only one or more specific features instead of all features. The features to which they are to be applied must therefore be specified when specific aggregators are chosen to be extracted. Specific aggregators can be very useful for representing in a low-dimensional way how different features change together. jAudio implements the following specific aggregators:

- **Area of Moments:** This aggregator combines the values for all selected features into a single array for each window, which results in a two-dimensional matrix, with one row for each window and one column for each feature. This matrix is then treated as a two-dimensional image, and the two-dimensional statistical moments (Fujinaga 1997) are calculated and output for this image.
- **Multiple Feature Histogram:** An aggregator that constructs a histogram indicating in how many windows various feature values fall within various bins of value ranges. This can be useful in tracking concurrent changes between features over an input signal.

As is the case with features and metafeatures, users can implement their own aggregators relatively easily. This is done by extending the Aggregator class.

It should be noted that the aggregator aspect of jAudio was designed, implemented and tested entirely by Daniel McEnnis.

### 3.4.9 Additional functionality

jAudio is intended to be used not only as a feature extractor, but also as a platform for developing new features. As such, it includes additional functionality to assist the testing of new functions, as follows:

- **Sound Synthesis:** Basic additive sound synthesis functionality is included that allows sine waves, complex tones, white noise, FM sweeps, decay pulses, etc. at various specified frequencies and amplitudes to be combined and saved.

- **Recording:** jAudio can record audio from microphones. The system sound can also be recorded, if permitted by the sound card and operating system.
- **Playback:** Audio and MIDI files can be played.
- **MIDI to Audio Conversion:** The recording and playback functionality can be combined to perform rudimentary real-time MIDI to audio conversion.
- **Format Conversion:** Sampling rate, bit depth, signing, byte order and file encoding format can all be varied.
- **Audio Processing:** Gain, normalization and channel combination can be performed on a file-by-file basis. This is in addition to jAudio's pre-processing functionality, which can automatically apply such operations to groups of files.
- **Display Functionality:** Audio waveforms and power spectra can both be graphed for any file or portion of a file. Individual sample values can also be listed by time.
- **Batch and Configuration Files:** jAudio includes a *configuration file* format that allows feature extraction and pre-processing settings to be saved between jAudio sessions. *Batch files* can also be saved that include this information as well as sets of files to extract features from. This facilitates the repetition of experiments after changes have been made to the software as well as the performance of multiple experiments with similar parameters.
- **Installation Utility:** Automatically installs jAudio, taking into consideration each user's particular operating system and Java installation.

## 3.5 jAudio's interface

Like all of the jMIR components, jAudio has been designed to be easily usable by researchers with a wide range of technical backgrounds, from users who want to use it simply as a dedicated feature extraction application, to developers who wish to add functionality or modify the code to meet their own research needs. jAudio therefore includes several different interfaces, namely a GUI, a command-line interface and an API for embedding jAudio's functionality in other applications. The ability to embed jAudio in other applications was designed with the particular needs of the OMEN project in mind, although jAudio can certainly be embedded in other applications as well.

The command-line interface is designed to enable users to quickly extract features from batches of audio files. Although all of the functionality provided by the command-line interface is also accessible via the GUI, some users prefer the speed of access offered by a command-line interface. In any case, the best way to use the command-line interface is typically to save batch files via the GUI interface, and then simply load these files via the command-line. Of course, some settings can be set directly via command-line arguments, and batch files can be modified manually using a text editor if preferred.

The GUI makes all of the user-oriented functionality described in Section 3.4 available to users. This functionality will not be explicitly repeated in detail here, but the details of how some of the highlights can be accessed via the GUI will be discussed. More details are available in jAudio's documentation, particularly in its embedded manual (Figure 3.9).

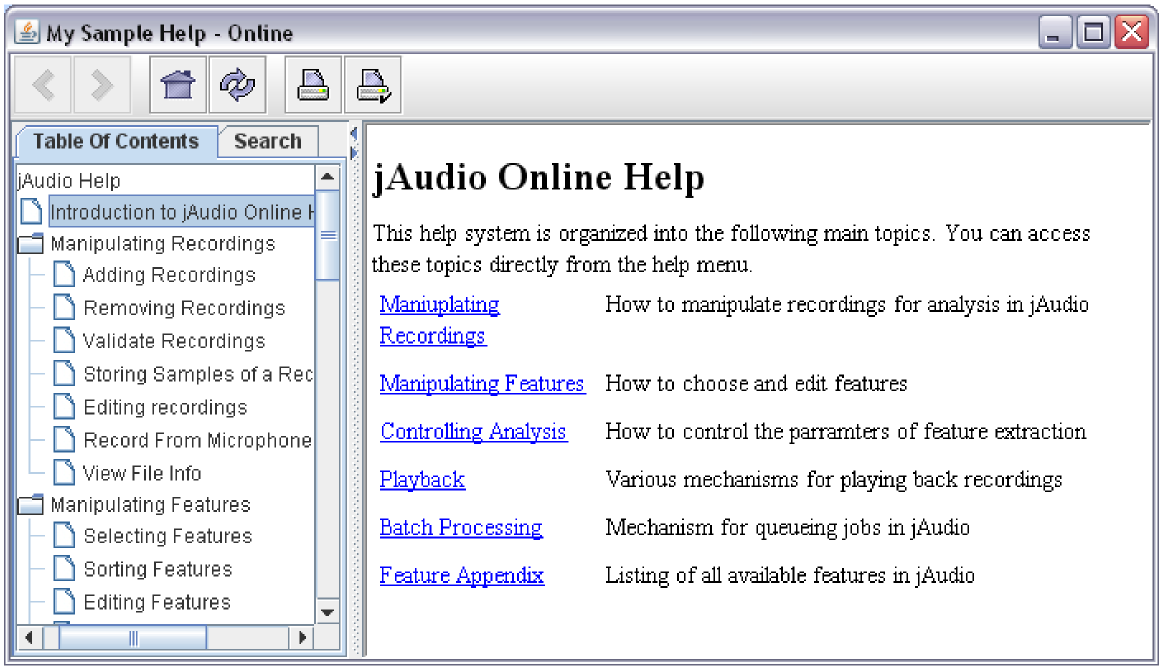
jAudio's fundamental functionality is accessible via the *Feature Extraction Window* (Figure 3.10). The left side of this window lists all audio files that the user has selected to have features extracted from. The right side of this window lists all features and metafeatures that are detected by jAudio at runtime. The basic features are listed in bold, and available metafeatures are automatically generated under each such feature. The user may select which features and metafeatures to extract using the checkboxes presented here, and parameters for individual features and metafeatures, as well as identifying descriptions, can be accessed by double clicking on feature names. Aggregators may be accessed and configured via separate windows (Figure 3.11).

The Feature Extraction Window also provides direct access to some of jAudio's most commonly used functionality, including the ability to add or delete recordings, set feature save paths and set window settings. jAudio's other functionality can be accessed via its various menus.

All of the user settings relating to features, metafeatures, aggregators and selected recordings can be saved to configuration or batch files, as can certain other settings relating to pre-processing audio. Loading these will restore the appropriate settings on the GUI.

jAudio also includes functionality for viewing information on audio files and applying basic processing operations to them, as shown in Figure 3.12. This includes the ability to display time indexed sample values, waveforms and spectral analyses for all or parts of audio files (Figure 3.13). A separate window (Figure 3.14) also allows users to change the encoding settings for individual files.

In order to facilitate the testing of new features, jAudio also allows users to record audio from a microphone or from the system sound (Figure 3.15) or to perform rudimentary additive sound synthesis (Figure 3.16).

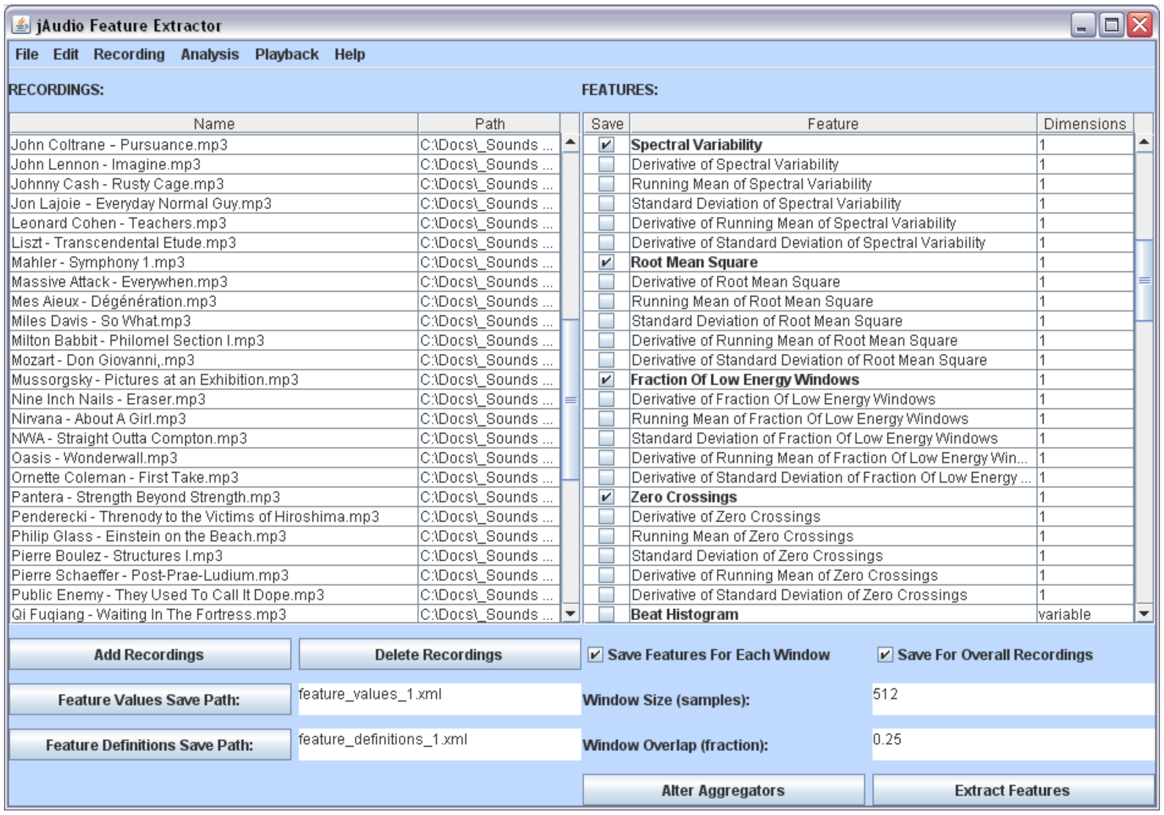


jAudio Online Help

This help system is organized into the following main topics. You can access these topics directly from the help menu.

| | |
|---|---|
| Maniuplating Recordings | How to manipulate recordings for analysis in jAudio |
| Manipulating Features | How to choose and edit features |
| Controlling Analysis | How to control the parramters of feature extraction |
| Playback | Various mechanisms for playing back recordings |
| Batch Processing | Mechanism for queueing jobs in jAudio |
| Feature Appendix | Listing of all available features in jAudio |

**Figure 3.9:** jAudio’s manual, which can be accessed via jAudio’s *Help* menu.



RECORDINGS:

| Name | Path |
|---|---|
| John Coltrane - Pursuance.mp3 | C:\Docs_Sounds ... |
| John Lennon - Imagine.mp3 | C:\Docs_Sounds ... |
| Johnny Cash - Rusty Cage.mp3 | C:\Docs_Sounds ... |
| Jon Lajoie - Everyday Normal Guy.mp3 | C:\Docs_Sounds ... |
| Leonard Cohen - Teachers.mp3 | C:\Docs_Sounds ... |
| Liszt - Transcendental Etude.mp3 | C:\Docs_Sounds ... |
| Mahler - Symphony 1.mp3 | C:\Docs_Sounds ... |
| Massive Attack - Everywhen.mp3 | C:\Docs_Sounds ... |
| Mes Aieux - Dégénération.mp3 | C:\Docs_Sounds ... |
| Miles Davis - So What.mp3 | C:\Docs_Sounds ... |
| Milton Babbit - Philomel Section I.mp3 | C:\Docs_Sounds ... |
| Mozart - Don Giovanni,.mp3 | C:\Docs_Sounds ... |
| Mussorgsky - Pictures at an Exhibition.mp3 | C:\Docs_Sounds ... |
| Nine Inch Nails - Eraser.mp3 | C:\Docs_Sounds ... |
| Nirvana - About A Girl.mp3 | C:\Docs_Sounds ... |
| NWA - Straight Outta Compton.mp3 | C:\Docs_Sounds ... |
| Oasis - Wonderwall.mp3 | C:\Docs_Sounds ... |
| Ornette Coleman - First Take.mp3 | C:\Docs_Sounds ... |
| Pantera - Strength Beyond Strength.mp3 | C:\Docs_Sounds ... |
| Penderecki - Threnody to the Victims of Hiroshima.mp3 | C:\Docs_Sounds ... |
| Philip Glass - Einstein on the Beach.mp3 | C:\Docs_Sounds ... |
| Pierre Boulez - Structures I.mp3 | C:\Docs_Sounds ... |
| Pierre Schaeffer - Post-Prae-Ludium.mp3 | C:\Docs_Sounds ... |
| Public Enemy - They Used To Call It Dope.mp3 | C:\Docs_Sounds ... |
| Qi Fuqiang - Waiting In The Fortress.mp3 | C:\Docs_Sounds ... |

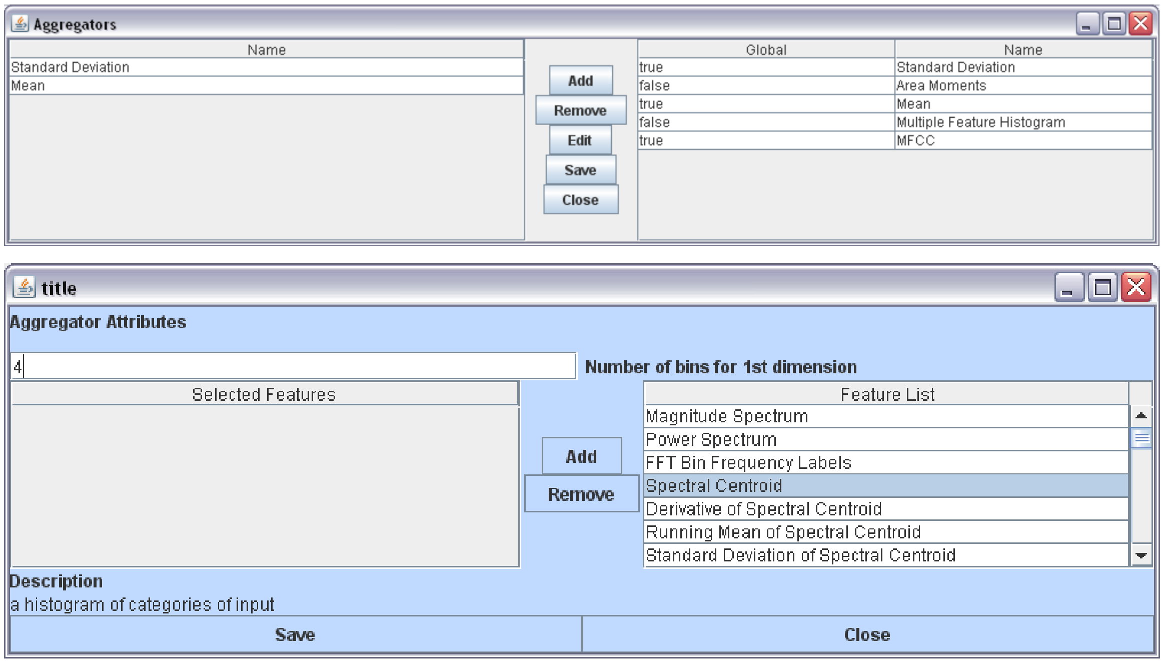
FEATURES:

| Save | Feature | Dimensions |
|---|---|---|
| ✓ | **Spectral Variability** | 1 |
|  | Derivative of Spectral Variability | 1 |
|  | Running Mean of Spectral Variability | 1 |
|  | Standard Deviation of Spectral Variability | 1 |
|  | Derivative of Running Mean of Spectral Variability | 1 |
|  | Derivative of Standard Deviation of Spectral Variability | 1 |
| ✓ | **Root Mean Square** | 1 |
|  | Derivative of Root Mean Square | 1 |
|  | Running Mean of Root Mean Square | 1 |
|  | Standard Deviation of Root Mean Square | 1 |
|  | Derivative of Running Mean of Root Mean Square | 1 |
|  | Derivative of Standard Deviation of Root Mean Square | 1 |
| ✓ | **Fraction Of Low Energy Windows** | 1 |
|  | Derivative of Fraction Of Low Energy Windows | 1 |
|  | Running Mean of Fraction Of Low Energy Windows | 1 |
|  | Standard Deviation of Fraction Of Low Energy Windows | 1 |
|  | Derivative of Running Mean of Fraction Of Low Energy Win... | 1 |
|  | Derivative of Standard Deviation of Fraction Of Low Energy ... | 1 |
| ✓ | **Zero Crossings** | 1 |
|  | Derivative of Zero Crossings | 1 |
|  | Running Mean of Zero Crossings | 1 |
|  | Standard Deviation of Zero Crossings | 1 |
|  | Derivative of Running Mean of Zero Crossings | 1 |
|  | Derivative of Standard Deviation of Zero Crossings | 1 |
|  | **Beat Histogram** | variable |

[✓] Save Features For Each Window
[✓] Save For Overall Recordings

Feature Values Save Path: feature_values_1.xml
Feature Definitions Save Path: feature_definitions_1.xml
Window Size (samples): 512
Window Overlap (fraction): 0.25

**Figure 3.10:** jAudio's main window. This window allows the user to choose both the recordings to extract features from and the particular features and metafeatures to extract, as well as to specify other settings.



| Name |
|---|
| Standard Deviation |
| Mean |

| Global | Name |
|---|---|
| true | Standard Deviation |
| false | Area Moments |
| true | Mean |
| false | Multiple Feature Histogram |
| true | MFCC |

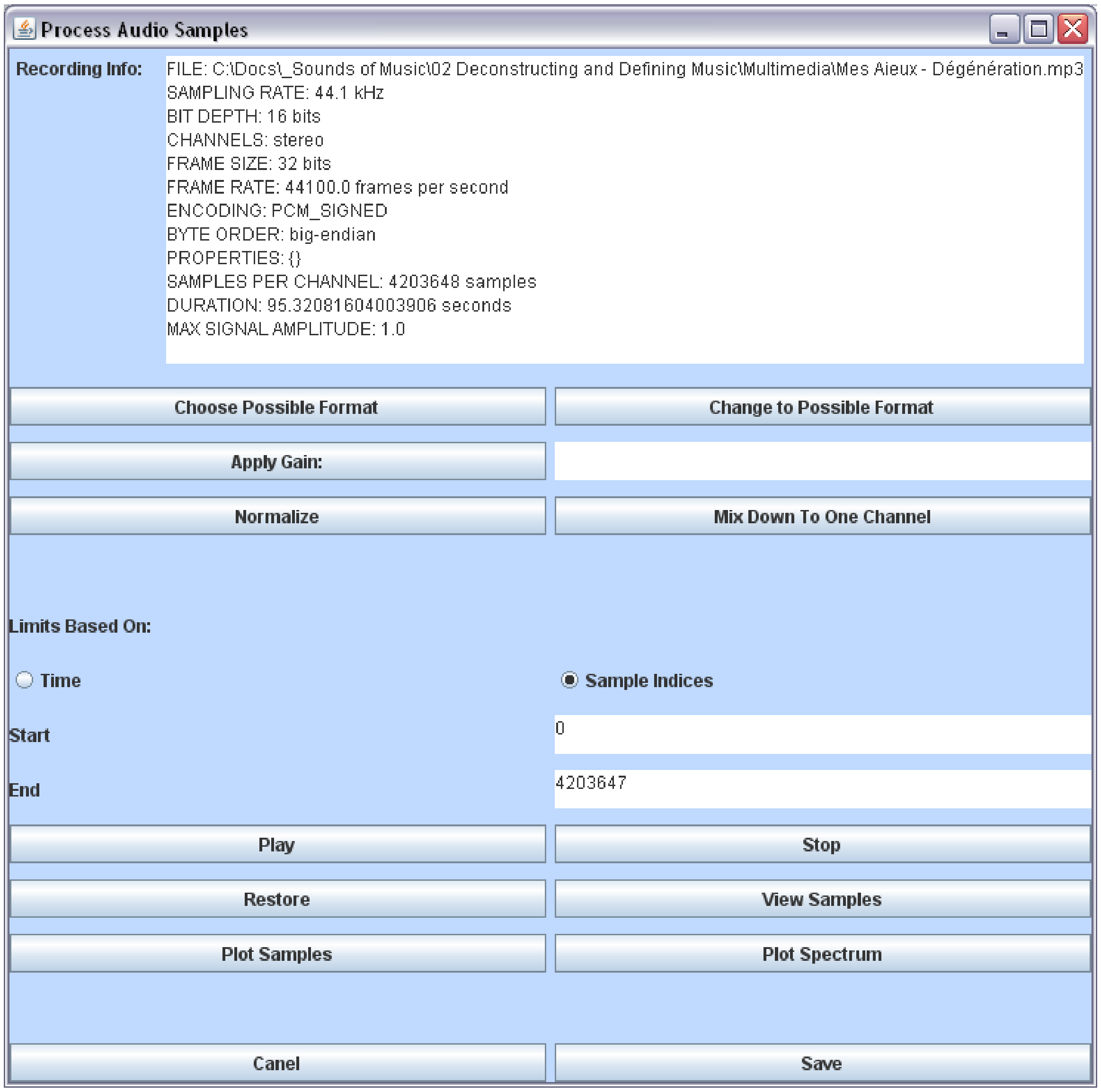
4 Number of bins for 1st dimension

| Selected Features |
|---|

| Feature List |
|---|
| Magnitude Spectrum |
| Power Spectrum |
| FFT Bin Frequency Labels |
| Spectral Centroid |
| Derivative of Spectral Centroid |
| Running Mean of Spectral Centroid |
| Standard Deviation of Spectral Centroid |

Description
a histogram of categories of input

**Figure 3.11:** jAudio windows allowing aggregators to be selected (top) and configured (bottom).



Recording Info:

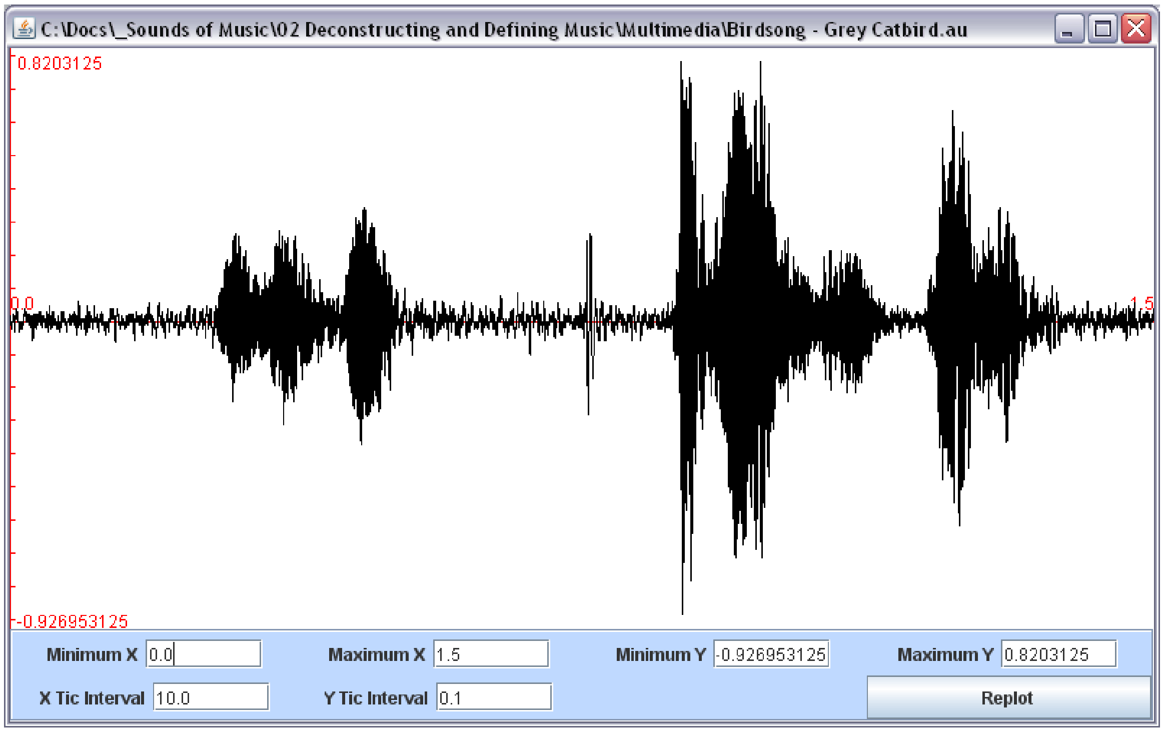
```
FILE: C:\Docs\_Sounds of Music\02 Deconstructing and Defining Music\Multimedia\Mes Aieux - Dégénération.mp3
SAMPLING RATE: 44.1 kHz
BIT DEPTH: 16 bits
CHANNELS: stereo
FRAME SIZE: 32 bits
FRAME RATE: 44100.0 frames per second
ENCODING: PCM_SIGNED
BYTE ORDER: big-endian
PROPERTIES: {}
SAMPLES PER CHANNEL: 4203648 samples
DURATION: 95.32081604003906 seconds
MAX SIGNAL AMPLITUDE: 1.0
```

Limits Based On: ○ Time [✓ ● Sample Indices]

Start: 0

End: 4203647

**Figure 3.12:** jAudio window allowing the user to see various types of information about individual recordings as well as apply some basic processing operations to them.



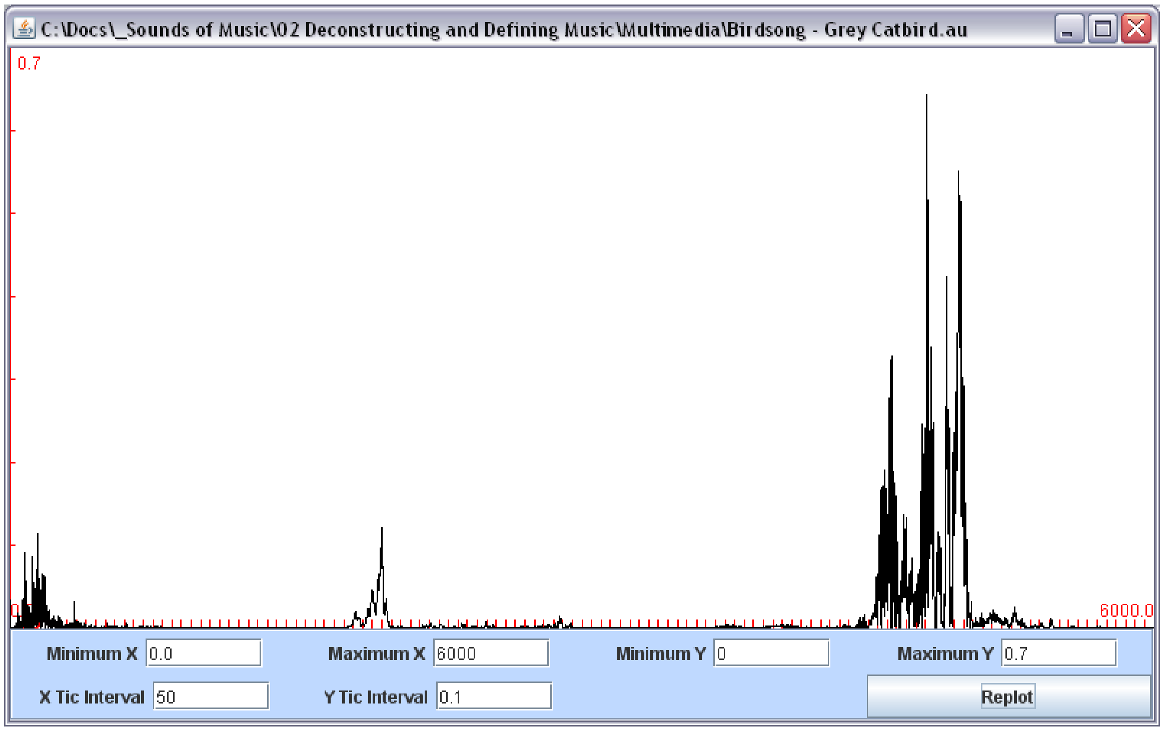
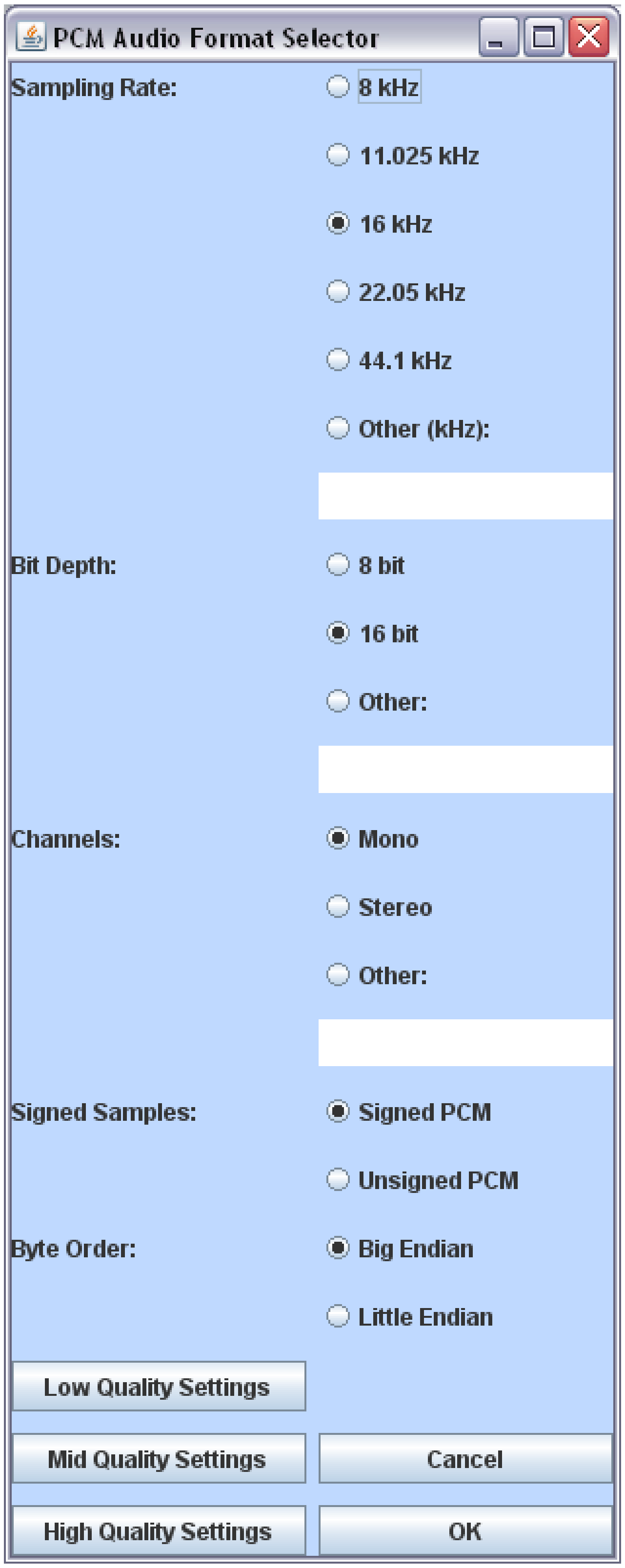


**Figure 3.13:** jAudio windows for displaying waveforms (top) and power spectra (bottom) for all or parts of audio files.



Sampling Rate: 8 kHz, 11.025 kHz, [✓ 16 kHz], 22.05 kHz, 44.1 kHz, Other (kHz):

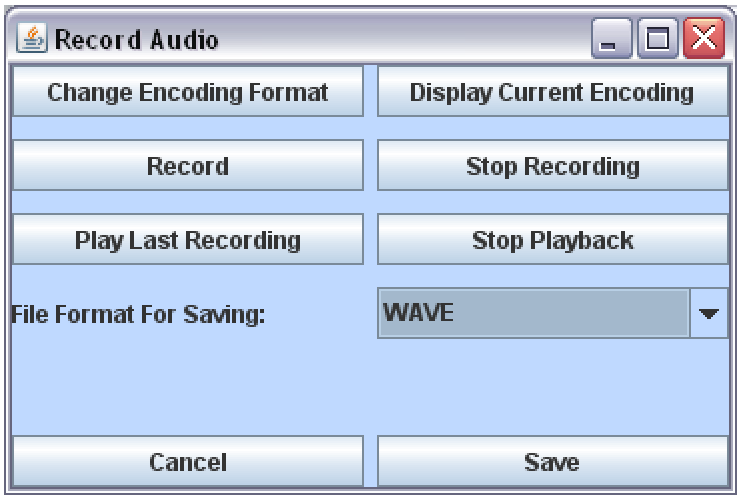
Bit Depth: 8 bit, [✓ 16 bit], Other:

Channels: [✓ Mono], Stereo, Other:

Signed Samples: [✓ Signed PCM], Unsigned PCM

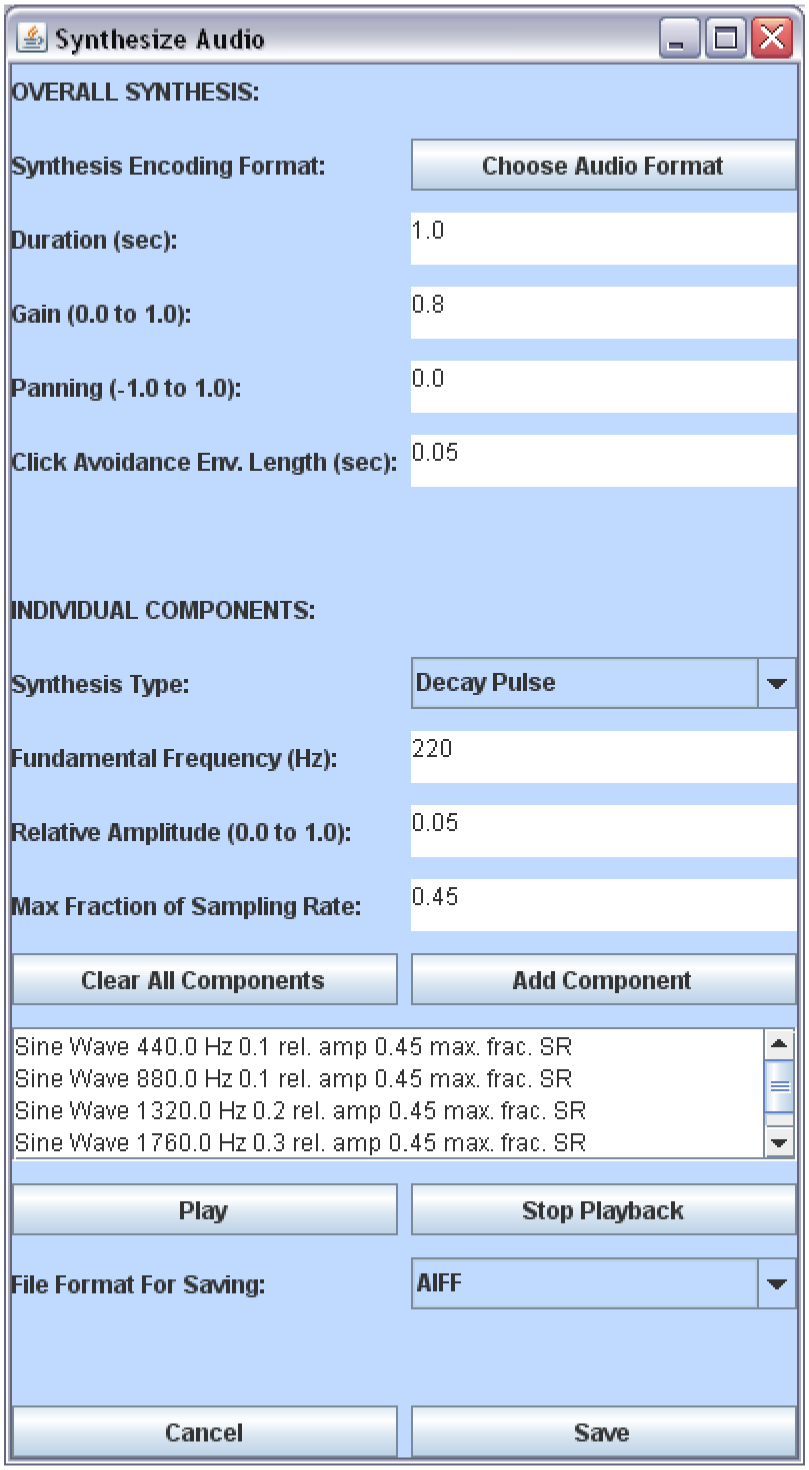
Byte Order: [✓ Big Endian], Little Endian

**Figure 3.14:** The jAudio window for selecting or changing encoding settings for individual audio files.



File Format For Saving: WAVE

**Figure 3.15:** jAudio window allowing the user to record sound from a microphone or from the system sound in order to facilitate the testing of new features.


Synthesize Audio

OVERALL SYNTHESIS:

Synthesis Encoding Format: Choose Audio Format

Duration (sec): 1.0

Gain (0.0 to 1.0): 0.8

Panning (-1.0 to 1.0): 0.0

Click Avoidance Env. Length (sec): 0.05

INDIVIDUAL COMPONENTS:

Synthesis Type: Decay Pulse

Fundamental Frequency (Hz): 220

Relative Amplitude (0.0 to 1.0): 0.05

Max Fraction of Sampling Rate: 0.45

Clear All Components | Add Component

```
Sine Wave 440.0 Hz 0.1 rel. amp 0.45 max. frac. SR
Sine Wave 880.0 Hz 0.1 rel. amp 0.45 max. frac. SR
Sine Wave 1320.0 Hz 0.2 rel. amp 0.45 max. frac. SR
Sine Wave 1760.0 Hz 0.3 rel. amp 0.45 max. frac. SR
```

Play | Stop Playback

File Format For Saving: AIFF

Cancel | Save

**Figure 3.16:** jAudio window allowing the user to perform rudimentary additive sound synthesis in order to facilitate the testing of new features.

## 3.6 Summary of original contributions

jAudio is designed to be a dedicated audio feature extraction application, a code library that can be embedded in other applications and a feature development platform. There are, as discussed in Section 3.3.2, already a number of existing alternative high-quality audio feature extraction applications designed for use in MIR research. jAudio does, however, have a number of characteristics that distinguish it from these alternatives. Many, but not all, of these advantages are associated with jAudio's emphasis on providing a platform for developing, testing and sharing new features. The highlights of jAudio's distinguishing characteristics are as follows:

- Functionality facilitating the use of multidimensional features in a logically natural way, including features with variable dimensionalities, both in terms of using them to derive other features and in terms of maintaining a logical relationship between the values of multidimensional features that facilitates their use with specialized classifier ensembles.
- The ability to save extracted feature values in ACE XML files. This has many advantages over alternatives such as Weka ARFF (a format that jAudio can also save to if required). These benefits include the ability to store the specific relationships between analysis windows and audio files as a whole, the ability to preserve the relationship between the dimensions of multidimensional features, the ability to save features for files overall in combination with features only extracted from portions of these files and the ability to automatically store metadata about recordings as well as features themselves. ACE XML files are discussed in more detail in Chapter 7.
- The ability to implement features as plug-ins using a simple API and distribute compiled new features to other users without requiring these new users to recompile their jAudio installations.
- Automatic feature dependency resolution and scheduling.
- Metafeatures.
- Feature aggregators.

- Functionality facilitating feature testing, including the ability to record audio, sonify MIDI files, synthesize audio, convert between audio formats and graph audio waveforms and spectral analyses. The ability to save configuration settings and batch files also facilitates retesting after implementing bug fixes.
- Easy-to-use GUI making the software usable by non-technically oriented researchers. An installation utility is also included.

## *3.7 Future research*

The primary emphasis in future research will be to develop, test and share new features. The features that are distributed with jAudio provide a good basis for developing higher-level features, which can then in turn be used to implement increasingly high-level features. Research in music theory, of the type discussed in Chapter 4 with respect to jSymbolic, could provide useful inspiration for developing such high-level features. Psychological research in audio perception and cognition could also be helpful, such as the general work described by Moore (2003) or the grouping principles outlined by Bregman (1990). Such research, as discussed in Chapter 2, could help to provide an understanding of what basic types of information humans extract from audio signals.

There are also a number of existing and as-of-yet unthought-of low-level features that remain to be implemented. Examples include spectral flatness measure and spectral crest factor (Jayant and Noll 1984), as well as chroma vectors (Goto 2003). Features derived from the discrete wavelet transform rather than the FFT could also be useful. Additional metafeatures and aggregators could also prove helpful, especially ones designed for extracting various statistical characteristics from histogram-based features.

As discussed in Section 3.2.5, much research remains to be done on the extent to which various features are dependent on audio file format and associated representational parameters. jAudio would provide an excellent platform for carrying out such research.

With respect to improvements to the jAudio software itself, a priority is to increase jAudio's processing speed. Although fast enough as is for many practical MIR purposes, it would ideally be nice to have jAudio be fast enough to extract features in real-time. Features could then be broadcast to real-time applications for purposes such as interactive performance systems (e.g., McKay 2005). Faster feature extraction would also make jAudio more generally applicable to rapid processing of very large audio collections.

There are also a number of low-level optimizations that could improve jAudio's performance. In particular, more efficient techniques and representations could be used to generate the array of samples that serve as the basic input to individual features. Minor improvements could also be made that would reduce the small amount of rounding error currently found in jAudio's sample calculations. jAudio also currently requires a large amount of memory during processing, particularly when applied to long audio files, something that could be ameliorated in the future.

There are also several useful types of functionality that could be added to jAudio. For example, it would be helpful to have more flexible control over windowing. Being able to extract features from only a subset of all windows of a recording could be useful for some tasks, such as genre classification or music recommendation. It could be helpful for users to be able to either choose where in a recording such a subset of windows are to be selected, or to be able to instruct jAudio to choose the locations randomly.

Another potential improvement would be to incorporate functionality for representing extracted features more sparsely, for the purposes of conserving space and machine learning processing time as well as, potentially, increasing performance of classification by removing feature noise. One way of doing this would b to allow users to extract *buckets* of windows, where ten or so sets of windows, each spanning a few dozen or so milliseconds, are selected at random points in a recording. Since each such bucket spans only a small amount of time, the features for all windows in a bucket could reasonably be averaged together. This approach would dramatically reduce the data that would need to be stored and then processed by machine learning algorithms, but still provide hopefully representational probes at various points in each recording.

It would also be useful to provide users with a choice of windowing functions when calculating FFTs, as jAudio currently automatically applies a Hann window.

Functionality could also be added to jAudio for parsing additional audio file formats. Ogg Vorbis, AAC/MP4, FLAC and SDIF would be particularly useful choices. It could also be useful to incorporate functionality into jAudio allowing it to extract features from live audio streams.

There also plans to make minor improvements to the GUI interface, particularly with respect to the ways that aggregators are accessed and with respect to expanding the details covered by jAudio's manual. jAudio is easy-to-use and well-documented compared to most existing MIR software, but there is still certainly room for improvement.

# 4. jSymbolic: Extracting features from symbolic music

## 4.1 Introduction

### 4.1.1 Overview of symbolic feature extraction and jSymbolic

Symbolic musical file formats represent musical information in a fundamentally different way than audio files. As discussed in Chapter 3, audio files store actual sound signals by digitally approximating the signals themselves. Symbolic files, in contrast, store higher-level abstractions about music rather than direct representations of the sound itself. So, while an audio file such as an AIFF file would store an approximation of the actual sound waves produced by a bass and alto sax duet, for example, a symbolic file would store information such as the pitch of each note, which instrument played each note and the start and stop times of each note. Examples of symbolic musical representations range from sheet music written by composers and read by musicians to the holes punched in player piano rolls to modern digital file formats such as MIDI, Humdrum and MusicXML.

jSymbolic is the jMIR software application devoted to extracting features from musical data stored symbolically, specifically in MIDI files. As is the case with jAudio, jSymbolic is designed to be used directly as a feature extraction application as well as a platform for iteratively developing new features that can then be shared amongst researchers. jSymbolic also shares many of the design characteristics of jAudio that emphasize feature extensibility. This includes a modular design that facilitates the implementation and incorporation of new features, automatic provision of all other feature values to each feature and dynamic feature extraction scheduling that automatically resolves feature dependencies. Unlike jAudio, however, jSymbolic does not yet have functionality for automatically generating metafeatures and aggregators.

As is the case with all of the jMIR components, jSymbolic is designed to be accessible to users with a variety of technical skill levels, and has a simple and easy-to-use GUI. Also like all of the jMIR components, jSymbolic is implemented in platform-independent Java, and is entirely open-source and freely available.

jSymbolic includes a library of 111 implemented features that may be extracted from symbolic files. A further 42 features that remain to be implemented are also proposed in this document, for a total catalogue of 153 features, including both single-value features and multi-dimensional features. These features can be loosely divided into the following categories:

- **Instrumentation:** Which instruments are present, and which are emphasized relative to others? Both pitched and non-pitched instruments are considered.
- **Texture:** How many independent voices are there and how do they interact (e.g., polyphonic or homophonic)? What is the relative importance of different voices?
- **Rhythm:** Features are calculated based on the time intervals between note attacks and the durations of individual notes. What meter and what rhythmic patterns are present? Is rubato used? How does rhythm vary from voice to voice?
- **Dynamics:** How loud are notes and what kinds of variations in dynamics occur?
- **Pitch Statistics:** How common are various pitches relative to one another, in terms of both absolute pitches and pitch classes? How tonal is the piece? What is its range? How much variety in pitch is there?
- **Melody:** What kinds of melodic intervals are present? How much melodic variation is there? What can be observed from melodic contour measurements? What types of phrases are used and how often are they repeated?
- **Chords:** What vertical intervals are present? What types of chords do they represent? How much harmonic movement is there, and how fast is it?

The 153 jSymbolic features were originally developed through extensive analysis of publications in the fields of music theory, musicology and MIR as part of the Bodhidharma symbolic genre classification project (McKay 2004). The features have since been refined. Most of these features had not previously been applied to MIR research, and many of them are entirely novel.

The effectiveness of these features has been experimentally demonstrated by the fact that Bodhidharma placed first in all four categories of the MIREX 2005 Symbolic Genre

Classification Contest (McKay and Fujinaga 2005b; www.music-ir.org/evaluation/mirex-results/), the most recent MIREX symbolic genre evaluation. The refinement and porting of these features to jSymbolic now allows these features to be used for a range of MIR tasks beyond only genre classification.

A special emphasis was placed on ensuring the diversity of the features in the jSymbolic feature catalogue, which is part of the reason why so many features were implemented. General-purpose suites such as jMIR must be able to deal with many different types of music, and the features that might be relevant to certain types of music might be useless with respect to others. jSymbolic thus presents researchers with a large palette of features from which to choose, based on either their own musical expertise or ACE's automatic feature selection functionality (see Chapter 6).

The jSymbolic Java bytecode and the associated source code are freely available at jmir.sourceforge.net/index_jSymbolic.html. The software makes use of the Xerces XML Parser,[118] which is also freely available. There is also some initial published work available on jSymbolic (McKay and Fujinaga 2007b; McKay and Fujinaga 2006a; McKay and Fujinaga 2006c).

### 4.1.2 Benefits of extracting features from symbolic data

There has, at least to date, been much more MIR research performed on features that can be extracted from audio data than from symbolic data. This is to be expected, given that most commercial MIR applications and users in general are much more interested in processing audio files than symbolic files such as MIDI.

One must not underestimate the advantages offered by symbolic file formats, however. In stark contrast to the majority of MIR researchers, musicologists and music theorists have focused, and continue to focus, on symbolic musical representations, namely printed or written scores. There is therefore an immense existing body of knowledge that can be taken advantage of to design features that can be extracted from symbolic data and, correspondingly, many potential applications for such features.

Furthermore, the information represented by features extracted from audio data generally have little intuitive meaning to humans. So, for example, while spectral centroid

[118] xerces.apache.org

and spectral flux values extracted over a sequence of audio windows can certainly be of potential use to an automatic classification system, they are unlikely to give a music theorist or composer any musical insights or inspiration. These types of features can be referred to as *low-level features,* as they represent information about musical signals at a relatively low level of abstraction.

The features extracted from symbolic data, on the other hand, can be much more directly meaningful to humans. Features directly related to tempo or key, for example, can prove helpful in providing humans with meaningful insights on music. Such features can be referred to as *high-level features,* because they incorporate high levels of musical abstraction.

Symbolic data by its nature already fundamentally includes a high level of musical abstraction. While the fundamental unit of an audio representation is typically a sample, the fundamental unit of a symbolic representation might be a note, for example. This means that it is much easier to extract high-level musical features from symbolic representations than from audio representations. Despite ongoing efforts, few high-level features can currently be reliably extracted from arbitrary audio data. It is for this reason that one might argue that the MIR research community has made a misstep by focussing so much more on audio features than on symbolic features. Symbolic data yields much more immediate and reliable access to musically meaningful research.

It is also important to point out that research in the fields of audio feature extraction and symbolic feature extraction will likely be much more linked in the future than they are at the moment. Research on automatic transcription, which is essentially the task of generating symbolic representations from audio representations, is an important area that receives significant research interest. If future advances allow audio to symbolic transcription to be reliably performed, then existing research on symbolic feature extraction could be profitably applied to symbolic transcriptions extracted as intermediate data structures from audio files. A strong body of research on symbolic feature extraction now could thus pay dividends in the future by providing immediate access to a wide range of high-level features that could be indirectly extracted from audio data.

Although general polyphonic transcription is still very much an unsolved problem, significant achievements have already been made in areas such as monophonic transcription or piano-only transcription, and general improvements continue to be made (Klapuri and Davy 2006). Furthermore, automatic music classification tends to be much more noise tolerant than other applications of automatic audio transcription. While an audio transcription, where 10% of the notes are incorrect, might be entirely unacceptable to a human performing an automatically transcribed score, for example, such an error level would not necessarily prove to be a serious detriment to automatic classification algorithms, many of which are specifically designed to deal with such noisy data. In addition, the averaging out of features during the feature aggregation process could also effectively smooth out much of this transcription noise. So, research on high-level features performed now using symbolic data could be of significant use in the future with respect to audio data as well, perhaps even in the very near future.

Of course, feature extraction from symbolic musical data is an important area of research in and of itself, even without reference to audio feature extraction, as there are already many existing symbolic musical recordings. There is a large and active commercial and volunteer MIDI encoding community, researchers have encoded a large amount of music in other symbolic formats (see Section 4.2.1), and there is also music available in the formats used by commercial score editors such as Finale and Sibelius.

Existing optical music recognition techniques can also be used to process printed or written scores into symbolic file formats from which features can then be extracted. Most academic music libraries have many more scores than audio recordings, thus making free and legal access to the music cheaper and easier, and there are also many scores for which audio recordings do not exist or are hard to acquire. Extracting high-level features from processed scores can in fact be in some ways preferable from a musicological perspective to processing audio data, as it removes potential performance biases and errors, and focuses entirely on the artifact provided by the composer.

High-level features can also have important musicological and music theoretical research value outside the specific context of automatic music classification. For example, research has found that instrumentation is of particular importance when distinguishing between genres (McKay and Fujinaga 2005a), a conclusion that would be difficult to achieve if one were using only low-level features.

There is great potential that has barely begun to be tapped for automated research using high-level features in fields such as empirical musicology and music theory. Researchers in these fields typically currently use relatively primitive kinds of computer processing, when they use computers at all, and almost never use sophisticated machine learning tools. Software such as jSymbolic, potentially combined with pattern recognition software such as ACE, makes it possible to process much greater quantities of music than previously possible in order to search for regularities or to prove or disprove theoretical or musicological hypotheses. jSymbolic could therefore be of significant use to music researchers in the humanities as well as in MIR, and it is one of the goals of jSymbolic to encourage just such research.

Having noted all of this, it is important to recognize some of the important general limitations of symbolic data with respect to musical representation and feature extraction. One of the most significant of these limitations is that timbral data is usually only represented symbolically in a relatively coarse manner, usually via instrument identifiers. Although some additional information can in principle be represented using information such as MIDI Channel Pressure messages (see Section 4.2.2), for example, such encoding practices are not standardized, with the consequence that they are effectively meaningless if features are extracted from files coming from multiple sources. Timbral information plays an important role in human hearing, so this is a potentially serious issue.

A symbolic recording of a human performance will almost always sound significantly worse than an audio recording of the same performance. This is partly because it is impossible to properly record the full range of control parameters for most instruments, which is significant from the perspective of analytical feature extraction, and partly because of synthesizer limitations, which is less significant from this perspective.

An additional problem is that most symbolic formats were devised primarily with Western musical models in mind. This can limit their applicability to certain non-Western musics, although certainly not all. For example, MIDI cannot represent instruments that are not specified in General MIDI in a standardized way, although MIDI can represent microtonal pitch intervals.

Despite all of this, the increased accessibility of high-level features in symbolic formats, as discussed above, remains a very significant asset. Symbolic formats such as

MIDI also have a number of additional advantages over audio recordings. For example, they are typically much more compact, which in turn makes them easier to store or transmit, and much faster to process and extract features from. Symbolic recordings are also usually relatively simple to visualize in human-meaningful ways, such as via standard sheet music notation (as opposed to audio waveforms or spectrograms), and they can also be edited entirely non-destructively.

Symbolic data also makes it easier to extract meaningful features relating to entire recordings, such as key or meter, for example. Audio data, in contrast, typically has to be segmented into analysis windows prior to feature extraction, as explained in Chapter 3. This results in an explosion of data and processing complexity, where features extracted from individual windows may be overly noisy because of their short duration, and averaging out these features over many windows may smooth out some potentially meaningful information.

As an additional note, the loss of timbral data beyond instrumentation in symbolic formats may not be as significant as it may seem at first. For example, Aucouturier and Pachet (2003) suggest that the relationship between timbral features and musical genres may not be as strongly correlated as one might imagine. It is, of course, true that timbre clearly does play a role in many kinds of human auditory classification, but Aucouturier and Pachet's work still provides an indication that the importance of timbre may have been overemphasized in past research, and that other types of musical information may well make timbre unnecessary for some kinds of automatic classification. The fact that timbre-based features have played a sometimes overwhelming role in many of the existing audio classification systems may have even potentially impaired their performance.

Overall, one can say that audio encodings have certain significant advantages over symbolic encodings, and that symbolic encodings likewise have certain significant advantages over audio encodings. This is precisely why jMIR includes both jSymbolic and jAudio, so that the advantages offered by both approaches to encoding musical information may be taken advantage of, whether one searches out matching symbolic and audio versions of a particular piece, or whether transcription technology or synthesis technology is respectively used to generate one type of encoding based on the other.

### 4.1.3 Chapter outline

Section 4.2.1 of this document provides a brief overview of some of the most relevant symbolic musical file formats. Section 4.2.3 includes justifications for the particular choice of MIDI as the format from which jSymbolic extracts features. A relatively detailed description of the MIDI specification is also provided in Section 4.2.2, in order to ensure that the reader is fully prepared for the feature descriptions provided in Section 4.5.

Existing software that can be used for symbolic feature extraction is reviewed in Section 4.3. The main focus in this section is on software specifically designed for feature extraction, although some particularly pertinent tangential research is also mentioned.

Section 4.4 reviews and integrates research from several fields in order to arrive at general principles to consider when designing and choosing features to extract from symbolic data. The focus of this section is primarily on theoretical research in disciplines related to music theory and musicology, although this is not exclusively the case.

The features that are proposed as part of the jSymbolic feature catalogue are described individually in Sections 4.5.1 to 4.5.7. These features are grouped into the categories described above in Section 4.1.1 for the purpose of clarity, and include both the 111 features that are implemented in jSymbolic as well as the 42 features that remain to be implemented. For the purpose of illustration, Section 4.5.8 includes two short musical excerpts and examples of some of actual feature values extracted from these excerpts.

Section 4.6 outlines the basic functionality offered by jSymbolic, while the software's interface itself is described in Section 4.7. A summary of the original research contributions of this chapter is presented in Section 4.8, and some final ideas for future research in symbolic feature extraction and expansion of jSymbolic's functionality are provided in Section 4.9.

## *4.2 Background information*

### 4.2.1 Symbolic music file formats

It is generally possible to divide digital symbolic file formats into three broad categories: formats intended to communicate performance information between gestural controllers, computers and synthesizers; formats intended to represent musical scores and associated visual formatting information; and formats intended to facilitate theoretical and musicological analysis.

MIDI (MIDI Manufacturers Association. 2001; Rothstein 1995), which falls primarily into the first of the three categories described above, is the symbolic file format that is by far the best known by the general public. As a result, there is a very large quantity of music of many kinds that has been encoded in MIDI. Ironically, MIDI was originally intended as a real-time protocol for communication between instrumental controllers and synthesizers, and did not originally have any file format specification. Nonetheless, its relatively early arrival and wide industry adoption has led to its broad popularity as a format for storing symbolic musical data. More details on MIDI are provided in Sections 4.2.2 and 4.2.3.

Open Sound Control (Wright and Freed 1997), or OSC, is another real-time performance-oriented symbolic file format, and was designed specifically as a successor to MIDI. It is widely recognized as technically superior to the essentially obsolete MIDI protocol, but the wide industry penetration of MIDI has prevented OSC from making significant progress in replacing MIDI. Some of the advantages of OSC include explicit compatibility with modern networking technology and improved general flexibility and organization, as well as technical advantages such as improved time resolution.

The most commonly used file formats for representing formatted musical scores are likely the native file formats of the two dominant score editing applications, namely the Sibelius[119] .SIB format and the Finale[120] .MUS formats. Unfortunately, these are closed formats, which means that the details of how they represent scores are not openly published, so one must buy the associated software in order to read or write these formats. Furthermore, even if reverse engineered, the file standards may be changed unilaterally at any time by the software developers, something that Finale in particular is notorious for doing between versions of their score editing application. Problems such as these significantly limits the research value of these file formats.

[119] www.sibelius.com
[120] www.finalemusic.com

Fortunately, there are a variety of more research-oriented formats that can be used to represent musical scores, including GUIDO (Hoos et al. 2001) and Lilypond (Nienhuys and Nieuwenhuizen 2003), two relatively well-known text-based formats.

MusicXML (Guess 2001), an open XML-based format, has achieved a relatively high profile, particularly due to its adoption by a variety of commercial music notation programs, including Finale, Sibelius and the Steinberg Cubase[121] music sequencer. The use of MusicXML as an intermediate file format is currently the best approach available for transferring data between .SIB and .MUS files.

In the realm of music analysis, the file formats associated with the Humdrum Toolkit (Huron 2002) are particularly prominent. These formats include the fairly general **Kern format (Huron 1997) as well as formats intended to represent more specialized types of music, such as the **Hildegard[122] neume encoding or the **Koto[123] format for koto tablature.

There are many other symbolic file formats that have been developed, as well as a number of published guidelines for developing such formats. Although there is insufficient space to go into more detail here, Dannenberg (1993) and Selfridge-Field (1997) have provided excellent overviews of the classic symbolic formats.

### 4.2.2 The MIDI specification

MIDI is an encoding system that is used to represent, transfer and store symbolic musical information. As mentioned above, the MIDI standard was originally developed as a real-time communication protocol between instruments and synthesizers. However, even though the notion of a MIDI file was not part of the original MIDI specification, MIDI files are now very much part of the standard and are broadly used.

Only those parts of the MIDI specification that are relevant specifically to symbolic feature extraction are discussed in any kind of detail in this section. Those aspects of MIDI that are only relevant to live performance and not to MIDI files are for the most part omitted here due to the breadth of the MIDI specification. There are many books on MIDI, such as that by Rothstein (1995), which can be consulted for further information

[121] www.steinberg.net
[122] www.humdrum.org/Humdrum/representations/hildegard.rep.html
[123] koto.sapp.org/kotospec/

on MIDI if desired. The complete MIDI specification is published by the MIDI Manufacturers Association (2001).

Like all symbolic file formats, MIDI files store abstract representational instructions that can be sent to synthesizers or score editing software, and not actual sound samples. The quality of the sound that is produced when a MIDI file is played is therefore highly dependant on the particular synthesizer that the MIDI instructions are sent to.

MIDI essentially consists of sequences of instructions called *MIDI messages*. Each MIDI message corresponds to an event or change in a control parameter. MIDI messages each consist of one or more bytes of data, which fall into two types: *status bytes* and *data bytes*. The status byte is always the first byte of a MIDI message, always starts with a 1 bit and specifies the type of MIDI message and, implicitly, the number of data bytes that will follow to complete the message. Data bytes always start with a 0 bit, which means that each data byte has 7 remaining bits to specify values with ranges between 0 and 127.

MIDI allows the use of up to sixteen different *channels* on which different types of messages can be sent. Each channel operates independently of the others for most purposes. Channels are numbered from 1 to 16. There is no channel 0.

There are two important classes of MIDI messages: *channel messages* and *system messages*. The former influence the behaviour of only a single channel and the latter affect the MIDI system as a whole. *Channel voice messages,* a type of channel message, are the only type of messages that are relevant to symbolic feature extraction. The four least significant bits of the status byte of all channel voice messages indicate the channel number (0000 is channel 1 and 1111 is channel 16). *Note On, Note Off, Channel Pressure, Polyphonic Key Pressure, Program Change, Control Change* and *Pitch Bend* messages are all channel voice messages, as described below.

A *Note On* messages instructs a synthesizer to begin playing a note. This note will continue playing until a matching *Note Off* message is received. The four most significant bits of the status byte of all Note On messages must be 1001 and, as with all channel voice messages, the four least significant bits specify the channel on which the note should be played. Note On messages have two data bytes: the first specifies pitch, from 0 to 127, and the second specifies *velocity,* also from 0 to 127. Pitch is numbered in semitone increments, with note 60 being designated as middle C. Equal temperament tuning is used by default, although synthesizers can be instructed to use alternate tunings if desired. The velocity value specifies how hard a note is struck, which most synthesizers map to loudness.

A Note Off message has an identical format to a Note On message, except that the four most significant bits of the status byte are 1000. The pitch value specifies the pitch of the note that is to be stopped on the given channel and the velocity value specifies how quickly a note is to be released, which is generally mapped to the fashion in which the note dies away. Many synthesizers do not implement Note Off velocities, however. A Note On message with velocity 0 is equivalent to a Note Off message for the given channel and pitch.

*Channel Pressure* messages specify the overall pressure for all notes played on a given channel. This can be mapped by synthesizers in a variety of ways. *Aftertouch* volume (the loudness of a note while it is sounding) and vibrato are two common mappings. The status byte of Channel Pressure messages has 1101 as its four most significant bits, and there is one data byte that species the pressure (between 0 and 127).

*Polyphonic Key Pressure* messages are similar to Channel Pressure messages, except that they contain an additional byte (the one immediately following the status byte) that specifies pitch, thus restricting the effect of the message to single notes rather than to all notes on the channel. The most significant bits of such a status byte are 1010.

*Program Change* messages allow one to specify the instrumental timbre that is to be used to sonify all future notes played on the specified channel until a new program change message is received for it. *Program, patch* and *voice* are all terms that can be used equivalently in reference to the instrumental timbres specified by Program Change messages. The most significant bits of the status byte are 1100, and there is a single data byte specifying the patch number, from 0 to 127.

The particular 128 instrumental timbres and sound effects corresponding to particular patch numbers are specified by the *MIDI Program Table,* which is part of an addendum to the MIDI specification called *General MIDI.* The MIDI Program Table is relevant to all channels except channel 10. All notes played on channel 10 are considered to be percussion notes, and General MIDI includes a separate *Percussion Key Map* specifying 47 percussion timbres that are always used for notes sent to channel 10. The timbre that is used for each note played on channel 10 is determined based on the pitch value of its corresponding Note On message, not by Program Change messages. The pitch value of Note On messages on all other channels specifies pitch, and has nothing to do with instrumental timbre. Rothstein (1995) provides copies of both the MIDI Program Table and the General MIDI Percussion Map.

*Control Change* messages affect the sound of notes that are played on a specified channel in various ways. Volume and modulation are two parameters that are often controlled via Control Change messages. In total, there are 121 different MIDI controllers that may be accessed via Control Change messages, including some that are unspecified in the standard. However, many synthesizers do not implement most, or even any, of these controllers. The four most significant bits of the status byte of Control Change messages are 1011. The first data byte specifies the controller that is being referred to, from 0 to 120. There is a second data byte that specifies the setting of the controller, which can range from 0 to 127.

If a greater resolution is required, a second Control Change message may be sent to supplement the first, resulting in a control resolution of 16,384, as opposed to the standard resolution of 128. Controllers 0 to 31 represent the most significant byte in this case, and controllers 32 to 63 represent the least significant byte. Two Control Change messages can thus cause a single change to be implemented with much greater resolution than a single Control Change message. Rothstein (1995) provides details on particular Control Change messages.

Control Change messages are generally intended for use with continuous gestural instrumental controllers, and are only standardized to a limited extent. Control Change messages are often absent in MIDI files, and their lack of standardization could cause a good deal of noise in extracted feature values. They are therefore only utilized by jSymbolic in a very limited capacity.

*Pitch Bend* messages allow microtonal synthesis. The four most significant bits of the status byte of such messages are 1110. There are two data bytes, the first of which specifies the least significant byte of the Pitch Bend and the second of which specifies the most significant byte. Maximum downward bend corresponds to data byte values of 0 followed by 0, centre pitch (no bend) corresponds to values of 0 followed by 64 and maximum upward bend corresponds to values of 127 followed by 127. General MIDI specifies that the default Pitch Bend range is plus or minus two semitones. This setting can be altered on individual synthesizers, however, so one must be careful to ensure that the Pitch Bend range that is actually played corresponds to what is desired.

MIDI timing is controlled by a clock that emits *ticks* at regular intervals. Clock rates are usually related to temporal note durations in terms of parts per quarter note, or *ppqn*. A greater ppqn corresponds to a greater rhythmic resolution, which allows the representation of notes that push or lag the beat a little. A greater resolution also allows one to represent complex tuplets with a greater precision. The most commonly used resolution is 24 ppqn, which allows sufficient resolution to permit 64th note triplets. At 24 ppqn, a half note corresponds to 48 ticks, a quarter note to 24 ticks, an eighth note to 12 clicks, at sixteenth note to 6 clicks, etc. It should be noted that the actual speed of playback of quarter notes is controlled by *tempo change meta-events* (see below).

MIDI also permits an alternative to the ppqn representation of time, namely the *SMPTE time code* (Society of Motion Picture and Television Engineers 1994), which is divided into hours, minutes, seconds and frames. This is very useful when synchronizing MIDI to visual media such as film or video. There are variations of SMPTE for different frame rates. Thirty frames per second is the rate that is most commonly used by MIDI.

Although there are a variety of proprietary file formats that were used in the early history of MIDI to store MIDI data, nowadays MIDI is almost always encoded in one of the three open standard MIDI file formats, numbered 0, 1 and 2 (MIDI Manufacturers Association 2001). The main difference between these three formats is the manner in which they deal with *tracks,* which can be used by sequencers to segment different voices. *Format 0* files consist of a single multi-channel track, *Format 1* files have multiple tracks that must all have the same meter and tempo (the first track contains the tempo map that is used for all tracks), and Format 2 files have multiple tracks, each with its own tempo and meter. Format 1 files are the most commonly used format today, and Format 2 files are still rarely used.

Each of the three standard MIDI file types consist of groups of data called *chunks,* each of which consist of a four-character identifier, a thirty-two bit value indicating the length in bytes of the chunk and the chunk data itself. There are two types of chunks: *header chunks* and *track chunks*.

Track chunks contain all of the information and MIDI messages specific to individual tracks. The header chunk is found at the beginning of the file and includes the type of file format (0, 1 or 2), the number of tracks and the *division*. The division value can mean one of two things, depending on whether it specifies the use of either ppqn timing or SMPTE. In the former case, it specifies the timing resolution of a quarter note. In the latter case, it specifies the SMPTE frame rate and the number of ticks per SMPTE frame.

Time intervals between MIDI events are notated using *delta times,* which specify the amount of time that has elapsed between the current event and the previous event on the same track. This approach is used because it requires less space than simply listing the absolute number of MIDI ticks that pass before an event occurs.

MIDI messages stored in files and their associated delta times are called *track events*. Track events can involve both MIDI events and *meta-events*. Meta-events provide the ability to include information such as lyrics, key signatures, time signatures, tempo changes and track names in files.

Key signature meta-events include two pieces of information: *sf* and *mi*. sf indicates the number of flats (negative number) or sharps (positive number) in the key signature. For example, C major and A minor are represented by 0 (no sharps or flats), 3 represents 3 sharps (A major or F# minor) and -2 represents 2 flats (Bb major or G minor). *mi* indicates whether the piece is major (0) or minor (1).

Time signature meta-events contain four pieces of information: *nn, dd, cc* and *bb*. nn and dd are the numerator and denominator of the time signature respectively. It should be noted that dd is given as a power of 2, so a dd of 3 corresponds to 2^3 = 8. Thus, a time signature of 5/8 corresponds to nn = 5 and dd = 3. cc is the number of MIDI ticks in a metronome click and bb is the number of 32nd notes in a MIDI quarter note.

Tempo change meta-events consist of three data bytes specifying tempo in microseconds per MIDI quarter note. If no tempo is specified then the default tempo is 120 beats per minute.

### 4.2.3 Reasons for choosing MIDI as the format from which to extract features

Ideally, one would prefer to have a feature extractor that could extract features from music encoded in any symbolic file format. Unfortunately, the development of parsing functionality for each format can be a labour intensive process. Furthermore, certain formats cannot represent information as well as other formats, or at all. For example, one format might be able to represent dynamics, but not another format, or one format might be able to represent precise time values while another format might only be able to represent strictly quantized rhythmic values. Issues such as this can cause significant variations in extracted feature values that are dependent only on the encoding format, not the music itself, something that can be problematic during machine learning.

It was therefore decided to only implement parsing functionality for a single file format in jSymbolic, although one of the long-term development goals of jSymbolic is to develop functionality for parsing more formats. Specifically, the MIDI format was chosen.

At first glance, the particular choice of MIDI may seem an odd one. MIDI is in many ways an obsolete format with a number of established fundamental weaknesses that has survived only because it is a widely supported standard. There is a relatively low threshold on the amount of information that MIDI can encapsulate, for example, and it uses grossly outdated transmission protocols. Furthermore, it can be difficult and time consuming to properly record sophisticated synthesis instructions in MIDI, with the result that many encoders do not bother to do so. Many of the formats described in Section 4.2.1 have important advantages over MIDI, and OSC in particular is generally accepted as technically superior in almost every way.

So, why choose MIDI for jSymbolic? The answer lies principally in the fact that MIDI is still the most generally popular and widely used format. A good feature extractor should be able to extract features from the format that is the most used and in which the most music is encoded, not in a format that one might ideally prefer to be the most popular. There is more music encoded in MIDI than in any other symbolic format, including music in a wide range of popular, art and folk genres. Such variety is essential if a feature extractor like jSymbolic is to be widely applicable. Most other symbolic formats tend to only have music belonging to a few specific genres encoded in them.

MIDI also has the advantage that it is a performance-oriented format, and can thus encode information such as relatively precise timings and dynamics as well as microtonal pitches. This is not true for all symbolic formats, in particular those intended primarily for representing sheet music or for performing theoretical analyses, since these kinds of precise representation are not as significant in these domains. This makes it more appropriate to write translation algorithms to generate MIDI from other formats, since the reverse could result in the loss of feature-relevant information.

Having noted this, it is also true that the popularity of MIDI has been slowly waning in the past decade. There were large and very active MIDI encoding communities in the 1980's and 1990's that encoded a broad range of music into MIDI. Although these communities still exist and are still active, one now sees less new music encoded into MIDI than was previously the case.

Also, the emphasis of symbolic feature extraction will likely switch to machine generated files as polyphonic transcription technology improves in the future. It will be appropriate to work with a file format that is superior to MIDI at that point, such as OSC, since format popularity will no longer be an overriding concern if human encoders are removed from the equation. A goal for future development will therefore be the implementation of parsing libraries for additional file formats as recordings of many different kinds of music encoded in them become available.

## *4.3 Existing software related to symbolic feature extraction*

Most MIR feature extraction research to date has focused on audio data rather than symbolic data, as mentioned above. Although there is a significant amount of existing software designed for processing symbolic musical data, a relatively small proportion of it has emphasized feature extraction in particular, or the specific needs of MIR. Most of the existing systems emphasize either musical performance or automated search or analysis algorithms designed to assist very specific kinds of musicological or music theoretical research. Unfortunately, most of these systems generate only a relatively small amount of information that can be adapted for use as features in pattern recognition systems, and most of them also make musical assumptions that are only appropriate for certain types of music, such as classical music. Such systems also tend to emphasize search-oriented tasks and basic string comparisons rather than the broader types of information extraction processing that are most relevant to MIR research. Nonetheless, there are a few standout systems of this type that should be discussed before discussing work more directly related to symbolic feature extraction.

The Humdrum toolkit (Huron 2002) is perhaps the best-known of these systems, with its variety of query tools and its specialized musical representations. Although feature extraction is certainly not the software's primary intended purpose, Humdrum data has at times been used specifically for the purpose of feature extraction, such as in the work of Sapp, Liu and Selfridge-Field (2004). Knopke (2008) has written new implementations of much of the Humdrum software that makes it more usable for MIR applications, including functionality such as MIDI-compatible file format translation functionality.

The Melisma Music Analyzer (Temperley 2001) is another important analysis-oriented software system, with its tools for extracting information relating to meter, phrasing, contrapuntal structure, harmony, pitch spelling and key.

The RUBATO system (Mazzola and Zahorka. 1994) is a platform that was developed for extracting information from MIDI and other symbolic files. Although this relatively early system is based on the obsolete NEXTSTEP environment, and oriented towards music analysis rather than feature extraction, it nonetheless extracts some high-level information that could be used as features.

The MIDI Toolbox (Eerola and Toiviainen 2004) is one of the few systems that offers standardized and easily accessible symbolic processing software oriented towards MIR research. It includes tools for visualizing and processing MIDI files, including filtering functions and cognitively inspired analytical tools relating to melodic contour, similarity, key-finding, meter-finding and segmentation. Although this software is an excellent tool in general, it is not ideally adapted specifically for feature extraction. It is also a Matlab toolbox, which carries the advantage of making it directly compatible with all of Matlab's associated functionality, but also the disadvantage of requiring that the user have access to the Matlab platform, which can be expensive, and which requires familiarity with a fairly technical platform.

There are also a number of software systems that have been developed for use in specific research experiments, rather than for general application, but which nonetheless contain implementations of a number of useful features. Aarden and Huron (2001), for example, have performed an interesting study where corresponding characteristics of European folk songs were studied in terms of spatial location. A potentially very useful catalogue of sixty high-level features was used, although the system was limited to monophonic melodies. To give another example, Towsey et al. (2001) developed a system for compositional purposes that uses 21 high-level melodic features that have application beyond the aesthetic fitness judgements that they were used for in the original study.

Some of the most significant software directly related to symbolic feature extraction has been developed as part of genre, style, mood or other classification projects. Although the feature extraction software systems used in many of these research projects are not publicly available, or are not designed to be used easily outside of the particular projects for which they were designed, some excellent features have been developed as part of these projects. Since these features have actually been implemented in software and successfully applied experimentally to music, it is appropriate to provide highlights of this research here.

Some of the most relevant work of this kind is Ponce de León and Iñesta's (2002) system for processing MIDI tracks in order to extract melodic, harmonic and rhythmic features. The system used these features to form distinguishable categories using self-organising maps. This work has since been expanded (Ponce de León and Iñesta 2007; Rizo et al. 2006).

Ruppin and Yeshurun (2006) developed several features relating specifically to melodic contour. Of particular interest, these features were designed to be invariant to some of the transformations that one commonly finds in music, such as transpositions and rhythmic prolongation.

Twenty features were implemented by Backer and van Kranenburg (2005) in the context style classification. These features are for the most part only meaningful with respect to polyphonic compositions with clearly separated voices, however.

Basili, Serafini and Stellato (2004) used only five features, but were also among the earliest researchers, along with McKay (2004), to use instrument information extracted from symbolic data in their features. Some of these features are multi-dimensional. The five features are *Melodic Intervals, Instruments, Instrument Classes and Drumkits, Meter/Time Changes* and *Note Extension.*

Lin and his colleagues (2004) performed genre classification experiments by transforming sequences of MIDI data into intermediate representations based on rhythm and pitch that can also be treated as features. The authors then processed this data to break it into repeating patterns. Karydis, Nanopoulos and Manolopoulos (2006) also made the notion of repeating patterns an essential part of their feature extraction approach, with respect to pitch and duration information in particular.

Shan and Kuo (2003) extracted features based exclusively on melodies and chords for the purpose of automatic genre classification. This research is particularly valuable in terms of the ways in which melodic and chordal features were extracted.

Chai and Vercoe (2001) used hidden Markov models to classify monophonic melodies belonging to one of three different types of Western folk music (Austrian, German and Irish). They were able to achieve 63% accuracy in three-way classifications that used only melodic features. Interestingly, to include a machine learning note, they found that the number of hidden states had only a relatively minor effect on success rates, and that simple Markov models outperformed more complex models.

Dannenberg, Thom and Watson (1997) designed a real-time system to classify performance styles. Improvisations were classified as *lyrical, frantic, syncopated, pointillistic, blues, quote, high* and *low.* The system was trained with MIDI recordings of trumpet performances. The following features were extracted from the MIDI data: averages and standard deviations of MIDI key numbers, durations, duty factors (the ratio of duration to inter-onset interval), pitches (which differ from key number in that Pitch Bend information is included) and volume levels, as well as counts of notes, Pitch Bend messages and volume change messages.

Lartillot et al. (2001) used two types of unsupervised learning to classify recordings based on musical style. This was done using analyses of musical sequences in terms of rhythm, melodic contour and polyphonic relationships.

Anagnostopoulou and Westerman (1997) also applied unsupervised learning to symbolic music. They used features based on melodic shape, rhythmic movement, interval patterns and instrument register.

Some of the earliest research on automatic genre classification was published by Gabura (1965). This paper only deals explicitly with classical music, however, which limits its applicability. Despite this, and the its age, this paper nonetheless offers some interesting ideas that appear to have been overlooked in many later publications, particularly in regard to the use of relatively sophisticated statistics and theoretical models to derive features.

Research has also been done in other specific areas of MIR that incorporated interesting features that can be adapted for other purposes. To give one relatively early example, Blackburn and De Roure (1998) implemented a system for classifying the musical parts of MIDI files, with a focus on pitch class methods.

Maxwell and Eigenfeldt (2008) implemented a symbolic query system that includes a number of searchable *type* parameters that could also be used as features in classification tasks. Bergeron and Conklin (2008) also developed an approach that they applied to retrieval from symbolic music that includes a number of useful simple features. Kirlin and Utgoff (2005) have also extracted simple features as part of query research, and Conklin and Bergeron (2008) have published associated research that is more specifically related to feature extraction.

Volk et al. (2008) considered a large number of features in the context of evaluating musical similarity. These features could also be adapted for other purposes, although some of them are subjective. Meudic (2003) has also proposed some important similarity-related measures that could also be used for other purposes.

There is also a significant body of work that is highly relevant to symbolic feature extraction, particularly from a high-level perspective, that has not involved actual software implementations of feature extraction algorithms. Such work is discussed in Section 4.4 below.

## *4.4 Considerations when designing and choosing features*

This section reviews and integrates research from several fields in order to arrive at general principles to consider when designing and choosing features to extract from symbolic data. The focus of this section is primarily on theoretical research in disciplines related to music theory and musicology, although research from other disciplines is discussed as well.

There has been some important applied work done in more technical disciplines that has involved not only designing features, but actually implementing automatic extraction algorithms. Such work is reviewed in Section 4.3. It should be noted that most of the corresponding publications do not provide theoretical support or motivations for the features that they use, and almost none of them provide guidelines for implementing new features, unlike the majority of the work reviewed in this section. Chapter 2 also reviews some general research in psychology and the humanities that relates to how humans form categories and make classifications, some of which is relevant to symbolic feature design.

People often claim that they “do not know what to listen to” when they are first exposed to an unfamiliar type of music. This demonstrates how listening methodologies that apply to one type of music may be of little value when applied to another type of music, and how difficult basic listening tasks can be in terms of feature extraction in such situations. Consequently, it seems reasonable that it would be useful for feature extraction software to make a broad range of features available, in order to at least have reasonably sized feature subsets that will be applicable to each of as many different types of music as possible.

This is one of the reasons that so many features are implemented in jSymbolic, as described in Section 4.5. Although machine learning issues relating to the curse of dimensionality (see Section 6.2.3) generally make it unwise to provide pattern recognition systems with too many features, having a very large feature catalogue available makes it possible to choose those particular features that are meaningful and relevant to both a specific given task and to the specific types of music that are under consideration. This selection of the particular features to use for a particular project may either be made by humans with expert musical knowledge, or by automatic dimensionality reduction algorithms (see Chapter 6.2.5). Conversely, having too few features implemented by the feature extraction software could make it impossible for pattern recognition software to achieve adequate success rates, because there might simply not be sufficient information encapsulated in the features that are available to discriminate properly between classes. Providing a large catalogue of features from which to choose also makes features available for future applications whose feature needs might not currently be understood or anticipated.

Nonetheless, having presented the case that it is useful to design many features, it is also important to point out that care should still be taken to avoid features that are unlikely to be useful. Too many features can overwhelm even sophisticated feature selection methodologies, be they manual or automatic, and only finite time and computational resources are available for feature development and extraction. It is therefore reasonable to prioritize those features that are likely to be the most promising, with the understanding that one would ideally like features that encapsulate as much information as possible that is relevant to as many types of music and to as many classification applications as possible. Features that represent information in ways that are easy for pattern recognition algorithms to process should also be prioritized.

It is clear that even people with little formal musical knowledge can perform sophisticated pattern recognition tasks like artist identification or the recognition of stylistically “wrong” notes. It might therefore be argued that music classification systems should pay special attention to features that are meaningful to the average, musically untrained listener, and that features based on more technical or sophisticated musical properties may be overkill.

It could also be argued, however, that the ultimate goal of a classification system is to produce a correct classification, and that whatever readily available features help to do this should be used, whether or not they are perceptible or meaningful to the average human listener. The fact that some expert musical skills, such as the ability to precisely and consciously perceive detailed rhythmic or harmonic characteristics, may not be necessary to distinguish between musical categories does not mean that such information might not be helpful to both humans and computers. In addition, pattern recognition systems operate using significantly different mechanisms than their human analogs, so there is no reason to assume that the types of percepts suited for one are necessarily best suited to the other. Since moderately high-level musical information is relatively readily available from symbolic formats such as MIDI, it might as well be taken advantage of. Having said this, it is also of course very important to take advantage of the kinds of information that the average human listener can in fact use, since “unskilled” listeners are demonstrably able to use it to perform successful classifications.

A related issue is the question of exactly how high level one’s features should be. Although it seems reasonable that one could glean meaningful features from fairly high-level information like chord progressions or melodic arcs, for example, should one also consider even higher-level musical concepts based on sophisticated analytical frameworks? There is a great deal of existing literature on music theory that could be used to design such features. The books of Cook (1987) and LaRue (1992), to give just two of many examples, provide good complementary surveys of analytical methods that could potentially be of use.

Ideally, one would like to have a grand unified analytical methodology that could be used to extract meaningful features from any type of music. Unfortunately, there is no generally accepted process of this sort. However, even though no single analytical system is complete, and most are only applicable to a limited range of musical genres, several systems could nonetheless be used in a complementary or parallel way. For example, Schenkerian analysis could be used to analyze harmony if other features indicate a significant degree of tonality in a recording, complimented perhaps by a set theory analysis to deal with non-tonal music. To extend this example, techniques such as those of Cooper and Meyer (1960) could also be used to analyze rhythm and melody, and the techniques of Reti (1951) could be used to gain insights by looking at motivic patterns. Semiotic analysis (Tarasti 2002) could also potentially be useful, although somewhat difficult to implement automatically from a content-based perspective. Analytical approaches that take advantage of insights from fields such as cognitive science and linguistics, such as the work of Lerdahl and Jackendoff (1983), could also be used.

Multi-purpose existing automatic analysis systems, such as some of those described in Section 4.3, could be taken advantage of in order to facilitate this process. Even though these types of analyses are intrinsically different and typically built on disjoint or even incompatible musical assumptions, they could nonetheless each be used to generate individual features that could prove to be complementary.

There are, unfortunately, a number of disadvantages with using such an approach of combining sophisticated analytical systems for use in feature extraction. To begin with, many of these techniques require a certain amount of intuitive subjective judgement, as the rules specified in analytical models are sometimes vague or ambiguous. This is demonstrated by the inconsistencies that one often encounters between analyses of a single piece by different people using the same analytical system. Another problem is that sophisticated theoretical analyses are often computationally expensive, thus making their use inappropriate for very large musical collections or real-time classification applications.

In addition, most analysis techniques have been designed primarily from the perspective of Western art music, which limits their applicability to popular and non-Western musics. For example, feature sets based too heavily on chord progressions could incorporate too many assumptions relating to tonal harmony to be applicable to types of music that do not operate on a tonal basis. This general problem, however, may be less crippling than it seems, as analyses could still be generated that are internally consistent, even if the analytical systems themselves may not be fundamentally meaningful to a given type of music. Future experimentation is necessary to investigate whether features based on such theoretically compromised analyses could nonetheless be useful for performing classifications.

In any event, a generally accepted software system capable of quickly performing a wide range of sophisticated theoretical analyses has yet to be implemented. One must therefore make do, at least for the moment, with occasionally taking simple and incomplete concepts from a range of analytical systems, of necessity somewhat haphazardly, and combining them with intuitively derived musical characteristics in order to arrive at suitable features. Features that are used for classification purposes need not be consistent or meaningful in any overarching theoretical sense. All that matters for the purpose of classification is that each feature helps to distinguish between relevant classes, which is to say that they need only represent characteristics that consistently differ statistically between classes. It is certainly nice if one can also use features to derive musically meaningful insights, but this is by no means a necessity from the perspective of applied automatic classification.

In any case, as discussed above, most humans are certainly unable to perform sophisticated theoretical analyses, but are nonetheless able to perform sophisticated classification tasks like genre recognition. It is therefore clear that such analyses are not strictly necessary in order to implement a successful automatic classification system. Furthermore, a study of how well children of different ages can judge the similarities of music belonging to different styles found that there was almost no difference between the success rates of eleven-year olds compared to college sophomores, despite the fact that, unlike the college students, the eleven-year olds displayed almost no conscious knowledge of musical theory or stylistic conventions (Gardner 1973). To provide an additional supporting experimental finding, Perrott and Gjerdingen (1999) found that humans with little to moderate formal musical training are able to make genre classifications agreeing with those of experts 72% of the time (among a total of 10 genres) based on only 300 milliseconds of audio. This is far too little time to perceive musical form or structure, which are key aspects of most sophisticated analytical systems. This suggests that there must be a sufficient amount of information available in very short segments of music to successfully perform classifications, and that more sophisticated analyses may not be strictly necessary. Of course, this does not necessarily mean that features based on sophisticated analytical systems might not eventually be useful to an automated classification system, but it does appear that they are not absolutely necessary, which is fortunate, given the difficulty that is currently involved in extracting them.

In the case of jSymbolic, it was decided to emphasize simple features that could arguably be immediately perceptible to musically well-trained humans, but would not require them to make extended analyses. Well-trained humans are more likely to be skilled music classifiers, so the features that they use are therefore more likely to be useful in general. This does not mean that other features were ignored in the jSymbolic music library, however, as there are very likely other important discriminating characteristics of music that musical experts are not consciously aware of. This approach provides a good compromise between music theoretical simplicity and sophistication that permits the development of many potentially useful features, while helping to avoid the development of features that would be too computationally expensive to extract, or of irrelevant features that would introduce unproductive noise into classification systems.

Once such a design philosophy is decided upon, the next step is to consider relevant research by musicologists and music theorists in order to search for inspiration for features that might be of particular use. Ethnomusicologists in particular have done significant work on comparing the musics of different cultures, something that is important to consider if one wishes to make features available that are likely to have a high discriminating power outside the limited scope of Western art music. An additional advantage of ethnomusicological research is that it tends to focus more on empirical observation than other musicological or music theoretical disciplines, something that is very helpful from the perspective of feature design. Empirical ethnomusicological research also tends to be less likely to be intrinsically tied to specific types of music or to limiting theoretical assumptions.

Perhaps the most extensive work in this vein was performed by Alan Lomax and his colleagues as part of the Cantometrics project (Lomax 1968). Although this work has been criticized on a number of political fronts (Nattiez 1990), it remains valuable from the perspective of feature design for automatic classification systems. This project compared several thousand songs from hundreds of different cultural groups based on thirty-seven features that were extracted from audio recordings by hand:

- **Leader chorus:** the importance of the lead singer relative to the chorus.
- **Relation of orchestra to vocal part:** importance and independence of the orchestra relative to the vocal part.
- **Relation within orchestra:** relative independence of the different parts of the orchestra.
- **Choral musical organization:** texture of the choral singing.
- **Choral tonal integration:** degree to which the chorus blends singing together to create the perception of unity and resonance.
- **Choral rhythmic organization:** degree of rhythmic coordination of the chorus.
- **Orchestral musical organization:** texture of the orchestra.

- **Orchestral tonal concert:** degree to which the orchestra blends together to create the perception of sonority.
- **Orchestral rhythmic concert:** degree of rhythmic coordination of the orchestra.
- **Text part:** whether singers tend to use words or other sounds. Also measures amount of repetition of text.
- **Vocal rhythm:** complexity of meter used by singers.
- **Vocal rhythmic organization:** degree to which singers use polyrhythms.
- **Orchestral rhythm:** complexity of meter used by orchestra.
- **Orchestral rhythmic organization:** degree to which orchestra uses polyrhythms.
- **Melodic shape:** melodic contour of most characteristic phrases.
- **Melodic form:** complexity of form.
- **Phrase length:** temporal length of phrases.
- **Number of phrases:** average number of phrases occurring before full repeats.
- **Position of final tone:** position of the final pitch relative to the range of the song.
- **Range of melody:** pitch interval between the lowest and highest notes of the song.
- **Average interval size:** average melodic interval.
- **Type of vocal polyphony:** type of polyphony present, ranging from a drone to counterpoint.
- **Embellishment:** amount of embellishment used by the singer(s).
- **Tempo:** speed of song from slow to very fast.
- **Volume:** loudness of song.
- **Vocal rhythm:** amount of rubato in the voice part.
- **Orchestral rhythm:** amount of rubato in the orchestral part.
- **Glissando:** degree to which voice(s) slide to and from notes.

- **Melisma:** number of pitches sung per syllable.
- **Tremolo:** amount of undulation on held notes.
- **Glottal effect:** amount of glottal activity present.
- **Vocal register:** whether singers are singing near the bottom, middle or top of their ranges.
- **Vocal width and tension:** degree to which voice(s) sound thin or rich.
- **Nasalization:** how nasal the singing sounds.
- **Raspy:** amount of raspiness in singing.
- **Accent:** strength of attack of sung tones.
- **Consonants:** precision of enunciation in singing.

There are a number of difficulties in implementing automatic extraction algorithms for some of these features. Some of these features, such as *nasalization,* require somewhat subjective judgements, and others require information that is not typically stored in symbolic formats, such as *glottal effect*. Nonetheless, some of these features can currently be extracted relatively easily from symbolic data, and future advances might make many of the remaining features accessible via audio/symbolic hybrid systems. Lomax did find a good correlation between these features and cultural patterns, and they intuitively seem as if they might perform well, so the work necessary to automatically extract these features may well be worth the effort. Many of these features might be of particular utility to automatic classification projects focussing on non-Western music.

One feature of particular interest is Lomax's *melodic shape* feature, which relates to the overall shape of individual melodies or phrases, something that is commonly referred to as *melodic contour* or *melodic arc*. This is an area in which a significant amount of research has been done, and it is worthy of special attention. Charles Adams, for example, found that examining melodic contour can allow one to differentiate successfully between different musics (Adams 1976). Adams based his analyses on only the initial note, highest note, lowest note and final note of melodies, a simplicity of approach that is intuitively appealing from the perspective of automatic feature extraction.

There are, unfortunately, some complications encountered in automatically applying Adams' approach to typical MIDI recordings. To begin with, automatically isolating the individual melodies of a piece can be difficult when dealing with music with multiple voices, as the melodies can involve notes contained in only one or in many voices. In the case of polyphonic music, one must deal with simultaneous melodies. One solution would be to implement a specialized melodic segmentation pre-processing system. Although some research has been done on such systems, none of have yet been successfully developed that could be applied to arbitrary types of music. Fortunately, it is possible to extract at least some features related to melodic contour using relatively simple assumptions and pre-processing. Section 4.5.6 specifies several features in the jSymbolic catalogue that were inspired by the work of Adams and others.

A number of writers have emphasized certain broad areas that they suggest would be particularly useful to concentrate on for the purpose of musical classification. Nettl (1990), for example, has proposed the following feature areas as having significance to a range of different cultures:

- Sound and singing style
- Form
- Polyphony (texture)
- Rhythm and tempo
- Melody and scale

Cumming (1999) has suggested a number of features in relation to motets in particular. With the understanding that "voices" can be adapted to mean voices or instruments, a number of these features appear to have a good deal of general applicability:

- Texture
- Number of voices
- Voice ranges

- Melodic behaviour (leaps, steps)
- Relative speed of voices
- Coordination of phrases between voices
- Use of rests
- Length
- Complexity
- Tone

Philip Tagg has proposed the following "checklist of parameters of musical expression" that can be used as a basis for designing features for both symbolic and audio extraction systems:

1. *Aspects of time: duration of analysis object and relation of this to any other simultaneous forms of communication; duration of sections within the analysis object; pulse, tempo, meter, periodicity; rhythmic texture motifs.*
2. *Melodic aspects: register; pitch range; (melodic) motifs; tonal vocabulary; contour; timbre.*
3. *Orchestration aspects: type and number of voices, instruments, parts; technical aspects of performance; timbre; phrasing; accentuation.*
4. *Aspects of tonality and texture: tonal centre and type of tonality (if any); harmonic idiom; harmonic rhythm; type of harmonic change; chordal alteration; relationship between voices, parts, instruments; compositional texture and method.*
5. *Dynamic aspects: levels of sound strength; accentuation; audibility of parts.*
6. *Acoustical aspects: characteristics of (re-)performance 'venue'; degree of reverberation; distance between sound source and listener; simultaneous 'extraneous' sound.*
7. *Electromusical and mechanical aspects: panning, filtering, compressing, phasing, distortion, delay, mixing, etc.; muting, pizzicato, tongue flutter, etc.* (Tagg 1982, 45-46).

Although this checklist was designed with the analysis of Western music in mind, many of the ideas embedded in it have a more general applicability. Another useful list of parameters has been suggested by David Cope (1991b), although this list also emphasizes parameters specific to Western art music.

Once one decides upon the general types of information that should be emphasized in features, it is then necessary to define more specifically the form that the features themselves will take. As a general principle, it is best to concentrate primarily on features that can be represented by simple numbers.

Continuous numerical values, as opposed to strings or discrete numbers, have the advantage that they generally allow a more precise representation of degree. For example, a numerical feature used to represent the key of a tonal piece could indicate not only its key, but also how strongly the piece can be said to belong to that key. This could be used to indirectly encapsulate information about the amount of chromaticism, for example, in a way that a discrete letter value cannot. Even in the case where discrete feature values are the most appropriate, integer representations are typically a better choice than string representations, as they can be most naturally and consistently processed by the majority of machine learning algorithms.

Based on informal experience gleaned from developing jSymbolic and other feature extractors, it was found that the easiest to use and, often, most effective features consist of either a single numerical value or a vector of numerical values. Larger feature arrays can also be useful at times as well, although not all machine learning algorithms can process them directly.

Vectors and arrays tend to be best suited to information that consists of a set of related values that have limited significance when considered individually, but can reveal meaningful patterns when considered together. For example, the average duration of melodic arcs in a piece would be a good example of a feature that is best represented as a single value, and the bin magnitudes of a histogram indicating the relative frequency of different melodic intervals might be a good feature vector. Although the individual bin frequencies could certainly each be represented as a separate feature, combining them into a vector highlights their particular interrelatedness. This division into single-value features and feature vectors is useful from the perspective of machine learning because it makes it possible to use classifier ensembles that capitalize on the particular relatedness of the components of each multi-dimensional feature, such as an ensemble constructed by training a separate classifier (e.g., a neural net) on each multi-dimensional feature and a single nearest neighbour classifier on all single-value features. This particular approach was used successfully in genre classification experiments (McKay 2004).

Simple statistical techniques can be useful in capturing overall characteristics of a piece, as well as how these characteristics change as the piece progresses. The calculation of metafeatures like the mean or variance of a feature across windows of a piece are good examples of this. With this in mind, it is important to point out that one would prefer to avoid feature noise due to non-representative local characteristics of a piece, while at the same time not ignoring characteristic local behaviour by smudging feature values over large analysis windows. For example, one would not want melodic ornamentations to obscure a feature that is measuring melodic structure, but might still want to measure the overall amount or type of ornamentation in a separate feature, as this itself could be characteristic of a class.

Histograms can serve as a particularly useful and convenient tool for dealing with the problem of wanting to encapsulate both local and overall behaviour, and statistical tools such as peak picking can be applied to histograms in order to derive further features from them. Examples of statistics often calculated from histograms include the number of strong peaks, the relative strengths of the highest peaks, the locations and harmonicity of peaks, the local spread around peaks and the relative contribution of bins not associated with peaks. Histograms can thus be used directly as feature vectors, or they can be used as intermediate representations for deriving other features. jSymbolic includes a variety of histogram-based features, as described in Section 4.5.

As a final point, it is desirable not only to have features that effectively partition recordings into different classes, but also to have features that can arguably be perceived by humans and are of musicological interest, at least when possible. Although this is by no means a requirement for systems with purely practical or commercial goals, it is certainly advantageous to research projects designed to provide insights on how humans process and consume music, and a good feature catalogue should have at least some features that can be used in such a fashion.

## 4.5 The jSymbolic feature catalogue

Sections 4.5.1 to 4.5.7 describe the features that make up the jSymbolic feature catalogue. These features were all designed in the context of the research described in Sections 4.3 and 4.4. Some of these features are very similar or identical to features that have been used in previous MIR research, others are formulated as features for the first time but are based on existing theoretical or musicological work, and still others are entirely original.

It was not possible to correctly implement all 153 of the features proposed in this section within the time constraints of this project. A total of 111 features were nonetheless implemented in the jSymbolic software, a much larger number of features than have been implemented in any other existing symbolic music classification system. The particular choice of which features to omit in the current implementation was based on a combination of the author's judgement of how useful each feature would be and of how time-consuming it would be to implement it, an important concern given the size and scope of the jMIR software. All of the features described in Sections 4.5.1 to 4.5.7 are implemented, except for T-11, T-14, T-16, T-17, T-18, T-19, R-16, R-26, R-27, R-28, R-29, P-26, M-16, M-20 and C-1 to C-28.

The cultural origins and training that inform any person's knowledge about music can lead them to give too much weight to some characteristics of music and too little to others. Correspondingly, the reader will notice a bias towards Western tonal music in examining the jSymbolic features, as one might expect given the background of jSymboloic's author and the research upon which many of the jSymbolic features are based. This particular emphasis is also partly due to the current dominance of Western genres in MIR research in general, and to the limitations of MIDI and other symbolic formats, which tend to be more oriented towards Western music. Despite this, efforts were certainly still made to include features that might not be obvious to one accustomed only to Western music whenever possible. The addition of more diversely representative features is certainly a priority for future research, and the jSymbolic feature catalogue is intended as a work in continual progress that can always be expanded and refined.

The reader will also notice that there is a small amount of redundancy in the jSymbolic feature catalogue, in the sense that one feature occasionally emphasizes a particular aspect of another feature. This was done in order to ensure that features are available that provide an overview of certain musical aspects as well as features that provide more focused information that might be particularly salient for some types of classification. The jSymbolic feature catalogue is intended as a catalogue or palette from which different feature subsets can be selected for different research projects, not as an indivisible feature set that must always be extracted in its entirety. Some features may be appropriate for certain tasks but not others. The feature catalogue was designed to give musical experts and/or feature selection algorithms as wide a range of features as possible from which to choose for any given research application. The redundancy in the features was consequently purposely included to facilitate this choice by making it possible to meet subtle musical classification needs.

Efforts were made when possible to utilize features that could be easily adapted to a variety of symbolic music representations, not just MIDI. Of course, MIDI is the format actually processed by jSymbolic, so the features are described in the following sub-sections using MIDI terminology. This allows the feature definitions to be more precise, but it does not mean that many of the features could not still easily be adapted to other symbolic formats. For example, one could substitute the MIDI term *velocity* for some other measure of loudness. The reader may wish to consult Section 4.2.2 if he or she is unfamiliar with MIDI terminology.

Unfortunately, from the perspective of standardized feature extraction, there are many different ways of encoding MIDI data. MIDI recordings can be produced by writing out music by hand in score writing software like Finale, or by performing actual real-time recordings using MIDI instruments, for example, and each of these approaches can result in significant differences in the encoded music. Rhythmic quantization is one common example of a common resulting difference. Differences in MIDI encoding styles or even in the particular choice of sequencer or score writing software can also result in differences in the encoded music. Such differences can result in feature noise that can be potentially problematic for certain types of classification. A composer classification system, for example, should be insensitive to the encoding style. Care was therefore taken, when possible, to use features that had as little sensitivity as possible to such differences in encoding style.

One will notice that there are a few "magic numbers" in the descriptions of some of the features. The values of these constants are based on intuition and informal experimental experience, and magic numbers are only used when the nature of a particular feature necessitates it, and are avoided whenever possible.

A number of intermediate representations of the MIDI recordings, including histogram-based representations, were constructed in order to derive some of the features described below. The most interesting of these representations are explicitly mentioned in the following sub-sections, particularly in cases where the intermediate representations were themselves used as features as well.

### 4.5.1 Features based on instrumentation

Although there is a significant amount of literature on instrumentation with respect to composing and arranging, very few music analytical systems take instrumentation into consideration. This is a shame, as information on instrumentation can in fact be very helpful in discriminating between certain types of classes. One study, for example, indicated that features based on instrumentation were the most effective type of features when classifying symbolic music by genre (McKay and Fujinaga 2005a).

The jSymbolic software capitalizes on the fact that the General MIDI (level 1) specification allows MIDI files to include 128 different pitched-instrument patches, and the MIDI Percussion Key Map permits a further 47 percussion instruments. Although these MID instruments are certainly much fewer in number than the full range of extant instruments, particularly with respect to non-Western musics, they are nonetheless diverse enough for a reasonable variety of musical types.

MIDI instrumentation notation can be somewhat sensitive to encoding inconsistencies between different MIDI authors in some cases. In a few fortunately rare cases, authors fail to specify patch numbers, with the result that all notes are played using a piano patch by default. Another problem is the inconsistency in the choice of patches that are used to represent sung lines, since there is no good General MIDI patch for solo vocal lines.

Despite these occasional problems, however, features based on instrumentation can still be highly characteristic of various musical categories, and the complementary use of other types of features can help to counterbalance inconsistencies in individual authors' choices of patches.

The jSymbolic feature catalogue includes the following instrumentation-related features:

- **I-1 Pitched Instruments Present:** A feature vector with one entry for each of the 128 General MIDI Instruments. Each value is set to 1 if at least one note is played using the corresponding patch, or to 0 if that patch is never used.
- **I-2 Unpitched Instruments Present:** A feature vector with one entry for each of the 47 MIDI Percussion Key Map instruments. Each value is set to 1 if at least one note is played using the corresponding patch, or to 0 if that patch is never used.
- **I-3 Note Prevalence of Pitched Instruments:** A feature vector with one entry for each of the 128 General MIDI Instruments. Each value is set to the number of Note Ons played with the corresponding MIDI patch, divided by the total number of Note Ons in the piece.
- **I-4 Note Prevalence of Unpitched Instruments:** A feature vector with one entry for each of the 47 MIDI Percussion Key Map instruments. Each value is set to the number of Note Ons played with the corresponding MIDI patch, divided by the total number of Note Ons in the piece.
- **I-5 Time Prevalence of Pitched Instruments:** A feature vector with one entry for each of the 128 General MIDI Instruments. Each value is set to the total time in seconds in a piece during which at least one note is being sounded with the corresponding MIDI patch, divided by the total length of the piece in seconds.
- **I-6 Variability of Note Prevalence of Pitched Instruments:** Standard deviation of the fraction of total notes in a piece played by each General MIDI instrument that is used to play at least one note.

- **I-7 Variability of Note Prevalence of Unpitched Instruments:** Standard deviation of the fraction of total notes played by each MIDI Percussion Key Map instrument that is used to play at least one note.
- **I-8 Number of Pitched Instruments:** Total number of General MIDI patches that are used to play at least one note.
- **I-9 Number of Unpitched Instruments:** Total number of MIDI Percussion Key Map patches that are used to play at least one note.
- **I-10 Percussion Prevalence:** Total number of Note Ons belonging to percussion patches divided by total number of Note Ons in the recording.
- **I-11 String Keyboard Fraction:** Fraction of Note Ons belonging to string keyboard patches (General MIDI patches 1 to 8).
- **I-12 Acoustic Guitar Fraction:** Fraction of Note Ons belonging to acoustic guitar patches (General MIDI patches 25 and 26).
- **I-13 Electric Guitar Fraction:** Fraction of Note Ons belonging to electric guitar patches (General MIDI patches 27 to 32).
- **I-14 Violin Fraction:** Fraction of Note Ons belonging to violin patches (General MIDI patches 41 or 111).
- **I-15 Saxophone Fraction:** Fraction of Note Ons belonging to saxophone patches (General MIDI patches 65 to 68).
- **I-16 Brass Fraction:** Fraction of Note Ons belonging to brass patches, including saxophones (General MIDI patches 57 to 68).
- **I-17 Woodwinds Fraction:** Fraction of Note Ons belonging to woodwind patches (General MIDI patches 69 to 76).
- **I-18 Orchestral Strings Fraction:** Fraction of Note Ons belonging to orchestral string patches (General MIDI patches 41 to 47).
- **I-19 String Ensemble Fraction:** Fraction of Note Ons belonging to orchestral string ensemble patches (General MIDI patches 49 to 52).

- **I-20 Electric Instrument Fraction:** Fraction of Note Ons belonging to electric non-“synth” patches (General MIDI patches 5, 6, 17, 19, 27 to 32, 34 to 40).

### 4.5.2 Features based on musical texture

Although the term *texture* is associated with several different musical meanings, the features falling into this category of the jSymbolic catalogue relate specifically to the number of independent voices in a piece and how these voices relate to one another.

jSymbolic takes advantage of the fact that MIDI notes can be assigned to different channels, thus making it possible to segregate the notes belonging to different voices. Although it might seem natural to use MIDI tracks instead of channels to distinguish between voices, since only a maximum of sixteen MIDI channels are available, this is an ineffective approach in practice. Using MIDI tracks would mean that it would be impossible to extract texture-based features from all Type 0 MIDI files, since this format only allow permits a single track to be represented. Even in the case of Type 1 files, which do allow tracks to be specified, it is still not unusual to find all MIDI data saved on a single track in practice. Almost all MIDI files do use different channels for different voices, however, and it is possible to take advantage of Program Change messages to multiplex multiple voices onto a single channel in order to avoid being restricted to only sixteen voices. It was therefore decided to use MIDI channels in order to distinguish between voices rather than MIDI tracks.

This approach is not perfect, as it is possible to use a single channel to hold multiple voices even without regular program change messages. A piano could be used to play a four-voice chorale, for example, with all notes occurring on one MIDI channel. This problem is unavoidable, unfortunately, unless one implements a sophisticated voice partitioning pre-processing module to automatically segregate voices prior to feature extraction, something that is beyond the current scope of this work. Fortunately, this problem does not occur all that often in MIDI files.

The jSymbolic feature catalogue includes the following texture-related features:

- **T-1 Maximum Number of Independent Voices:** Maximum number of different channels in which notes are sounded simultaneously.

- **T-2 Average Number of Independent Voices:** Average number of different channels in which notes are sounded simultaneously. Rests are not included in this calculation.
- **T-3 Variability of Number of Independent Voices:** Standard deviation of number of different channels in which notes are sounded simultaneously. Rests are not included in this calculation.
- **T-4 Voice Equality – Number of Notes:** Standard deviation of the total number of Note Ons in each channel that contains at least one note.
- **T-5 Voice Equality – Note Duration:** Standard deviation of the total duration of notes in each channel that contains at least one note.
- **T-6 Voice Equality – Dynamics:** Standard deviation of the average volume of notes in each channel that contains at least one note.
- **T-7 Voice Equality – Melodic Leaps:** Standard deviation of the average melodic leap distance for each channel that contains at least one note.
- **T-8 Voice Equality – Range:** Standard deviation of the differences between the highest and lowest pitches in each channel that contains at least one note.
- **T-9 Importance of Loudest Voice:** Difference between the average loudness of the loudest channel and the average loudness of the other channels that contain at least one note, divided by 64 (128 / 2).
- **T-10 Relative Range of Loudest Voice:** Difference between the highest note and the lowest note played in the channel with the highest average loudness divided by the difference between the highest note and the lowest note in the piece overall.
- **T-11 Relative Range Isolation of Loudest Voice:** Number of notes in the channel with the highest average loudness that fall outside the range of any other channel divided by the total number of notes in the channel with the highest average loudness.

- **T-12 Range of Highest Line:** Difference between the highest note and the lowest note played in the channel with the highest average pitch divided by the difference between the highest note and the lowest note in the piece overall.
- **T-13 Relative Note Density of Highest Line:** Number of Note Ons in the channel with the highest average pitch divided by the average number of Note Ons in all channels that contain at least one note.
- **T-14 Relative Note Durations of Lowest Line:** Average duration of notes (in seconds) in the channel with the lowest average pitch divided by the average duration of notes in all channels that contain at least one note.
- **T-15 Melodic Intervals in Lowest Line:** Average melodic interval in semitones of the line with the lowest average pitch divided by the average melodic interval of all lines that contain at least two notes.
- **T-16 Simultaneity:** Average number of notes sounding simultaneously.
- **T-17 Variability of Simultaneity:** Standard deviation of the number of notes sounding simultaneously.
- **T-18 Voice Overlap:** Number of notes played within the range of another voice divided by total number of notes in the piece overall.
- **T-19 Parallel Motion:** Fraction of all notes that move together in the same direction within 10% of the duration of the shorter note.
- **T-20 Voice Separation:** Average separation in semi-tones between the average pitches of consecutive channels (after sorting based on average pitch) that contain at least one note, divided by 6.

### 4.5.3 Features based on rhythm

Several scholars have expressed the view that rhythm plays a very important or even dominant role in many types of music. Richard Middleton (2000), for example, stresses the importance of rhythm in characterising music in a discussion of ways to approach creating a widely applicable method for music analysis. It is unfortunate that the majority of traditional analytical frameworks, with a few exceptions like the work of Cooper and

Meyer (1960), tend to give rhythm less attention than it deserves. Rhythmic features are often used successfully in audio analysis systems, in contrast, and they are correspondingly included in the jSymbolic feature catalogue in order to address the traditionally limited role of rhythm in traditional symbolic analysis.

The two elementary pieces of information from which most rhythmic features can be calculated are the times at which notes begin (called *note onsets*) relative to one another, and the durations of notes. Note onsets can be extracted relatively reliably from audio data, at least in cases where note density is not too high, but durations are more difficult to extract reliably. In the case of symbolic data, however, both note onsets and durations are easily and precisely available. As one might expect, several of the rhythmic features that are based on note onsets in the jSymbolic catalogue are very similar to features that are often used in audio feature extraction systems. Duration-based features, in contrast, are very rarely currently used by audio feature extraction software, but can easily be extracted from symbolic data, and are thus included in the jSymbolic feature catalogue in order to allow their utility to be empirically evaluated.

Before proceeding to discuss the details of the jSymbolic rhythmic feature catalogue, it is important to emphasize a detail of how MIDI encodes rhythmic information that must be considered when designing rhythmic features, whether for jSymbolic or for some other MIDI software. MIDI timings are affected by both the number of MIDI ticks that go by between Note On events and by tempo change meta-events that control the rate at which MIDI ticks go by. Tempo change meta-events must therefore be monitored by the feature extraction software, something which jSymbolic does of course do.

One disadvantage of symbolic data is that some important rhythmic features are related to performance characteristics that are not always available in symbolic data, or available in only a very coarse sense. For example, musical scores may indicate that a piece should be played rubato or with a swing rhythm. There is a great deal of variety in the ways in which these rhythmic performance styles can be implemented, however, something that can be of essential importance for tasks such as performer identification. Although formats such as MIDI certainly can represent precise note onset timings, and many recorded MIDI performances do indeed take advantage of this, MIDI files that are generated using score writing software are often strictly quantized, which means that performance timing information is not always consistently available with the precision that would ideally be preferred.

Nonetheless, even quantized rhythms can still result in very useful feature values. One of the nice things about MIDI is that it allows one to access timing information in terms of raw time of note onsets as well as in terms of rhythmic note values (i.e., half notes, quarter notes, etc.), thus providing both low-level and high-level rhythmic information. This information, along with time signature and tempo change meta-events, can potentially provide features with a high discriminating power.

Of course, as mentioned above, MIDI rhythmic information is somewhat sensitive to MIDI encoding style. This inconsistency is precisely the reason why the jSymbolic feature catalogue places a particular emphasis on rhythmic features derived from *beat histograms,* as described below. This approach helps to statistically smooth over some inconsistencies due to encoding style.

Beat histograms are an approach that was first applied to MIR research by Brown (1993), and was later publicized and used for automatic genre classification by Tzanetakis and his colleagues in a number of papers (Tzanetakis, Essl & Cook 2001; Tzanetakis & Cook 2002; Tzanetakis 2002). A slightly modified version of Tzanetakis' histogram is used by jSymbolic to derive a number of rhythmic features. Although Section 3.2.8 explains how beat histograms are calculated by jAudio, the concept is briefly described again below from the specific perspective of symbolic MIDI data.

It is necessary to have some understanding of how *autocorrelation* works in order to understand how beat histograms are constructed. Autocorrelation essentially involves comparing a signal with versions of itself delayed by successive intervals. This technique is often used to find repeating patterns in signals of any kind, as it yields the relative strengths of different periodicities within a signal. In terms of musical data, autocorrelation allows one to find the relative strengths of different rhythmic pulses.

jSymbolic constructs its rhythmic histograms by processing sequences of MIDI Note On events, with MIDI ticks comprising the time scale. The autocorrelation function is:

$$autocorrelation[lag] = \frac{1}{N}\sum_{n=1}^{N} Y[n]Y[n-lag] \qquad (4.1)$$

where *Y* is the sequence of MIDI data, *n* is the input sequence index (in MIDI ticks), *N* is the total number of MIDI ticks in the sequence and *lag* is the delay in MIDI ticks ($0 \leq lag$

< *N*). The value of *Y[n]* is calculated to be proportional to the velocity of Note Ons in order to ensure that beats are weighted based on the strength with which notes are played. This autocorrelation function is applied iteratively to each MIDI sequence, once for each value of *lag* within the domain $0 \leq lag < N$. The values of *lag* correspond to both rhythmic periodicities as well as, after processing, the bin labels of the beat histogram, and the autocorrelation values provide the magnitude value for each bin.

Once the histogram is populated using all permissible values of *lag* for a given MIDI sequence, jSymbolic then downsamples and transforms it so that each bin corresponds to a rhythmic periodicity with units of beats per minute. The histogram is then normalized so that different MIDI sequences can be compared. The end result is a histogram whose bins correspond to rhythmic pulses with units of beats per minute and whose bin magnitudes indicate the relative strength of each such rhythmic pulse. In effect, a beat histogram portrays the relative strength of different beats and sub-beats within a piece.

Consider, for example, the beat histograms extracted from MIDI representations of *I Wanna Be Sedated,* by the punk band The Ramones, and *'Round Midnight,* by the jazz performer and composer Thelonious Monk, as shown in Figures 4.1 and 4.2 respectively. It is clear that *I Wanna Be Sedated* has significant rhythmic looseness, as demonstrated by the spread around each peak, each of which represents a strong beat periodicity. *I Wanna Be Sedated* also has several clear strong beats, including ones centred at 55, 66, 82, 111 (the actual tempo of the song) and 164 beats per minute, the latter two of which are harmonics of 55 and 82 beats per minute. *'Round Midnight,* in contrast, has one very strong beat at 76 beats per minute, the actual tempo of the piece, and a wide range of much lower-level beat strengths. This indicates that, as might be expected, *'Round Midnight* is more rhythmically complex and is also performed more tightly.

This type of information can be very representative of different musical classes, such as genre. Techno, for example, often has very clearly defined beats, without any surrounding spread, because the beats are precisely generated electronically. Much modern Classical music, to provide a contrasting example, often has much less clearly defined beats.

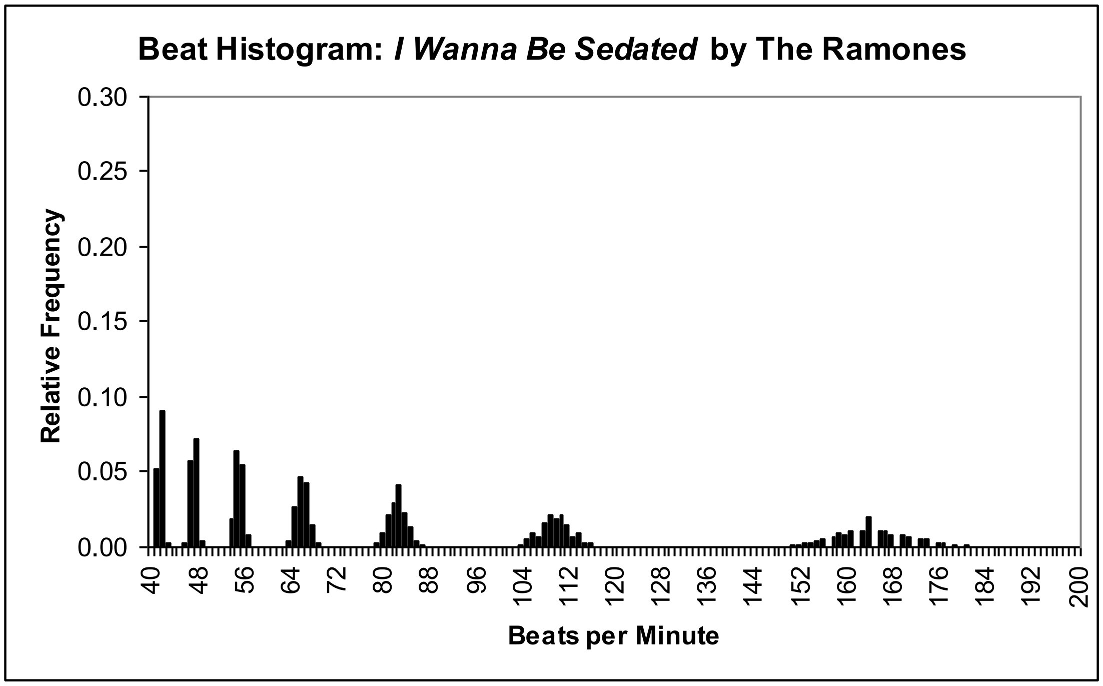


**Figure 4.1:** Beat histogram for *I Wanna Be Sedated* by The Ramones.

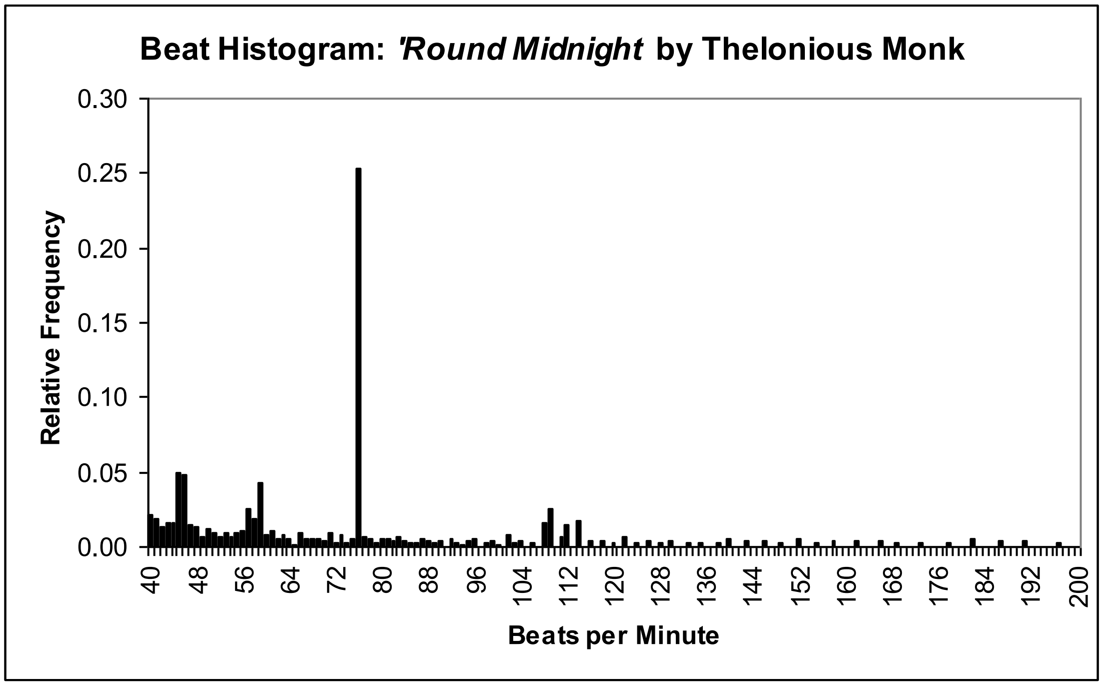


**Figure 4.2:** Beat histogram for *'Round Midnight* by Thelonious Monk.

Part of the challenge of histogram-related features is that one must find a way to represent the information embedded in them as useful features. Although beat histograms certainly can be used directly as feature vectors, and they sometimes are in jSymbolic, experience has shown that machine learning algorithms can sometimes have trouble learning to extract useful information from them in this raw form if they are too large. Beat and other feature histograms are, however, very useful in providing an intermediate data structure from which other features can be extracted. Experience has shown informally that the two highest peaks of beat histograms tend to be of particular importance in extracting such information, as they are the most likely to represent the main beat of the music or one of its multiples or factors.

The jSymbolic feature catalogue includes the following rhythm-related features:

- **R-1 Strongest Rhythmic Pulse:** Bin label of the bin of the beat histogram with the highest magnitude.
- **R-2 Second Strongest Rhythmic Pulse:** Bin label of the beat histogram peak with the second highest magnitude.
- **R-3 Harmonicity of Two Strongest Rhythmic Pulses:** Bin label of the higher (in terms of bin label) of the two beat histogram peaks with the highest magnitude divided by the bin label of the lower.
- **R-4 Strength of Strongest Rhythmic Pulse:** Magnitude of the beat histogram bin with the highest magnitude.
- **R-5 Strength of Second Strongest Rhythmic Pulse:** Magnitude of the beat histogram peak with the second highest magnitude.
- **R-6 Strength Ratio of Two Strongest Rhythmic Pulses:** Magnitude of the higher (in terms of magnitude) of the two beat histogram bins corresponding to the peaks with the highest magnitude divided by the magnitude of the lower.
- **R-7 Combined Strength of Two Strongest Rhythmic Pulses:** The sum of the magnitudes of the two beat histogram peaks with the highest magnitudes.

- **R-8 Number of Strong Pulses:** Number of beat histogram peaks with normalized magnitudes over 0.1.
- **R-9 Number of Moderate Pulses:** Number of beat histogram peaks with normalized magnitudes over 0.01.
- **R-10 Number of Relatively Strong Pulses:** Number of beat histogram peaks with magnitudes at least 30% as high as the magnitude of the peak with the highest magnitude.
- **R-11 Rhythmic Looseness:** Average width of beat histogram peaks (in beats per minute). Width is measured for all peaks with magnitudes at least 30% as high as the highest peak, and is defined by the distance between the points on the peak in question that have magnitudes closest to 30% of the height of the peak.
- **R-12 Polyrhythms:** Number of beat histogram peaks with magnitudes at least 30% of the highest magnitude whose bin labels are not integer multiples or factors (using only multipliers of 1, 2, 3, 4, 6 and 8, and with an accepted error of +/- 3 bins) of the bin label of the peak with the highest magnitude. This number is then divided by the total number of bins with frequencies over 30% of the highest magnitude.
- **R-13 Rhythmic Variability:** Standard deviation of the beat histogram bin magnitudes (excepting the first 40 empty ones).
- **R-14 Beat Histogram:** A feature vector consisting of the bin magnitudes of the beat histogram described above.
- **R-15 Note Density:** Average number of notes per second.
- **R-16 Note Density Variability:** The recording is broken into windows with 5 second durations. The note density is then calculated for each window, and the standard deviation of these windows is then calculated.
- **R-17 Average Note Duration:** Average duration of notes in seconds.

- **R-18 Variability of Note Duration:** Standard deviation of note durations in seconds.
- **R-19 Maximum Note Duration:** Duration of the longest note (in seconds).
- **R-20 Minimum Note Duration:** Duration of the shortest note (in seconds).
- **R-21 Staccato Incidence:** Number of notes with durations of less than 0.1 seconds divided by the total number of notes in the recording.
- **R-22 Average Time Between Attacks:** Average time in seconds between Note On events (regardless of channel).
- **R-23 Variability of Time Between Attacks:** Standard deviation of the times, in seconds, between Note On events (regardless of channel).
- **R-24 Average Time Between Attacks For Each Voice:** Average of the individual channel averages of times in seconds between Note On events. Only channels that contain at least one note are included in the average.
- **R-25 Average Variability of Time Between Attacks For Each Voice:** Average standard deviation, in seconds, of time between Note On events on individual channels that contain at least one note.
- **R-26 Incidence of Complete Rests:** Total amount of time in seconds in which no notes are sounding on any channel divided by the total length of the recording.
- **R-27 Maximum Complete Rest Duration:** Maximum amount of time in seconds in which no notes are sounding on any channel.
- **R-28 Average Rest Duration Per Voice:** Average, in seconds, of the average amounts of time in each channel in which no note is sounding (counting only channels with at least one note), divided by the total duration of the recording.
- **R-29 Average Variability of Rest Durations across Voices:** Standard deviation, in seconds, of the average amounts of time in each channel in which no note is sounding (counting only channels with at least one note).
- **R-30 Initial Tempo:** Tempo in beats per minute at the start of a recording.

- **R-31 Initial Time Signature:** A feature vector consisting of two values. The first is the numerator of the first occurring time signature and the second is the denominator of the first occurring time signature. Both are set to 0 if no time signature is present.

- **R-32 Compound Or Simple Meter:** Set to 1 if the initial meter is compound (numerator of time signature is greater than or equal to 6 and is evenly divisible by 3) and to 0 if it is simple (if the above condition is not fulfilled).

- **R-33 Triple Meter:** Set to 1 if numerator of initial time signature is 3, set to 0 otherwise.

- **R-34 Quintuple Meter:** Set to 1 if numerator of initial time signature is 5, set to 0 otherwise.

- **R-35 Changes of Meter:** Set to 1 if the time signature is changed one or more times during the recording.

### 4.5.4 Features based on dynamics

The ways in which musical dynamics are used in a piece can also be characteristic of different types of musical classes. Once again, however, this information is only rarely used in traditional analytical systems, and is generally notated only very coarsely in musical scores. Fortunately, MIDI velocity values make it possible to annotate dynamics much more precisely, even though MIDI encodings generated by score editing software admittedly generally fail to take full advantage of this.

One important point to consider with respect to MIDI dynamics is that, while MIDI velocities are generally used to indicate the strength with which notes are sounded, this is not the only way in which loudness is controlled. MIDI channel volume can also be changed independently. jSymbolic takes this into account by using the following formula to find *loudness* values used to calculate the features described in this sub-section:

*loudness = note velocity x (channel volume / 127)* *(4.2)*

It should also be noted that all of the jSymbolic features related to dynamics use relative measures of loudness rather than absolute measures because the default volume and velocity values set by sequencers can vary.

The jSymbolic feature catalogue includes the following features related to dynamics:

- **D-1 Overall Dynamic Range:** The maximum loudness value minus the minimum loudness value.
- **D-2 Variation of Dynamics:** Standard deviation of loudness levels of all notes.
- **D-3 Variation of Dynamics in Each Voice:** The average of the standard deviations of loudness levels within each channel that contains at least one note.
- **D-4 Average Note To Note Dynamics Change:** Average change of loudness from one note to the next note in the same channel.

### 4.5.5 Features based on overall pitch statistics

The majority of traditional analytical systems place a particular emphasis on information related to pitch, and this type of information certainly has value with respect to symbolic features as well. The pitch-related features in the jSymbolic catalogue are divided into three groups: Section 4.5.6 deals with melody and features based on sequences of pitches in general, Section 4.5.7 deals with chords and features based on vertical intervals in general, and this section deals with overall statistics on the pitches used in pieces as a whole. Unlike the features in Sections 4.5.6 and 4.5.7, the features in this section do not take into account the temporal locations of notes.

Just as beat histograms are useful for calculating a variety of rhythmic features, there are several histograms which can be used to calculate features related to pitch statistics. The jSymbolic feature catalogue uses slightly modified versions of the three pitch histograms implemented by Tzanetakis and his colleagues (Tzanetakis and Cook 2002; Tzanetakis, Ermolinskyi and Cook 2002; Tzanetakis 2002).

The first type of histogram is a *basic pitch histogram.* It consists of 128 bins, one for each MIDI pitch. The magnitude of each bin is first set to the number of Note On messages in the piece with the corresponding pitch, and the histogram is normalized after all Note On messages have been accounted for. This type of histogram gives particular insights into the range and variety of pitches used in a piece.

To provide practical examples, Figure 4.3 shows the basic pitch histogram for a Duke Ellington jazz piece and Figure 4.4 shows the histogram for a Dr. Dre rap song. A number of genre-typical differences are immediately apparent from even a rough visual comparison of these two histograms, such as the fact that the rap song uses far fewer pitches than the jazz piece, for example.

The second type of histogram is called a *pitch class histogram*. It has one bin for each of the twelve pitch classes, which means that it is essentially a version of the basic pitch histogram where octaves are collapsed for each of the pitch classes. The magnitude of each bin is set to the number of Note On messages with a MIDI pitch that can be wrapped to this pitch class, with enharmonic equivalents assigned to the same pitch class number. The histogram is normalized, and the bins are translated so that the first bin corresponds to the pitch class with the highest magnitude, with the successive bins ordered chromatically in semitone increments. This type of histogram provides insights into areas such as the types of scales used and the amount of transposition that is present, for example.

The third type of histogram is called a *folded fifths pitch histogram,* and is derived directly from the pitch class histogram. This histogram is calculated by reordering the bins of the original unordered pitch class histogram such that adjacent bins are separated by perfect fifths rather than semitones. This is done using the following equation

$$\beta = (7\alpha)\mathrm{mod}(12) \qquad (4.3)$$

where $\beta$ is the folded fifths pitch histogram bin and $\alpha$ is the corresponding pitch class histogram bin. The number seven is used because this is the number of semitones in a perfect fifth, and the number twelve is used because there are twelve pitch classes in total. This histogram is useful for measuring dominant tonic relationships and for looking at types of transpositions.

The utility of the folded fifths pitch histogram can be seen by comparing Figure 4.5, which shows the folded fifths pitch histogram for a Baroque Vivaldi concerto, and Figure 4.6, which shows the folded fifths pitch histogram for an atonal Schoenberg piano miniature. The Vivaldi piece never or rarely uses five of the twelve pitch classes, and the pitch classes that are used are clustered around one section of the circle of fifths. These are characteristics that one would typically expect of basic tonal music without many tonally distant modulations or significant use of chromaticism. In contrast, all of the pitch classes are used to a significant degree in the Schoenberg piece, and the most frequently used pitch classes are not clustered together on the circle of fifths, both of which are characteristics that one would expect of such an atonal piece.

All three of these histogram types are included directly as features in the jSymbolic feature catalogue, and are also used to calculate a number of other features.

It should be mentioned that all notes occurring on MIDI channel ten are ignored for all of the features described in this section. This is because the "pitch" values on channel ten correspond to (mostly unpitched) percussion patches, not to pitches.

Some of the features in this section are based on MIDI Pitch Bends. Although the use of Pitch Bends is somewhat variable from MIDI encoder to MIDI encoder, and therefore not entirely dependant on the music itself, features relating to Pitch Bends can nonetheless have a high discriminating power, so they are included here. Efforts were made to use features with as limited a sensitivity to non-musical factors as possible.

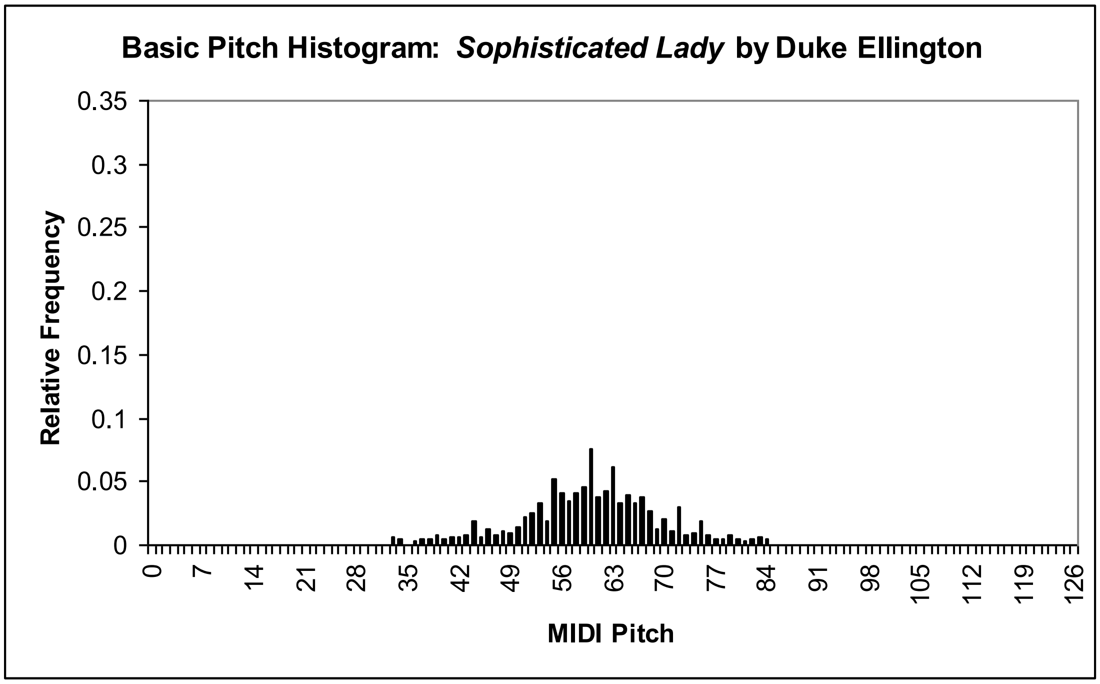


**Figure 4.3:** Basic pitch histogram for *Sophisticated Lady* by Duke Ellington.

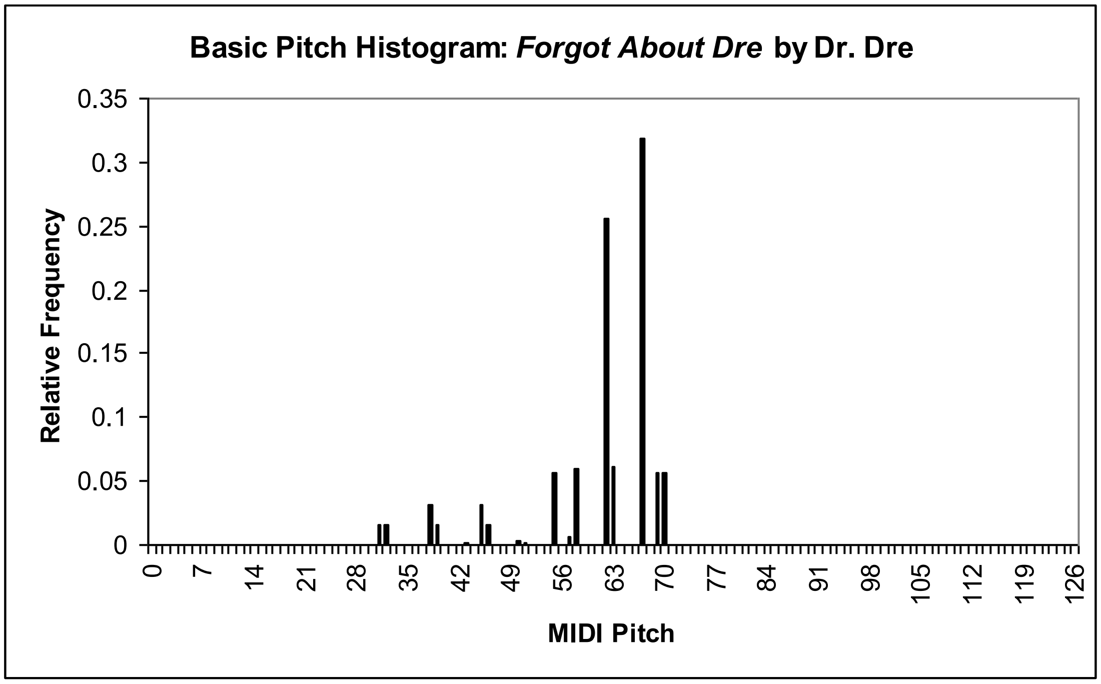


**Figure 4.4:** Basic pitch histogram for *Forgot About Dre* by Dr. Dre.

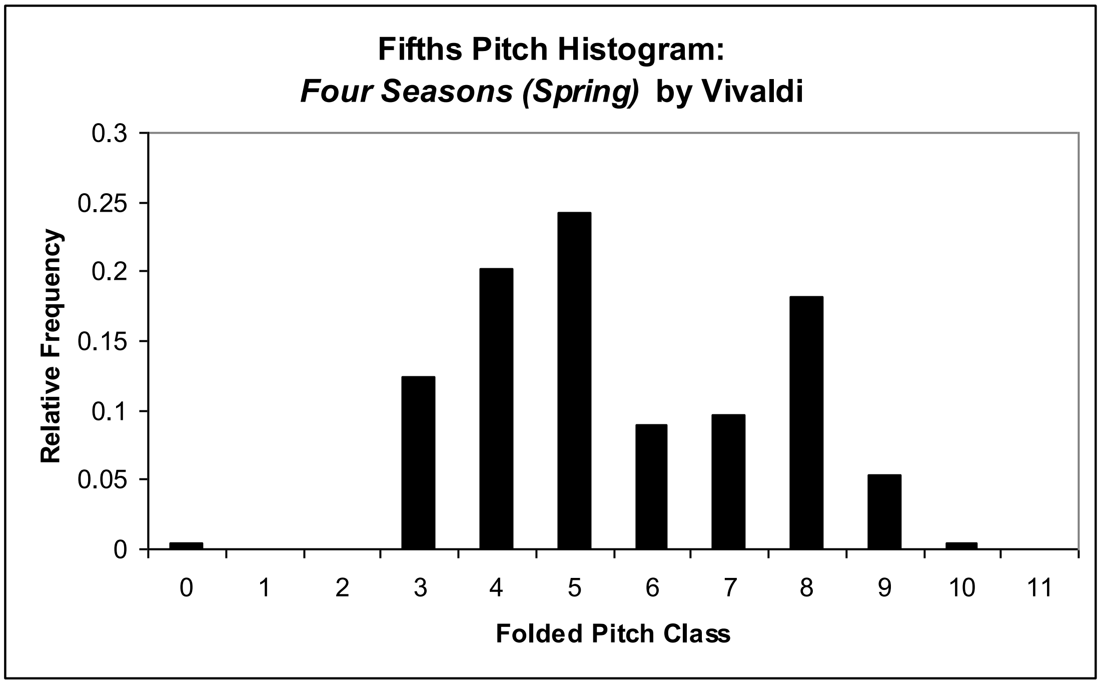


**Figure 4.5:** Folded fifths pitch histogram for *The Four Seasons (Spring)* by Vivaldi.

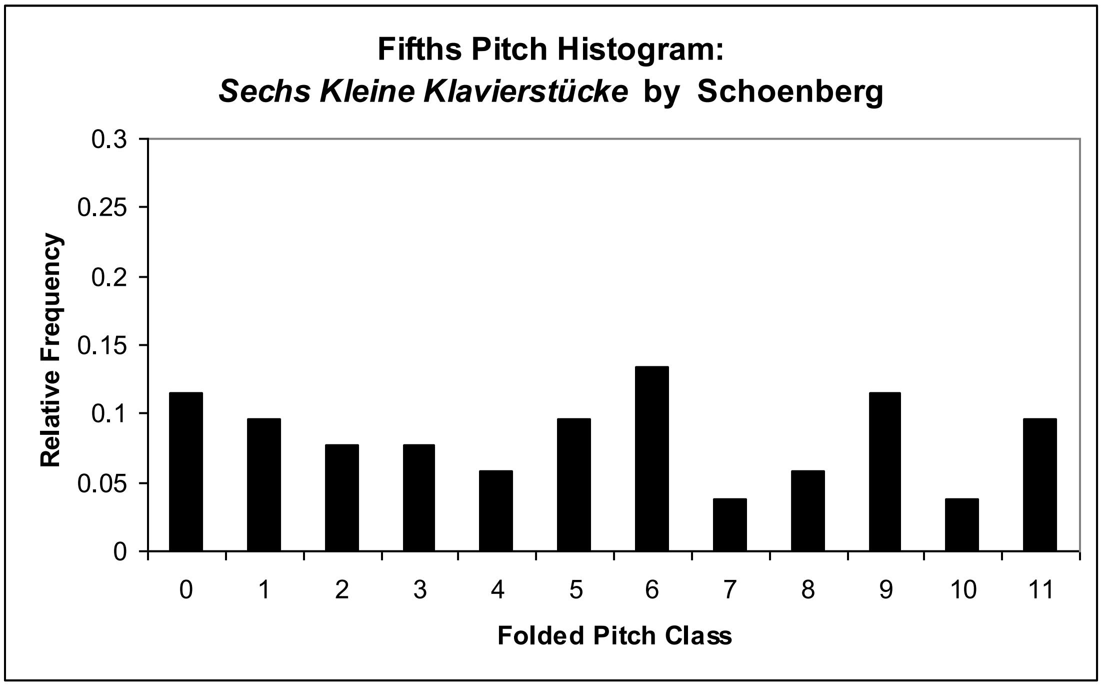


**Figure 4.6:** Folded fifths pitch histogram for a piano miniature from *Sechs Kleine Klavierstücke* by Schoenberg.

The jSymbolic feature catalogue includes the following features related to overall pitch statistics:

- **P-1 Most Common Pitch Prevalence:** Fraction of Note Ons corresponding to the most common pitch.
- **P-2 Most Common Pitch Class Prevalence:** Fraction of Note Ons corresponding to the most common pitch class.
- **P-3 Relative Strength of Top Pitches:** The magnitude of the second most common pitch divided by the magnitude of the most common pitch.
- **P-4 Relative Strength of Top Pitch Classes:** The magnitude of the second most common pitch class divided by the magnitude of the most common pitch class.
- **P-5 Interval Between Strongest Pitches:** Absolute value of the difference in semitones between the pitches of the two most common pitches.
- **P-6 Interval Between Strongest Pitch Classes:** Absolute value of the difference in semitones between the pitches of the two most common pitch classes.

- **P-7 Number of Common Pitches:** Number of pitches that account individually for at least 9% of all notes.
- **P-8 Pitch Variety:** Number of pitches used at least once.
- **P-9 Pitch Class Variety:** Number of pitch classes used at least once.
- **P-10 Range:** Difference in semitones between the highest and lowest pitches.
- **P-11 Most Common Pitch:** MIDI pitch value of the most common pitch divided by the number of possible pitches.
- **P-12 Primary Register:** Average MIDI pitch.
- **P-13 Importance of Bass Register:** Fraction of Note Ons between MIDI pitches 0 and 54.
- **P-14 Importance of Middle Register:** Fraction of Note Ons between MIDI pitches 55 and 72.
- **P-15 Importance of High Register:** Fraction of Note Ons between MIDI pitches 73 and 127.
- **P-16 Most Common Pitch Class:** Bin label on the pitch class histogram of the most common pitch class.
- **P-17 Dominant Spread:** Largest number of consecutive pitch classes separated by perfect 5ths that accounted for at least 9% each of the notes.
- **P-18 Strong Tonal Centres:** Number of peaks in the fifths pitch histogram that each account for at least 9% of all notes.
- **P-19 Basic Pitch Histogram:** A feature vector consisting of the bin magnitudes of the basic pitch histogram described above.
- **P-20 Pitch Class Distribution:** A feature vector consisting of the bin magnitudes of the pitch class histogram described above.
- **P-21 Fifths Pitch Histogram:** A feature vector consisting of the bin magnitudes of the fifths pitch histogram described above.

- **P-22 Quality:** Set to 0 if the key signature indicates that a recording is major, set to 1 if it indicates that it is minor and set to 0 if the key signature is unknown.
- **P-23 Glissando Prevalence:** Number of Note Ons that have at least one MIDI Pitch Bend associated with them divided by the total number of pitched Note Ons.
- **P-24 Average Range of Glissandos:** Average range of MIDI Pitch Bends, where "range" is defined as the greatest value of the absolute difference between 64 and the second data byte of all MIDI Pitch Bend messages falling between the Note On and Note Off messages of any note.
- **P-25 Vibrato Prevalence:** Number of notes for which MIDI Pitch Bend messages change direction at least twice divided by total number of notes that have Pitch Bend messages associated with them.
- **P-26 Prevalence of Micro-Tones:** Number of Note Ons that are preceded by isolated MIDI Pitch Bend messages as a fraction of the total number of Note Ons.

### 4.5.6 Features based on melody and melodic intervals

Although features based on overall pitch statistics are often meaningful and useful, they do not reflect information relating to the order in which pitches occur. Melody is a very important part of how many humans hear and think about music, so features based on such sequential information are needed to complement features based on overall pitch statistics. Fortunately, ample theoretical work has been done that can be taken advantage of when designing melodic features, ranging from compositional resources like manuals on writing Baroque counterpoint, to more analytically formulated ideas like melodic contour. Section 4.4 highlights several relevant resources.

Unfortunately, the tasks of detecting and partitioning musical phrases and melodies, and of determining which notes belong to which phrases, are not trivial. Although expert humans can perform such tasks relatively easily, automatic systems for performing them have still achieved only limited general success, particularly in cases where the notes in a phrases are shared across voices. So, although a phrase detection pre-processing system would make many potentially useful melodic features accessible, such a system is not currently available.

What one can do fairly easily, however, is collect basic statistics about melodic intervals and melodic motion. Although such statistics may be relatively rudimentary compared to expert melodic analyses, they can still potentially be very effective in performing classifications. One can also extract somewhat more sophisticated features related to melodic contour by making a few naïve but often effective basic assumptions, such as the assumptions that all notes belonging to a phrase will be on the same MIDI channel and that phrases will each follow the overall shape of a basic concave or converse arc. Although such assumptions are clearly false, and certainly not acceptable for any wide-ranging analytical framework, they do make it possible to extract some potentially discriminating higher-level melodic features without a sophisticated phrase detection system.

A *melodic interval histogram* is proposed here as a way of facilitating the extraction of certain basic features relating to melodic intervals. Each bin of this histogram represents a different melodic interval, and is labelled with a number indicating the number of semitones in the interval. The magnitude of each bin is set to the number of Note On messages in the piece that have a pitch interval from the preceding Note On message on the same MIDI channel corresponding to the bin label. The direction of the interval (i.e., up or down in pitch) is ignored in this histogram. The histogram is then normalized, so that the magnitude of each bin indicates the fraction of all melodic intervals that correspond to the melodic interval of the given bin.

This histogram clearly has a few limitations. It treats all voices equally, for example, even though the highest line of a piece often carries the most significant melodic information. It is also problematic for polyphonic instruments such as pianos that can play harmonies or multiple melodies simultaneously. It is, however, a quick and easy approach that has been found experimentally to often be helpful in discriminating between classes.

Another intermediate data structure is also used to help calculate some of the features listed below. This consists of an array where each indice corresponds to a MIDI channel and each entry consists of a list of all melodic intervals, in semitones, for the associated channel. The numbers representing the intervals in this second intermediate data structure are set to negative for downward motion and to positive for upward motion.

Once again, all notes occurring on MIDI channel ten are ignored for all of the features described in this section. This is because the "pitch" values on channel ten correspond to percussion patches, not to pitches.

The jSymbolic feature catalogue includes the following features related to melody and melodic intervals:

- **M-1 Melodic Interval Histogram:** A feature vector consisting of the bin magnitudes of the melodic interval histogram described above.
- **M-2 Average Melodic Interval:** The average melodic interval, in semitones.
- **M-3 Most Common Melodic Interval:** The most frequently occurring melodic interval, in semitones.
- **M-4 Distance Between Most Common Melodic Intervals:** Absolute value of the difference between the most common melodic interval and the second most common melodic interval, in semitones.
- **M-5 Most Common Melodic Interval Prevalence:** Fraction of melodic intervals that belong to the most common interval.
- **M-6 Relative Strength of Most Common Intervals:** Fraction of melodic intervals that belong to the second most common interval divided by the fraction of melodic intervals belonging to the most common interval.
- **M-7 Number of Common Melodic Intervals:** Number of melodic intervals that represent at least 9% of all melodic intervals.
- **M-8 Amount of Arpeggiation:** Fraction of melodic intervals that are repeated notes, minor thirds, major thirds, perfect fifths, minor sevenths, major sevenths, octaves, minor tenths or major tenths.
- **M-9 Repeated Notes:** Fraction of notes that are repeated melodically.
- **M-10 Chromatic Motion:** Fraction of melodic intervals that correspond to a semitone.

- **M-11 Stepwise Motion:** Fraction of melodic intervals that correspond to a minor or major second.
- **M-12 Melodic Thirds:** Fraction of melodic intervals that are major or minor thirds.
- M**-13 Melodic Fifths:** Fraction of melodic intervals that are perfect fifths.
- **M-14 Melodic Tritones:** Fraction of melodic intervals that are tritones.
- **M-15 Melodic Octaves:** Fraction of melodic intervals that are octaves.
- **M-16 Embellishment:** Fraction of notes that are surrounded on both sides by Note Ons on the same MIDI channel that have durations at least three times as long as the central note.
- **M-17 Direction of Motion:** Fraction of melodic intervals that are rising rather than falling.
- **M-18 Duration of Melodic Arcs:** Average number of notes that separate melodic peaks and troughs in any channel.
- **M-19 Size of Melodic Arcs:** Average melodic interval separating the top note of melodic peaks and the bottom note of melodic troughs.
- **M-20 Melodic Pitch Variety:** Average number of notes that go by in a channel before a note is repeated. Notes that do not recur after sixteen notes are not counted.

### 4.5.7 Features based on chords and vertical intervals

Chords in general, and tonal harmony in particular, are the areas that have typically received the most attention in traditional Western analytical systems. As a result, there is a great deal of background information that can be taken advantage of in designing features based on chords. Section 4.4 highlights some relevant references that were used to design many of the features in this section. In particular, some of the techniques for chord analysis discussed by Rowe (2001) were particularly useful.

It is essential to point out that the existing theoretical frameworks based on tonal harmony do not apply to many kinds of music, as they were developed primarily with respect to Western classical music. As a consequence, most of the chord-based features proposed as part of the jSymbolic feature library are not based on harmonic function, and emphasize instead basic statistical information about the vertical intervals between pitches that sound simultaneously. Having recognized this, it is once again important to recall that features are useful simply if they help to differentiate between classes statistically, even if they are inspired by theoretical assumptions that do not apply to some of the music under consideration. There are therefore a few features in the jSymbolic catalogue that make use of certain basic concepts that are specific to Western harmonic theory.

Two new histograms are proposed as intermediate data structures for calculating chord-based structures. The first, called a *vertical interval histogram,* consists of bins associated with different vertical intervals and labelled with the number of semitones in the corresponding interval. The magnitude of each bin is found by going through a MIDI recording tick by tick and noting all vertical intervals that are sounding at each tick. This is done exhaustively, so that multiple vertical intervals will be noted per tick if there are more than two pitches sounding simultaneously. The histogram is then normalized. The end result is a histogram that indicates which vertical intervals are most common relative to one another, with a weighting based on duration rather than simply the number of notes sounded corresponding to each interval. This is reasonable, since long notes often have greater harmonic significance than short notes. This histogram does not incorporate any tonal assumptions, although it does require quantization into the twelve standard pitch classes.

It is also potentially useful to have a histogram that can be used to extract features more directly related to tonal harmony, since many music classification projects do involve music based on the basic chord ontologies of Western music. A *chord type histogram* is proposed with this need in mind. This histogram has bins labelled with types of chords: vertical intervals consisting of two pitch classes, minor triads, major triads, other triads, diminished chords, augmented chords, dominant seventh chords, major seventh chords, minor seventh chords, other chords with four pitch classes and chords with more than four pitch classes. The bin magnitudes are calculated by going through MIDI ticks one by one and incrementing the counter for the bin that corresponds to the chord, if any, that is present during a given tick. All inversions are treated as equivalent and octave doubling is ignored in the calculation of this histogram. The histogram is normalized and is weighted based on chord duration, just like the vertical interval histogram.

Neither of these histograms provides any information about arpeggiation, unfortunately, but some information related to this is collected during the melodic feature extraction. A more sophisticated system in the future could integrate vertical statistics with arpeggios, and could also collect information about inversions as well as chord transitions in order to obtain more sophisticated and accurate features.

Once again, all notes occurring on MIDI channel ten are ignored for all of the features described in this section. This is because the "pitch" values on channel ten correspond to percussion patches, not to pitches.

The jSymbolic feature catalogue includes the following features related to chords and vertical intervals:

- **C-1 Vertical Intervals:** A feature vector consisting of the bin magnitudes of the vertical interval histogram described above.
- **C-2 Chord Types:** A feature vector consisting of the frequencies of each of the bins in the chord type histogram discussed above.
- **C-3 Most Common Vertical Interval:** The interval in semitones corresponding to the vertical interval histogram bin with the highest magnitude.
- **C-4 Second Most Common Vertical Interval:** The interval in semitones corresponding to the vertical interval histogram bin with the second highest magnitude.
- **C-5 Distance Between Two Most Common Vertical Intervals:** The difference between the bin labels of the two most common vertical intervals.
- **C-6 Prevalence of Most Common Vertical Interval:** The fraction of vertical intervals corresponding to the most common vertical interval.

- **C-7 Prevalence of Second Most Common Vertical Interval:** The fraction of vertical intervals corresponding to the second most common vertical interval.
- **C-8 Ratio of Prevalence of Two Most Common Vertical Intervals:** The fraction of vertical intervals corresponding to the second most common vertical interval divided by the fraction of vertical intervals corresponding to the most common vertical interval.
- **C-9 Average Number of Simultaneous Pitch Classes**: Average number of different pitch classes sounding simultaneously.
- **C-10 Variability of Number of Simultaneous Pitch Classes:** Standard deviation of the number of different pitch classes sounding simultaneously.
- **C-11 Minor Major Ratio:** Number of minor vertical intervals divided by number of major vertical intervals.
- **C-12 Perfect Vertical Intervals:** Fraction of all vertical intervals that are perfect intervals.
- **C-13 Unisons:** Fraction of all vertical intervals that are unisons.
- **C-14 Vertical Minor Seconds:** Fraction of vertical intervals that are minor seconds.
- **C-15 Vertical Thirds:** Fraction vertical intervals that are thirds.
- **C-16 Vertical Fifths:** Fraction of vertical intervals that are fifths.
- **C-17 Vertical Tritones:** Fraction of vertical intervals that are tritones.
- **C-18 Vertical Octaves:** Fraction of vertical intervals that are to octaves.
- **C-19 Vertical Dissonance Ratio:** Total number of vertical 2nds, tritones, 7ths and 9ths divided by the total number of vertical unisons, 4ths, 5ths, 6ths, octaves and 10ths.
- **C-20 Partial Chords:** Fraction of simultaneously sounding pitch groups that consist of only two pitch classes.

- **C-21 Minor Major Triad Ratio:** Number of minor triads divided by number of major triads.
- **C-22 Standard Triads:** Fraction of all chords that are either major or minor triads.
- **C-23 Diminished and Augmented Triads:** Fraction of all chords that are either diminished or augmented triads.
- **C-24 Dominant Seventh Chords:** Fraction of all chords that are dominant sevenths.
- **C-25 Seventh Chords:** Fraction of all chords that are dominant seventh, major seventh or minor seventh chords.
- **C-26 Complex Chords:** Fraction of all chords that contain more that four pitch classes.
- **C-27 Non-Standard Chords:** Fraction of all simultaneously sounding pitches that consist of more than two pitch class chords and are not major or minor triads or seventh chords.
- **C-28Chord Duration:** Average duration of a chord in seconds.

### 4.5.8 Examples

For the purpose of illustration, Table 4.1 shows the values of twenty sample features extracted from two measures each of a Chopin nocturne and a Mendelssohn piano trio (Figures 4.7 and 4.8, respectively). A comparison of the two examples and their features makes it apparent how such features can be useful in distinguishing between pieces and musical categories of various kinds. For example, the Average Note To Note Dynamic Change (D-4), Overall Dynamic Range (D-1) and Variation of Dynamics (D-2) features demonstrate the greater range in dynamics of the nocturne, the Note Density (R-15) feature demonstrates the greater number of notes per second of the trio, the Orchestral Strings Fraction (I-18) feature indicates that strings play roughly half the notes in the trio but are absent in the nocturne, and the Variability of Note Duration (R-18) feature shows that this portion of the nocturne has more rhythmic variety than the trio. More traditionally-oriented features are also present, such as the Chromatic Motion (M-10) feature, which demonstrates that this portion of the trio has more chromatic motion, or the Range (P-10) feature, which shows that the lowest and highest notes of this section of the nocturne span a greater interval. Although such features are not necessarily significant when considered individually, pattern recognition systems can simultaneously examine many of them in order to find meaningful patterns and discriminate between classes. Of course, the features would typically be extracted over the entire pieces, not just two measures.

| **Feature Name** | ***Nocturne*** | ***Piano Trio*** |
|---|---|---|
| Average Note To Note Dynamics Change (D-4) | 6.03 | 1.46 |
| Chromatic Motion (M-10) | 0.0769 | 0.244 |
| Dominant Spread (P-17) | 3 | 2 |
| Harmonicity of Two Strongest Rhythmic Pulses (R-3) | 1 | 1 |
| Importance of Bass Register (P-13) | 0.2 | 0.373 |
| Interval Between Strongest Pitch Classes (P-6) | 3 | 7 |
| Most Common Pitch Class Prevalence (P-2) | 0.433 | 0.39 |
| Note Density (R-15) | 3.75 | 29.5 |
| Number of Common Melodic Intervals (M-7) | 3 | 6 |
| Number of Strong Pulses (R-8) | 5 | 6 |
| Orchestral Strings Fraction (I-18) | 0 | 0.56 |
| Overall Dynamic Range (D-1) | 62 | 22 |
| Pitch Class Variety (P-9) | 7 | 7 |
| Range (P-10) | 48 | 39 |
| Relative Strength of Most Common Intervals (M-6) | 0.5 | 0.8 |
| Size of Melodic Arcs (M-19) | 11 | 7.27 |
| Stepwise Motion (M-11) | 0.231 | 0.439 |
| Strength of Strongest Rhythmic Pulse (R-4) | 0.321 | 0.173 |
| Variability of Note Duration (R-18) | 0.293 | 0.104 |
| Variation of Dynamics (D-2) | 16.4 | 5.98 |

**Table 4.1:** Comparison of twenty sample features extracted from the first two measures of Fryderyk Chopin’s *Nocturne in B, Op. 32, No. 1* (Figure 4.7) and from measures 10 and 11 of the first movement of Felix Mendelssohn’s *Piano Trio No. 2 in C minor, Op. 66* (Figure 4.8).

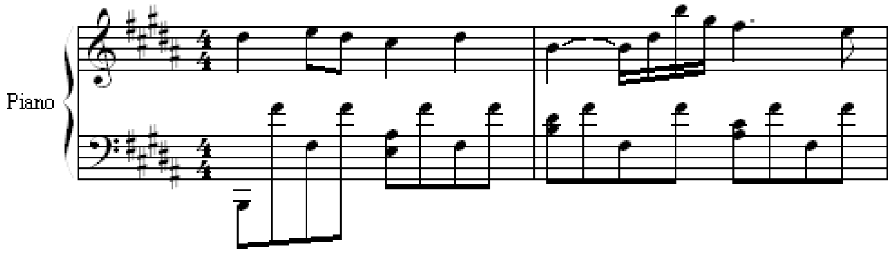


**Figure 4.7:** First two measures of Fryderyk Chopin's *Nocturne in B, Op. 32, No. 1,* from which the features in Table 4.1 were extracted.

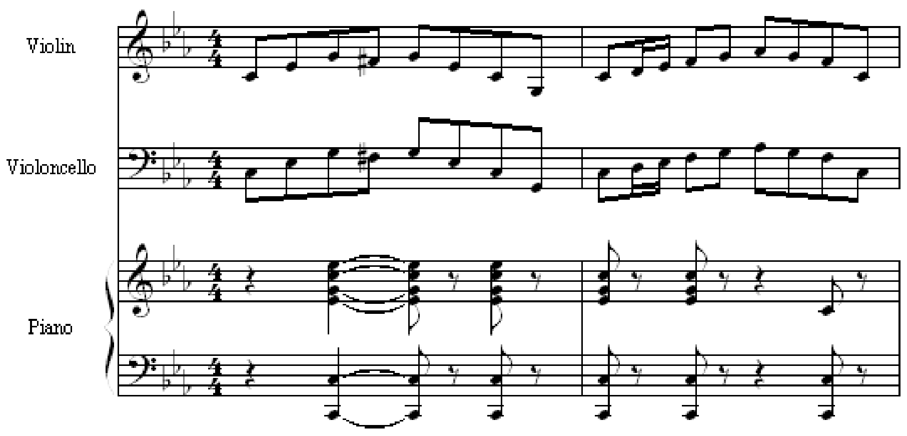


**Figure 4.8:** Measures 10 and 11 of the first movement of Felix Mendelssohn's *Piano Trio No. 2 in C minor, Op. 66,* from which the features in Table 4.1 were extracted.

## *4.6 jSymbolic's functionality*

The essential functionality offered by jSymbolic is the ability to extract features from MIDI files, including files in Format 1 as well as Format 0. Extracted features can be saved as ACE XML files (see Chapter 7), which may then be converted to Weka ARFF files or to other formats. Both the actual feature values and metadata about the features can be saved by jSymbolic, in ACE XML Feature Value files and ACE XML Feature Description files, respectively.

As is the case with all jMIR components, jMIR is designed to serve not only as a feature extraction application, but also as a platform for developing new features. As such, jSymbolic has a modular and extensible architecture with well-documented code. New features can be implemented simply by extending the MIDIFeatureExtractor class. The jSymbolic software then automatically provides the new features with the MIDI data for each file, a library of MIDI processing tools and any other needed feature values, in case a new feature is calculated based on other existing features. As is the case with jAudio, feature extraction order is automatically scheduled dynamically, so all feature dependencies are handled automatically. This approach makes it unnecessary for developers of new features to have any knowledge of how jAudio's interface or control structures work, thereby allowing them to focus directly on feature development itself.

The mechanisms used by jSymbolic to do all of this are very similar to those used by jAudio, as described in Sections 3.4.5 and 3.4.6. However, although jSymbolic does include a number of explicitly implemented metafeatures and aggregators, it does not include automatic metafeature and aggregator feature generation like jAudio, and jSymbolic must still be recompiled when new features are added. Like all jMIR components, jSymbolic's Java implementation is open-source and platform-independent.

The MIDIIntermediateRepresentations and MIDIMethods classes offer developers a variety of MIDI data representation and processing tools. This means that a great variety of features can be implemented at a high level, without actually working directly with the MIDI data, unless, of course, one wishes to.

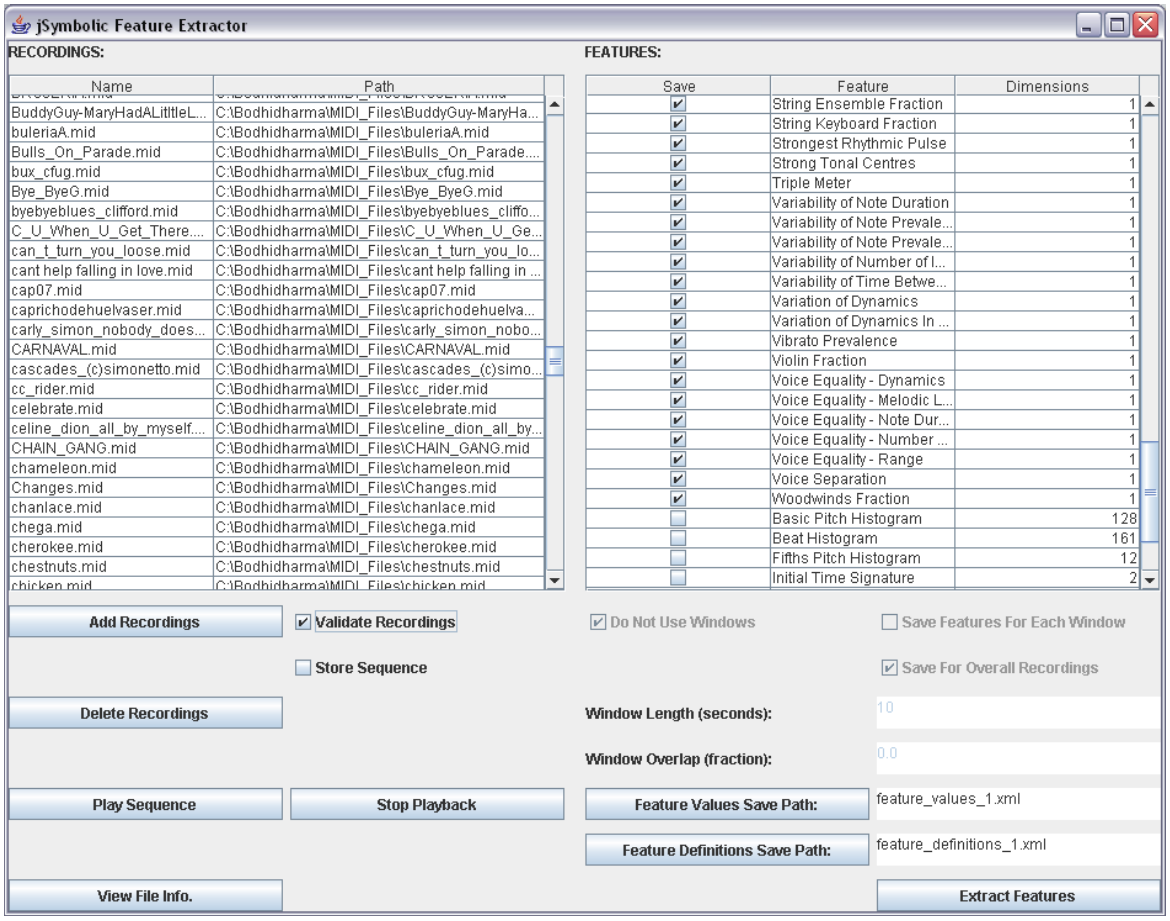
## *4.7 jSymbolic's interface*

jSymbolic is designed for users with varying levels of computer expertise. It is particularly targeted towards musicologists and music theorists who might not currently be using computer technology in their research, so the interface is a simple and easy-to-learn graphical user interface, as shown in Figure 4.9.

This GUI is very simple relative to the other jMIR components, and consists of only a single window. The left side of the window holds an area displaying the MIDI files that have been selected for feature extraction, and the right side includes a list of all features that may be extracted. Individual features may be selected or deselected for extraction. In cases where one feature is selected that requires another feature that is not selected in order to be calculated the software automatically simply extracts the feature that is not selected, uses it in the calculation of the selected feature and then discards its value without saving it.

Basic functionality is also offered for playing MIDI files and viewing metadata about them. There is also a section of the GUI for entering preferences related to windowed feature extraction, but it is currently greyed out, as this functionality is not yet implemented.

Unlike jAudio and ACE, jSymbolic does not yet include a command interface or API designed specifically for integration into external software. The jSymbolic features themselves are very modular, however, and there is a general well-documented API, so jSymbolic's functionality certainly can still be integrated into other software with relatively little work.



RECORDINGS:

| Name | Path |
|---|---|
| BuddyGuy-MaryHadALititleL... | C:\Bodhidharma\MIDI_Files\BuddyGuy-MaryHa... |
| buleriaA.mid | C:\Bodhidharma\MIDI_Files\buleriaA.mid |
| Bulls_On_Parade.mid | C:\Bodhidharma\MIDI_Files\Bulls_On_Parade.... |
| bux_cfug.mid | C:\Bodhidharma\MIDI_Files\bux_cfug.mid |
| Bye_ByeG.mid | C:\Bodhidharma\MIDI_Files\Bye_ByeG.mid |
| byebyeblues_clifford.mid | C:\Bodhidharma\MIDI_Files\byebyeblues_cliffo... |
| C_U_When_U_Get_There.... | C:\Bodhidharma\MIDI_Files\C_U_When_U_Ge... |
| can_t_turn_you_loose.mid | C:\Bodhidharma\MIDI_Files\can_t_turn_you_lo... |
| cant help falling in love.mid | C:\Bodhidharma\MIDI_Files\cant help falling in ... |
| cap07.mid | C:\Bodhidharma\MIDI_Files\cap07.mid |
| caprichodehuelvaser.mid | C:\Bodhidharma\MIDI_Files\caprichodehuelva... |
| carly_simon_nobody_does... | C:\Bodhidharma\MIDI_Files\carly_simon_nobo... |
| CARNAVAL.mid | C:\Bodhidharma\MIDI_Files\CARNAVAL.mid |
| cascades_(c)simonetto.mid | C:\Bodhidharma\MIDI_Files\cascades_(c)simo... |
| cc_rider.mid | C:\Bodhidharma\MIDI_Files\cc_rider.mid |
| celebrate.mid | C:\Bodhidharma\MIDI_Files\celebrate.mid |
| celine_dion_all_by_myself.... | C:\Bodhidharma\MIDI_Files\celine_dion_all_by... |
| CHAIN_GANG.mid | C:\Bodhidharma\MIDI_Files\CHAIN_GANG.mid |
| chameleon.mid | C:\Bodhidharma\MIDI_Files\chameleon.mid |
| Changes.mid | C:\Bodhidharma\MIDI_Files\Changes.mid |
| chanlace.mid | C:\Bodhidharma\MIDI_Files\chanlace.mid |
| chega.mid | C:\Bodhidharma\MIDI_Files\chega.mid |
| cherokee.mid | C:\Bodhidharma\MIDI_Files\cherokee.mid |
| chestnuts.mid | C:\Bodhidharma\MIDI_Files\chestnuts.mid |
| chicken.mid | C:\Bodhidharma\MIDI_Files\chicken.mid |

FEATURES:

| Save | Feature | Dimensions |
|---|---|---|
| ✓ | String Ensemble Fraction | 1 |
| ✓ | String Keyboard Fraction | 1 |
| ✓ | Strongest Rhythmic Pulse | 1 |
| ✓ | Strong Tonal Centres | 1 |
| ✓ | Triple Meter | 1 |
| ✓ | Variability of Note Duration | 1 |
| ✓ | Variability of Note Prevale... | 1 |
| ✓ | Variability of Note Prevale... | 1 |
| ✓ | Variability of Number of I... | 1 |
| ✓ | Variability of Time Betwe... | 1 |
| ✓ | Variation of Dynamics | 1 |
| ✓ | Variation of Dynamics In ... | 1 |
| ✓ | Vibrato Prevalence | 1 |
| ✓ | Violin Fraction | 1 |
| ✓ | Voice Equality - Dynamics | 1 |
| ✓ | Voice Equality - Melodic L... | 1 |
| ✓ | Voice Equality - Note Dur... | 1 |
| ✓ | Voice Equality - Number ... | 1 |
| ✓ | Voice Equality - Range | 1 |
| ✓ | Voice Separation | 1 |
| ✓ | Woodwinds Fraction | 1 |
| ☐ | Basic Pitch Histogram | 128 |
| ☐ | Beat Histogram | 161 |
| ☐ | Fifths Pitch Histogram | 12 |
| ☐ | Initial Time Signature | 2 |

[✓] Validate Recordings
☐ Store Sequence
[✓] Do Not Use Windows
☐ Save Features For Each Window
[✓] Save For Overall Recordings
Window Length (seconds): 10
Window Overlap (fraction): 0.0
Feature Values Save Path: feature_values_1.xml
Feature Definitions Save Path: feature_definitions_1.xml

**Figure 4.9:** A screen shot of the jSymbolic GUI interface.

## 4.8 Summary of original contributions

jSymbolic is both a dedicated symbolic feature extraction application and a platform for feature development. It is the only application currently available designed specifically for MIR-oriented symbolic feature extraction that is intended for general use.

jSymbolic also includes 111 implemented features, far more than any other existing software that can be used for symbolic feature extraction.

The jSymbolic feature catalogue, as described in Section 4.5, is also an important contribution. It includes 153 features, including features relating to instrumentation, texture, rhythm, dynamics, pitch statistics, melody and chords. These features, as an aggregate, are designed to be relevant to a wide variety of musics, and can be adapted for use in areas such as analytical theoretical research and empirical musicological research, not just MIR and automatic music classification. This feature catalogue is, to the best of the author's knowledge, the largest and most diverse high-level musical feature catalogue currently available. This catalogue includes many original features.

Section 4.4 also offers important contributions by discussing important issues relevant to feature design and selection, and by offering guidelines for developing new high-level features. This discussion takes into account insights from a variety of disciplines, and can serve as a useful resource if expanding the jSymbolic feature catalogue or designing an entirely new catalogue of high-level features. The topics dealt with in this sub-section are particularly significant to MIR research in automatic classification, as most research in this field has been technically motivated, and has often failed to take into account highly relevant insights offered by disciplines such as musicology and music theory.

## 4.9 Future research

As mentioned above, jSymbolic is intended not only as a tool for extracting features from music stored in symbolic music files, but also as a platform for iteratively developing new features. Correspondingly, an important part of future research will focus on designing and implementing new features. The first priority will be the implementation of the 42 remaining unimplemented features in the current jSymbolic feature catalogue, so that all 153 features will be implemented. Once this is done, the catalogue itself can be expanded, with special attention paid to features that represent increasingly higher-level information.

The implementation of pre-processing functionality of various kinds could also be very useful. For example, automatic voice partitioning algorithms could help detect and correct MIDI files where voices are multiplexed on a single channel or track using a single patch. Phrase detection pre-processing would also make it possible to extract features relating to melodic contour more reliably.

An important design priority of the current jSymbolic feature catalogue was to ensure that the majority of features could be applied to a broad range of musics. As a result, the catalogue contains only a limited number of features that are designed for specific types of music, such as Western tonal music, for example. New features and pre-processing functionality related specifically to harmonic analysis, for example, could be of particular utility to researchers only interested in such particular types of music.

It could be useful to build even a relatively rudimentary automatic harmonic analysis pre-processing system into jSymbolic if sophisticated analytical techniques are too difficult to implement automatically. Work such as that by Raphael and Stoddard (2003) or the techniques used by Rowe (2001) could be useful in this respect. Features derived from simple Roman numeral analysis and chord voicings could be extracted, for example. Although such automatically generated analyses would very likely contain errors due to the difficulty and subjectivity of harmonic analysis, and would have limited value in and of themselves, features derived from such analyses could still provide rough but effective information that could help distinguish between classes. The ability to intelligently take arpeggios into account as well as vertical chords would also be valuable.

More sophisticated statistical analyses could also be applied to the jSymbolic's different histogram features. The calculation of higher-order moments, skew, scatter and excess, for example, could all be useful. Gabura's work (1965) provides a good starting point for such an approach. A more in-depth study of the techniques used by ethnomusicologists to compare different types of music could also provide additional ideas. It might also be beneficial to use alternative ways of representing pitch, as suggested by Chai and Vercoe (2001).

Further research on more sophisticated features based on sequential information could be useful as well. Phrasing and repetition, in terms of melodies, chord progressions and rhythms, can be very important musically. The degree and regularity with which such patterns are repeated, as well as their general character, length and register, could all furnish useful features. It could also be helpful to collect features related specifically to transposition, decoration and inversion of motifs.

There has been some interesting research done on models of how humans code sequential musical information and on applications of generative grammars to music. Stevens and Latimer (1997) present relevant references on both the strengths and weaknesses of such approaches. Research on detecting and processing repeating musical patterns, such as that by Hsu et al. (2001), Typke et al. (2003) or Lartillot (2003), could be worthwhile. Work such as that by Tseng (1999) or Foote (1999b) could also be valuable in devising a system to extract melodies and segment recordings. An alternative and potentially very interesting approach to extracting features from phrases would be to characterize melodies by fitting them to functions, as was done by Laine and Kuuskankare (1994), in order to search for patterns and then apply functional data analysis techniques.

Existing research on query-by-humming systems could also provide a useful resource for the extraction of features based on sequences and phrases. Features could be extracted by collecting and analyzing n-grams based on melodies, rhythms and chord progressions. There are a number of resources that could be beneficial in this respect (Agrawal and Srikant 1995; Hsu, Liu and Chen 2001; Selfridge-Field 1998; Uitdenbogerd and Zobel 1998). The work of Shan and Kuo (2003) could also be of particular use, as it considers the problem in the context of style classification. The work of Yip and Kao (1999) also provides some helpful background on melody-based features.

It would also be convenient if jSymbolic could extract features from other symbolic file formats, such as Humdrum **Kern, MusicXML or OSC. Functionality could be built into jSymbolic to either extract features from such formats directly, or at least to automatically translate them into MIDI streams from which features could then be extracted.

The ability to extract features over small windows of time instead of only for files as a whole could also be advantageous. This includes the use of disjoint as well as overlapping windows. The musical characteristics of different parts of a piece can vary significantly, with the consequence that feature values that are the average of such dissimilar sections may in fact not be representative of any of the sections. Windowed feature extraction is also a requirement for automatically segmenting pieces or performing general structural analyses. The infrastructure for such windowed feature extraction is already in place in jSymbolic, and has already been implemented in jAudio.

jAudio also includes other useful functionality that could be ported to jSymbolic. This includes automatic metafeature and feature aggregator generation, as well as the ability to add new features as plug-ins without recompiling the software. The ability to save feature values as ARFF files directly would also be a helpful addition

Additional ways of accessing jSymbolic's functionality will also be implemented. A command-line interface would facilitate simple batch processing, and the jSymbolic API could be improved to even further facilitate the ability to incorporate jSymbolic's functionality into other software. The jSymbolic user documentation could also be improved to the level of jWebMiner and jMusicMetaManager, although jSymbolic's documentation is already at least as good as most academic software systems.

# 5. jWebMiner: Extracting cultural features from the Internet

## *5.1 Overview of community metadata, cultural features and jWebMiner*

There is psychological and musicological reason to believe that cultural factors beyond the content of music itself play an essential role in how humans interpret and organize music. Fabbri (1981), for example, has argued that content-based aspects of music represent only one of five ways in which musical genres can be characterized, and North and Hargreaves (1997) found experimentally that the style of a piece can influence listeners' liking for it more than the piece itself. Cano and Koppenberber (2004) also found experimentally that the nature of cultural data accessible on the Internet has intriguing potential for MIR research.

Although there are many sources of cultural information that bear eventual targeted investigation, ranging from social networking data to critical writings to sales statistics, the decision was made for the purpose of jMIR to focus on the largest source of information available: text data on the web, which will be referred to here as *community metadata*. Rather than attempting to specialize in extracting features from only one or a few types of data sources, it was decided that it would be more profitable to make use of the functionality offered by search engines to extract information from the web as an amalgamated whole. In addition to the huge scope and diversity of information that may be mined, this approach has the additional significant advantage that the information on the web constantly self-updates.

In order for cultural data of any kind to be beneficial for the purposes of jMIR, or machine learning in general, it is necessary to first find ways of automatically extracting useful information in a systematic way and then, ideally, formulating it so that it can be expressed in the form of simple numerical features. Such features are referred to here as *cultural features*.

The jMIR software component developed to perform this cultural feature extraction has been named *jWebMiner* (McKay and Fujinaga 2007a). The details of its operation and interface are described in Sections 5.4 and 5.5. The Java bytecode, the source files and the manual are freely available at jmir.sourceforge.net/index_jWebMiner.html.

## *5.2 Background information*

### 5.2.1 Screen scraping and web services

The most direct way of extracting data from the web is called *screen scraping*. In essence, this involves downloading pages from a web site as if they were to be viewed on a web browsing application and then parsing them in some way to extract meaning. This can work well if the structuring and formatting of pages on the web site consistently conform to a set template. In practice, however, this approach can be problematic, as contemporary web sites typically use a variety of difficult-to-parse scripting languages to structure their web pages, and pages from a site are often not perfectly consistent with one another. Even a minor change to a site's design can cause the screen scraping software to become entirely ineffective. So, while screen scraping can be useful as a last resort, it is preferable to avoid it if possible.

*Web services* offer an alternative to web scraping. Some web sites choose to allow users to submit queries and receive results over a network using a platform-independent standardized protocol. It is thus possible to submit queries to Google, for example, and receive responses in a way that bypasses the HTML-based web site itself. Examples of web service protocols include REST, SOAP, BEEP and many others.

In theory, web services make it possible for applications to access a site's content without being disturbed by cosmetic or structural changes to the site. In practice, however, web sites can and do discontinue web services or change protocols, as happened in December 2006 when Google deprecated its SOAP API in favour of an AJAX API. Nonetheless, web services in general offer a much easier, more consistent and more reliable way of accessing data on the Internet compared to screen scraping. Web sites often improve convenience by providing simple APIs for accessing their web services, and often make available free corresponding code libraries for major programming languages such as Java or C++.

### 5.2.2 Search engines

Web search engines are some of the most powerful tools that can be taken advantage of using web services. Different search engines use a variety of algorithms to search the web, ranging from a significant amount of manual human editorial input to fully automated web crawlers. The most famous algorithm is arguably Google's[124] PageRank (Brin and Lawrence 1998), which causes sites crawled by the Goobglebot web crawler to be favoured in search result rankings based on how many other sites link to them. Google also uses many other criteria to generate results, including machine learning. Google's predecessor as the most popular search engine, Yahoo!,[125] began essentially as a set of bookmarks organized into a hierarchical set of categories, but grew to first use Google results and then its own web crawling-based search engine. Yahoo! has the advantage of offering a specialized music service.[126]

Although there are many good quality search engines, Google and Yahoo! have the special advantages of being particularly popular and of offering extensive web services. They are therefore the services emphasized by jWebMiner. Yahoo! offers "REST-like" web services, for which a simple overview has been written by Wenz (2007). The Google web services used by jWebMiner are based on the SOAP API, and are well documented by Calishain and Dornfest (2003). The SOAP API was used rather than its AJAX replacement because the AJAX API only returns HTML-formatted results which must then be scraped to be used in cultural feature extraction, a process that undermines one of the essential advantages of web services.

### 5.2.3 Sources of community metadata

The web as a whole provides a rich source of community metadata that has been taken advantage of as a whole by a number of researchers. However, as can be seen in Section 5.3, other researchers have largely chosen to pay particular attention to particular sites in an attempt to improve musical relevancy of results. Although there are far too many sites to cover with any completeness here, some key sites are briefly reviewed

[124] www.google.com
[125] www.yahoo.com
[126] music.yahoo.com

below. In general, the following categories of cultural information can be particularly helpful:

- Data from social networking web sites and music recommendation systems
- Traffic and individual usage data from peer-to-peer systems
- Published listener or radio playlists and compilation album track listings
- Interviews with musicians and composers
- The writings of musicologists, critics and bloggers
- Surveys and other methodological listener studies
- Sales, marketing and other industry data such as Billboard listings or results of awards shows

Last.FM[127] has received extensive attention from the MIR community, particularly since it began providing free access to its data via the Audioscrobbler[128] web services portal. Last.FM is an Internet radio station that uses the Audioscrobbler music recommendation system to recommend artists to users based on a collaborative filtering algorithm (see Section 5.2.4). A great deal of useful information can be freely accessed via Audioscrobbler's web services, including information such as artists and tracks preferred by particular users, metadata tags entered by users, similarity data for each artist amalgamated from many users, and much more.

There are a great many other existing commercial music recommendation systems that can potentially yield useful cultural features. Pandora is worth particular mention here because of its popularity.[129] Like Last.FM, Pandora couples an Internet radio station with music recommendation. However, Pandora bases recommendations primarily on expert-entered descriptive musical "attributes" (e.g., syncopation, key tonality, etc.) rather than Audioscrobbler's collaborative filtering. Unfortunately, Pandora is not available outside the U.S.A., and it does not offer free web services.

[127] www.last.fm
[128] www.audioscrobbler.net
[129] www.pandora.com

The AllMusic[130] site offers an invaluable source of musical data. It contains information on an essentially comprehensive list of music and artists, including extensive formatted metadata (including MIR-relevant fields such as genre, mood, style, themes, influences and similar artists and albums) as well as reviews and artist biographies. All information is entered by professional reviewers and editors. The key disadvantage of the site is that it does not offer free web services, which means that data must be either manually collected or scraped.

MusicBrainz[131] is a free audio fingerprinting (song identification) site that allows fingerprinting to be performed and metadata to be accessed via web services. Art of the Mix[132] publishes user playlists. freeDB[133] also allows information such as track listings or music tags to be downloaded. Gracenote[134] is a commercial site that maintains track listings of CDs including commercial CDs.

Established music blog sites, such as Pitchfork[135] or The Hype Machine,[136] are another useful source of community metadata, although they have a tendency to emphasize new music over older music. Social networking sites such as MOG[137] can also be rich sources of information. Unfortunately, such sites usually do not offer web services and often require logins. Information can sometimes be extracted from them indirectly using search engines, however.

Some general sites can also provide musical information that is particularly useful. Amazon,[138] for examples, allows data on purchasing patterns to be accessed via web services. Wikipedia[139] or Citizendium[140] can also be useful when queried with artist names, for example, although they do not yet offer web services.

[130] www.allmusic com
[131] musicbrainz.org
[132] artofthemix.org
[133] www.freedb.org
[134] www.gracenote.com
[135] www.pitchforkmedia.com
[136] hypem.com
[137] mog.com
[138] www.amazon.com
[139] wikipedia.org
[140] citizendium.org

### 5.2.4 Collaborative filtering and co-occurrence analysis

*Collaborative filtering* is one common way of extracting information from community metadata, and is generally related to recommendation-based applications. Collaborative filtering involves drawing links between entities, such as CDs or books, based on the expressed shared interests of people who share similar taste profiles and preference behaviours. Vendors such as Amazon, for example, use collaborative filtering to recommend items to customers that many other customers who have purchased similar items in the past have also purchased. If, for example, Customer A has bought many Duke Ellington CDs, and many other customers who bought Duke Ellington CDs have also purchased Count Basie CDs, then Amazon would recommend Count Basie to Customer A. There is a large body of existing research on collaborative filtering (e.g., Shardanand and Maes 1995; Herlocker, Konstan and Riedl 2000; Herlocker et al. 2004).

Although collaborative filtering can certainly be useful for research areas such as musical similarity analysis and music recommendation, it suffers from a number of well-known weaknesses:

- A very large user group is needed in order for results to be meaningful.
- Items that have not been purchased by many other users are often ignored by collaborative filtering in favour of more popular choices. This is particularly problematic, as discovery of unknown music is a key aim of areas such as music recommendation, and recommending music to a user that is so well known that s/he is already aware of it is not useful. It can be especially difficult for new items not supported by strong external publicity campaigns to reach significant mass in the filtering mechanism to be recommended to users by the system. This is known as the *cold start problem.*
- Even when techniques are implemented to attempt to counteract popularity biases, potentially significant bootstrapping time is still needed for new items to gain sufficient mass to be recommended via collaborative filtering.
- Individuals who purchase items as gifts for others can introduce significant noise.

- There is a bias against eclectic taste profiles and algorithms tend to have difficulty escaping from preference clusters once they are established. Such problems are well documented by Epstein (1996), among others.

*Co-occurrence analysis* offers a useful alternative. Co-occurrence analysis is based on the idea that if two items appear in the same context, such as a web page or a music playlist, then there is likely some relatedness or similarity between them. In essence, collaborative filtering can be viewed as a special case of co-occurrence analysis. There has been a significant amount of research in linguistics indicating that similarity measurements based on co-occurrence analysis are cognitively reasonable (e.g., Schütze 1992; Lowe and McDonald 2000).

For the sake of clarity, the term *co-occurrence* will be used here to refer specifically to the occurrence of two strings *from the same set of strings* in the same document (e.g., web page). *Cross tabulation* will be used here to refer specifically to the occurrence of two strings *from different sets of strings* on the same document. For example, examining how often different artists co-occur on the same document would correspond to co-occurrence, and measuring how often artist names co-occur with genre names would correspond to cross tabulation. Although generally speaking co-occurrence in fact refers to either of these two scenarios, this special distinction is made here for the sake of clarity.

As found experimentally by Aucouturier and Pachet (2003), co-occurrence analysis tends to generate meaningful clusters of similar entities. In the case of music, they found that these clusters tend to be based in general on thematic, genre and period relationships, although relationships can also be more "metaphorical." The vague nature of the clusters is both a disadvantage and a benefit from the perspective of MIR applications, in the sense that they are poorly suited for precise classification, but can be interesting for music discovery. It should be noted that Aucouturier and Pachet did not perform cross tabulation experiments with artist names and genre names, for example, and such an approach is likely better suited for direct classification, as was found in a number of the research projects discussed in Section 5.3.

With respect to the web, one way of using co-occurrence analysis is to measure how often or in which ways certain terms co-occur on the same web pages or web sites, usually based on search engine hit counts. This is the essence of the approach used by jWebMiner. The fundamental assumption here is that there is a sufficient amount of expertise embedded in the web to overwhelm erroneous or misleading information that is also present. As will be presented throughout this chapter, there are a number of ways of helping to improve the relevancy and correctness of the data accessed by features that are extracted using co-occurrence analysis. In particular, co-occurrence analysis algorithms based on web search engine hit counts should consider and correct for the following sources of noise:

- Synonyms can be problematic. For example, if one is extracting hit counts for the funk musician *George Clinton,* one would like to ensure that hits do not instead refer to the former U.S. Vice-President. Fortunately, such noise can at least be mitigated by requiring that filter words such as *music* also be present to help hit counts bias towards relevant hits.
- Misspellings or variations of a string (e.g., *Mister Mister* compared to *Mr. Mister*) in web pages or queries can artificially reduce counts. Search engines such as Google can account for such problems to some extent, but cannot always be relied upon.
- Strings consisting of multiple tokens may be present on a web page in various orders or subsets (e.g., *Buddy Guy* vs. *Guy, Buddy* vs. *Guy* vs. *Buddy*). Services such as Google are once again good at dealing with varying token orderings, but do not account for subsets unless they use OR-based searches that can introduce more distortion than they prevent (e.g., both the queries *Buddy* and *Guy* would result in many hits unrelated to the artist Buddy Guy).
- It can be difficult to ensure relevancy of hits. There is no guarantee that hits found on the same document indicate relatedness in the way that a researcher is looking for. For example, several years ago searches for *Chistina Aguilera* and *Eminem* would have resulted in a very high co-occurrence because of a feud between the two. This would have therefore resulted in a misleadingly high musical similarity measure.

- True private behaviour might not correspond to public behaviour. An individual might like listening to Paris Hilton, for example, but might be embarrassed by this, and would not publicize it on his or her blog or published playlists.
- Negative meanings can be deceptive. If attempting to classify songs by mood, for example, the song *Gloomy* Sunday might be referred to as *not happy* on a web site, but a naïve feature extractor might mark this as a co-occurrence between *Gloomy Sunday* and *happy*.

In general, the huge amount of information available on the web does help to smooth out erroneous hits, particularly when counts are extracted using well-crafted filtering and other techniques described in the following sections. Nonetheless, the precision and, in some limited but significant cases, the accuracy of hit counts can be limited, so features extracted using co-occurrence analysis should be treated as only approximations, albeit potentially very useful approximations.

As a final note, query complexity must be considered when choosing a co-occurrence analysis algorithm. Algorithms designed to counteract popularity biases (due to there being many more web pages on Rihanna than the Zoobombs, for example) that have exponential search complexities of O($n^2$), or worse, will not scale to realistic large-scale applications, particularly since many web services can consume several seconds per query, and there are often limits on daily usage.

### 5.2.5 More sophisticated text processing techniques

There is a significant body of research available on more sophisticated ways of mining information from text data that is far too large to cite with any completeness here. That being said, some of this work has been used in previous MIR research (see Section 5.3), and it could certainly be useful in future research. Of particular interest is work such as Brill's (1992) on classifying words into nouns, verbs, pronouns, adverbs, etc., or the work of Evans and Zhai (1996) and Evans and Klavans (2000) in extracting "noun phrases" from text documents. A noun phrase is essentially a noun packaged with descriptive text surrounding it. Noun phrases are extracted using pieces of software called *NP chunkers*, and can be very useful in extracting units of meaningful data from streams of text. A particularly useful and simple technique for the purpose of this and other text-processing tasks is to pre-process text by breaking it into *n-grams*, which are sequences of ordered words each consisting of *n* words.

## 5.3 Previous music information retrieval research

To the best of the author's knowledge there is no MIR research software other than jWebMiner that has been specifically designed for general-purpose cultural feature extraction and collaborative development and expansion by the MIR community. There are, however, a number of relevant publications related to using community metadata in specific MIR research areas such as playlist generation and music recommendation. This section describes some key examples, with an emphasis on surveying the variety of approaches that have been used to attempt to counterbalance problems such as popularity bias.

The earliest MIR research in automatically extracting information from community metadata focused on collaborative filtering (e.g., Cohen and Fan 2000; French and Hauver 2001; Pestoni et al. 2001). Pachet and his colleagues (2001) built upon this by utilizing co-occurrence and correlation techniques applied to radio playlists and compilation CD databases for the purpose of artist and song title classification. They calculated a normalized co-occurrence, $\mathrm{Cooc}_{\mathrm{norm}}$, as follows:

$$Cooc_{norm}(a,b) = \left( \frac{C(a,b)}{C(a)} + \frac{C(a,b)}{C(b)} \right) / 2 \tag{5.1}$$

where *a* and *b* indicate two different song titles and *C* is the number of documents that contain the indicated title(s). A similarity distance between titles, $S_1$, was defined as:

$$S_1(a,b) = 1 - Cooc_{norm}(a,b) \tag{5.2}$$

A more sophisticated similarity measurement, $S_2'$, was also used that takes indirect links into account (i.e., if *Stevie Ray Vaughan* co-occurs with *Eric Clapton* and *Eric Clapton* co-occurs with *Robert Johnson*, then this could indicate a link between *Stevie Ray Vaughan* and *Robert Johnson*):

$$S_2'(a,b) = \frac{Cov(a,b)}{\sqrt{Cov(a,a) \times Cov(b,b)}} \tag{5.3}$$

where the covariance, *Cov*, is defined as:

$$Cov(a,b) = E((a - \mu_a) \times (b - \mu_b)) \tag{5.4}$$

*E* is the mathematical expectation and $\mu_i$ is the expected value of *a* or *b*. The distance between *a* and *b*, $S_2$, was then calculated as:

$$S_2(a,b) = 1 - \left(1 + S_2'(a,b)\right)/2 \tag{5.5}$$

Whitman and Lawrence (2002) were among the first to use more sophisticated text processing techniques. They based their work upon a seed list of artist names and mined manually entered similarity data from AllMusic. They also queried search engines using the name of each artist and the filter terms *music* and *review*. The top 50 pages were then downloaded (the assumption being that the first pages returned are the most relevant.), and text was extracted from them and divided into noun phrase n-grams surrounding artist names. Statistical features were then calculated based on how often various terms occurred in relation to each artist, and artist similarity was measured based on the overlap of term frequencies (after various types of pre-processing).

Whitman and Lawrence also applied a collaborative filtering approach to peer-to-peer data available at the time. The following similarity measure, $S_3$, is an attempt to counteract the preference for well-known artists endemic to collaborative filtering approaches:

$$S_3(a,b) = \frac{C(a,b)}{C(b)}\left(1 - \frac{|C(a) - C(b)|}{C(c)}\right) \tag{5.6}$$

Here *a* and *b* each indicate two different artists, *c* is the most popular of all artists in the set of all artists under consideration, *S* indicates their similarity score and *C* indicates the number of peer-to-peer users that have the indicated artist(s) in their set. The second term is a popularity cost that decreases the similarity measure if one artist is much more popular than the other.

Whitman and Smaragdis (2002) combined a variant of this work with audio features to arrive at an impressive combined success rate of 100% when classifying among five musical genres. They also proposed as future research the intriguing idea of using a *culture ratio* to measure the relative effectiveness of cultural features relative to audio features for dealing with particular classes. This could then be used to differentially weight cultural and audio features in different classification scenarios.

Baumann, Klüter and Norlien (2002) also combined audio and community metadata, but with an emphasis on retrieving music using natural language queries. Labelling for audio segments was acquired from Gracenote, freeDB and the ID3 tags of MP3 files, an approach that can unfortunately produce very noisy and unreliable results. This publication also includes some very interesting work on lyric analysis and natural language processing in general.

Baumann and Hummel (2003) built on this work by applying it to the problem of artist recommendation. Like Whitman and Lawrence, they downloaded 50 web pages for each artist that they were studying and divided them into n-grams. They then weighted resulting terms using TFIDF weightings, which are based on the number of occurrences of each term in each set of 50 pages and the number of occurrences of the term in the pages downloaded for all artists as a whole. Unlike Whitman and Lawrence, they used a simple HTML-filtering stage to attempt to remove noise such as advertisements added to pages. They based their ground-truth evaluations on AllMusic and Yahoo! Launch.[141]

Ellis and his colleagues (2002) approached community metadata as a source of ground-truth for evaluating audio-based classification, rather than as a source of features to be used in classification themselves. In addition to using the techniques used by Whitman and his colleagues discussed above, they also performed a web survey asking informants to evaluate the similarity between various artists in a number of ways.

Zadel and Fujinaga (2004) made use of Amazon and Google web services to generate clusters of related artists. Amazon Listmania! Lists were traversed, starting from initial seed artists, to find potentially related artists. Google was then used to evaluate the co-occurrence of pairs of artists on web pages. Relatedness, $S_4$, was calculated as follows:

$$S_4(a,b) = \frac{C(a,b)}{\min\left(C(a),C(b)\right)} \tag{5.7}$$

Here *a* and *b* each indicate two different artists and *C* indicates the number of web pages containing the indicated artist(s).

Celma, Ramírez and Herrera (2005) proposed the alternative approach of extracting information from RSS feeds and FOAF documents.[142] This approach combined information about personal characteristics of users with their explicit musical preferences, and made use of sources of information previously untapped by the MIR community.

[141] The precursor of Yahoo! Music.
[142] www.foaf-project.org

Geleijnse and Korst (2006b) used Google to classify artists by genre and by mood using cross tabulation. The first method used was based on simple hit counts, with a correction for popularity bias based on pointwise mutual information theory (Manning and Schütze 1999). Each artist *a,* from the set of artists *A*, was assigned a score, $T_1$*,* indicating its membership in class *g* according to the following equation:

$$T_1(a,g) = \frac{C(a,g)}{1+\sum_{i\in A} C(i,g)} \tag{5.8}$$

where *C* indicates the number of web pages containing its parameters.

Geleijnse and Korst also experimented with a "pattern-based mapping" approach designed to help improve relevancy of hits. Instead of searching for artist *a* and class *g* when finding *C(a,g)*, they searched for phrases (or "patterns") such as a *is one of the biggest* g *artists* used in combination with phrases such as *artists such as* a or g *artists such as*. *C(a,g)* was thus recalculated as:

*C'(a,g) =* *"number of occurrences of* a *when querying patterns containing* g*"* +
*"number of occurrences of* g *when querying patterns containing* a*"* (5.9)

Geleijnse and Korst have considered both manually formatted patterns and automatically generated patterns (2006a). This approach also relates to work done by others, such as that by Cimiano and Staab (2004).

The third technique used by Geleijnse and Korst ("document-based mapping", similar to the work done on styles in the visual arts by Boer, Someren and Wielinga (2006)) was to download the first *k* web pages found by Google for artist *a* and then count the number of occurrences of each *g* in all of the downloaded pages (as opposed to the number of pages containing g, as was done in the first two techniques). The same was also done with the roles of *a* and *g* reversed. *C(a,g)* was then redefined as:

*C"(a,g) =* *"number of occurrences of* a *in documents found with* g*"* +
*"number of occurrences of* g *in documents found with* a*"* (5.10)

In order to improve results, Geleijnse and Korst posited that artists who belong to the same class are likely to be mentioned together on the same web pages more often than artists in different classes. A similarity rating for the artists *a* and *b*, $S_5$, was then calculated:

$$S_5(a,b) = \frac{C(a,b)}{1+\left(\sum_{i\in A, i\neq a} C(a,y) \times \sum_{j\in A, j\neq b} C(j,b)\right)} \tag{5.11}$$

where *C(a,b)* is the co-occurrence count of *a* and *b* on the same web page, *A* is the set of all artists studied and $a,b \in A$. This was implemented with both pattern-based and document-based mappings, as well as simple page-count mappings. $T_1$ was then calculated only for those *n* artists with the highest $S_5$ scores.

Geleijnse and Korst also used various synonyms to improve results. They, for example, combined and normalized results for *Indie*, *Alternative Rock* and *Alternative Rock/Indie*. They also used the filter word *music*. In the end, they found that document-based mapping approach was the best overall performer for both genre and mood classification.

Schedl and his colleagues (2006) experimented with using Google to classify artists by genre using co-occurrence analysis with two different scoring metrics. They also compared the effectiveness of the following filter string combinations: *music*, *music* + *genre*, *music* + *style* and *music* + *genre* + *style*. The two metrics used to measure an artists *a*'s membership in a genre *g*, $T_2$ and $T_3$, were:

$$T_2 = \frac{C(a,g)}{C(a)} \tag{5.12}$$

and

$$T_3 = \frac{C(a,g)}{C(g)} \tag{5.13}$$

where *C* indicates the number of web pages containing both *a* and *g*, just *g* or just *a*, as indicated. Schedl and his colleagues found that $T_2$ worked better with broad genre classes and $T_3$ worked better with narrower classes.

This work was based on previous work by varied subsets of the authors. Schedl, Knees and Widmer (2005a) used Google page counts to measure how often different artist names were mentioned on the same web pages, and then used this to construct an asymmetric artist similarity matrix. This was used to classify artists by genre based on k-nearest neighbour similarity to artists with known genres. The filter words *music* and *review* were used, and results were combined with different results where both artist names had to occur specifically in the title of a page for it to be counted as a hit.

Schedl, Knees and Widmer (2005b) used a similar technique to visualize artist similarities using genre prototypes as reference points. They later added a penalization function to counteract distortion due to bands whose names equal common speech words, such as *Kiss* (2006). Geleijnse and Korst (2007) have also developed an algorithm for dealing with such ambiguous artist names, by using the Google *define:* function to approximate the number of other definitions corresponding to an artist name, and using this to approximate the probability that a hit corresponds to the actual artist. Schedl et al. (2007) also used named entity detection and rule-based linguistic analysis to extract band members and instrumentations from the web.

In an earlier and somewhat different work, Knees, Pampalk and Widmer (2004) investigated genre classification and artist similarity measurement using a machine learning approach. They queried both Google and Yahoo! using artist names and a variety of filter words (combinations of *music*, *review*, *genre* and *style*) and downloaded the top 50 pages from each search engine. HTML tags and English stop words (e.g., *a, and*, *or*, *the*, etc.) were then removed. The number of occurrences of each artist, each unique "term" (word) and each term cross referenced with artists on the same page as the term were all counted. In order to reduce the thousands of resulting terms, all terms that did not occur in at least five of the downloaded pages and that did not pass a $\chi^2$ test were deleted. The result was a normalized vector of artist term weights, which was then fed to a support vector machine classifier to classify based on genre, to a k-nearest neighbour classifier to evaluate similarity and to self-organizing maps to visualize the artist space. Experimental results indicated that daily Internet fluctuations did not significantly impact results, that Google outperformed Yahoo!, and that the combination of the filter words *music* and *review* outperformed other filter word sets.

A related approach was also used, combined with audio features, to provide an interface for visually mapping music collections (Knees et al. 2006). These techniques were further developed into an additional interface for browsing music spaces (Pohle et al. 2007b) based on a technique where non-negative matrix factorisation was used to relate artists to archetypical bases (Pohle et al. 2007a).

Bergstra, Lacoste and Eck (2006) emphasized machine learning applied to extracting information from community metadata. They mined data from the freeDB database in order to correlate it with AllMusic data for the purpose of classifying music by genre. Neural networks were used to predict "canonical" AllMusic genre labels from uncurated FreeDB genre labels.

Pampalk and Goto (2007) built an artist recommendation visual interface based on community metadata extracted from the web. Google was used to search the web based on artist names in a way similar to that used by Whitman and Lawrence (2002). Pampalk and Goto, however, parsed the resulting web pages based on four different vocabularies: genres/styles, instruments/types, moods/adjectives and countries/regions. Words were selected to summarize each artist by selecting words that both occurred frequently and were statistically well suited to distinguishing sets of artists from other sets of artists. Similarity calculations were based on both such community metadata as well as audio features.

There has been a recent research trend towards using tags that have been assigned by users to artists or recordings, particularly since sites such as Last.FM have begun to make such tags available via web services. Geleijnse, Schedl and Knees (2007), for example, found experimentally that tag-based genre classification performed comparably with search-engine based classifications of types listed above, and found it to hold significant potential for other tasks. They also proposed a normalization method to remove meaningless tags. Eck, Bertin-Mahieux and Lamere (2007) have proposed the novel idea of using supervised learning on audio features to auto-tag music, or improve existing tags, and then use these tags for other classification tasks.

## *5.4 jWebMiner's functionality*

### 5.4.1 Fundamental functionality

At its most basic level, jWebMiner operates by accessing Google and/or Yahoo! via web services to acquire hit counts indicating the number of pages containing one or more search strings. These hit counts are then processed using a choice of user-selectable scoring metrics in order to generate the feature values output by jWebMiner.

jWebMiner allows two fundamental types of feature extractions to be performed: co-occurrence and cross tabulation. As discussed in Section 5.2.4, the term *co-occurrence analysis* is used here to refer specifically to the occurrence of two strings from the same set of strings on the same web page, and *cross tabulation analysis* is used to refer specifically to the occurrence of two strings from different sets of strings on the same page. So, for example, examining how often different artists co-occur on the same page would correspond to co-occurrence, and measuring how often artist names co-occur with genre names would correspond to cross tabulation. In general, co-occurrence analysis is useful for similarity tasks and cross tabulation analysis is useful for classification tasks, although queries can certainly be formulated for different purposes.

Research has indicated (e.g., Geleijnse and Korst 2006b; Schedl et al. 2006) that the best choice of procedures for obtaining and processing hit counts can vary. One must consider not only the accuracy of an approach, but also its search complexity, as web services are sometimes slow and typically impose daily limits on queries. jWebMiner therefore allows users to choose among a variety of approaches for formulating queries and generating features from their results, as described in the following sections.

jWebMiner has also been designed to be as flexible as possible so that it can be used for as a wide variety of MIR applications as possible. In fact, although it certainly was designed with the particular needs of the MIR community in mind, there is nothing about it that limits its use to music research, as arbitrary search strings may be used.

### 5.4.2 Web services utilized

The user may choose to use either one or multiple web services to generate features. jWebMiner comes packaged with functionality for accessing either Google or Yahoo!. Although in theory one might imagine that the two search engines would give similar results, informal experimentation has shown that they can often generate significantly different hit count ratios.

It should be noted that, while jWebMiner comes hard coded with a Yahoo! Application Key, users must enter their own Google License Key in order to access Google services. This is because Yahoo!'s usage limit of 5000 queries per day is applied per IP address, whereas Google's limit of 1000 queries per day is applied per key, regardless of IP address.

Users also have the option of adding additional web services if they wish, and jWebMiner's architecture has been designed specifically to make this a simple matter to implement. As discussed in Section 5.7, adding web services is a particular emphasis for future work.

One problem with dependence on web services is that they sometimes fail to respond to some queries, or sometimes temporarily go off-line entirely. jWebMiner therefore includes automatic cyclic time out and retry algorithms to account for such service discrepancies. Users are also automatically asked how they would like to proceed if certain queries or services repeatedly continue to fail over an extended period of time. A two-tiered progress bar is also provided that includes estimates of extraction time based on past and current response times of web services.

### 5.4.3 Features extracted and statistical processing

In the case of co-occurrence feature extractions, jWebMiner gives the user the choice of using either Equation 5.6 or 5.11 to generate similarity scores for pairs of strings. In the case of cross tabulation extractions, Equations 5.8, 5.12 or 5.13 may be used, as well as this original scoring metric, $T_4$, which is a variation of Equation 5.8:

$$T_4(a,g) = \frac{C(a,g)}{1+\sum_{i \in G} C(a,i)} \qquad (5.14)$$

Here C indicates the number of web pages containing its parameter strings, *a* is a string from the first set of queries and *g* is a string (usually a class name) from the second set of queries, *G*.

Users also have the choice of normalizing the resulting scores in several ways:

- **Normalize across web services:** If multiple web services are used (e.g., both Yahoo! and Google), the hit counts of each can be balanced so that services that produce fewer hits overall are not underweighted in the final scores.

- **Normalize across web sites:** As discussed in Section 5.4.6, a user may choose to specify specific web sites to search, as well as or in addition to the web as a whole. This option causes the number of hits returned by each specified source to be scaled so that it is not underweighted compared to other sources that generate more hits overall. This is separate from and does not affect manual source weights that the user can specify, as discussed in Section 5.4.6.

- **Normalize feature settings:** Final feature scores may be normalized to fall between 0 and 1, either over all classes per instance or over the whole data set.

The similarity scores output after all processing are the features output by jWebMiner. These may either be used directly as similarity metrics or class membership metrics, or they may be fed to machine learning systems such as ACE to generate more sophisticated mappings.

### 5.4.4 Synonyms

jWebMiner allows users to define *synonyms* between different search strings so that their hit counts will be combined during feature value calculations. For example, the genre class names *R and B* and *RnB* might be usefully combined. Synonyms can be important in avoiding falsely diminished hit counts for some entities because some web pages only refer to the entity in question using a word of equivalent meaning.

### 5.4.5 Filter strings

As discussed in Sections 5.2 and 5.3, one problem with algorithms based on co-occurrence analysis is that there is the danger of hit counts being biased by hits corresponding to non-relevant content. Simple searches on the band The Doors, for example, might be inflated by hits on carpentry or construction sites. Filter strings offer one way of reducing the effects of this kind of problem.

*Required filter strings* are the first kind of filter allowed by jWebMiner. These are strings that must be present on all web sites in addition to each specified query string or strings in order to be included in hit counts. The word *music*, for example, is one way of helping to ensure that counted hits are relevant to music. Other words, such as *review*, can perhaps increase the reliability and pertinence of counted hits.

*Excluded filter strings,* in contrast, indicate strings that, if found on a web page, will disqualify that page from appearing in hit counts. These are useful for eliminating non-relevant hits. For example, one might wish to exclude sites containing the words *paradise* or *zen* in order to avoid false hits relating to the band Nirvana.

Users have the option of using *pattern-based* required filter strings if they wish, as an alternative to simple filter strings. Pattern-based strings allow users to specify phrases that contain wildcards that will change on each query based on the particular search strings submitted for that query. So, for example, if one is classifying songs based on the albums that they appear on, one might use a pattern-based filter such as *<song> is on the <album> album* where the <song> and <album> variables will be replaced be each song and album title in the set of strings for which features are being extracted.

Although they can be powerful tools, one must be careful when choosing filter strings in order to avoid biasing hit counts. One must carefully strike a balance between eliminating irrelevant hits and including as many relevant hits as possible. Too many filter words or filter words that are too limiting or biasing can have a negative impact on feature values.

### 5.4.6 Source selections and weightings

jWebMiner's default setting is to obtain hit counts for the web as a whole. However, users also have the option of specifying specific sites, such as AllMusic, to be searched separately so that hit counts will be collected for pages on those sites exclusively. This is a useful way of emphasizing sites known to be relevant and reliable in hit counts.

Multiple sites may be specified, and individual hit counts will be collected for them exclusively or in addition to results from the web as a whole. Users also have the option of applying varying weights to each source if they consider some sources to be more trustworthy than others.

### 5.4.7 Other configurable options

There are a variety of other options that influence the hit counts that feature values are based on:

- **Treat strings literally:** Whether all search queries should be literal searches (e.g., for the query *heavy metal*, sites must have the two words adjacent if they are to be considered a hit if the search is literal).
- **Perform search as OR instead of AND:** Whether pages need only contain at least one of the specified query words in order to result in a hit, or whether they must all be present.

- **Include non-matching similar hits:** Whether results returned by search queries may include hits that do not contain one or more of the specified query words but do contain terms very similar to them (e.g., alternative spellings).
- **Suppress similar hits:** Whether to suppress similar hits when reporting results. Similar in this context means either sites with identical titles and/or descriptions, or multiple hits from the same host.
- **Suppress adult content:** Whether to suppress hits that are classified as containing adult content by the search service in question.
- **Limit to language:** Sets the name of a language that pages must be in in order to be counted as hits.
- **Limit to country:** Sets the country that pages must be hosted in in order to be counted as hits.
- **Search from region:** Sets the name of a country where searches will be performed (e.g., Google Canada rather than Google France). Results are not necessarily limited to this country, however.
- **Limit to file type:** Sets a file format (e.g., PDF, Word document, HTML page, etc.) that documents must be in order to be counted as hits.

### 5.4.8 Input and output

Users may manually enter any desired strings, and jWebMiner can also save and load settings for search strings, filters, source sites and weightings as delimited text files. In the case of search strings, jWebMiner can also parse strings from iTunes XML, ACE XML or Weka ARFF files. In the case of iTunes XML files, users can choose which field to parse strings from (e.g., song title, artist, genre, etc.) and jWebMiner will load all strings for that field for all recordings in the iTunes file, remove duplicate string values and sort them alphabetically.

Feature values are tabulated and displayed to the user following extraction. jWebMiner can also present a range of additional information to users following feature extraction, including the precise queries submitted to each web service and the results of all intermediate calculations. This information can be very helpful when debugging new functionality that is added to jWebMiner.

The reports displayed by jWebMiner can be saved exactly as they appear in the GUI as HTML files to be easily read by humans. Final feature values can also be exported to ACE XML, Weka ARFF or delimited text files to facilitate post-processing by software such as ACE or Weka.

## 5.5 jWebMiner's interface

Like all of the jMIR components, jWebMiner has been designed to be easily usable by researchers with a wide range of technical skills, from users who want to simply use it out of the box to extract cultural features to developers who wish to add functionality or modify the code to meet their own research needs. jWebMiner therefore includes a simple and intuitive GUI interface as well as modular and well documented code. An extensive HTML manual (Figure 5.1) is also provided that includes installation instructions, a tutorial, an interface reference guide, development instructions and general tips and pointers.

The jWebMiner GUI is divided into six panels. The first, the *Search Words Panel*, allows users to specify whether they would like to perform co-occurrence or cross tabulation analyses. In the former case, users enter a set of search strings such as artist names on which to extract similarity scores (e.g., Figure 5.2). In the latter case, users enter two sets of search strings corresponding, for example, to artists that are to be classified by genre (e.g., Figure 5.3). Figure 5.2 also demonstrates how the <SYNONYM> keyword can be used to specify one or more synonyms if desired.

As mentioned in Section 5.4.8, results are presented to the user in HTML formatted reports. Figure 5.4 shows sample results from the search words corresponding to Figure 5.3. Full reports can include much more information beyond the final feature values shown here, depending on user settings.

The *Required Filter Words Panel* and *Excluded Filter Words Panel* allow users to enter lists of filter words. Figure 5.5 shows the *Required Filter Words Panel* and demonstrates how keywords can be used to access pattern-based filters.

The *Site Weightings Panel* allows users to specify whether the entire available network and/or specific sites should be searched. As shown in Figure 5.6, the <WHOLE_NETWORK> keyword is used to refer to the overall results returned by web services, and additional individual sites can also be specified. The <WEIGHT> keyword allows different weights to be assigned to different sources when feature values are calculated. All entered weights are automatically normalized and scaled. Leaving this panel blank causes only the whole available network to be searched, and omitting the <WEIGHT> keyword results in equal weightings for all specified sources.

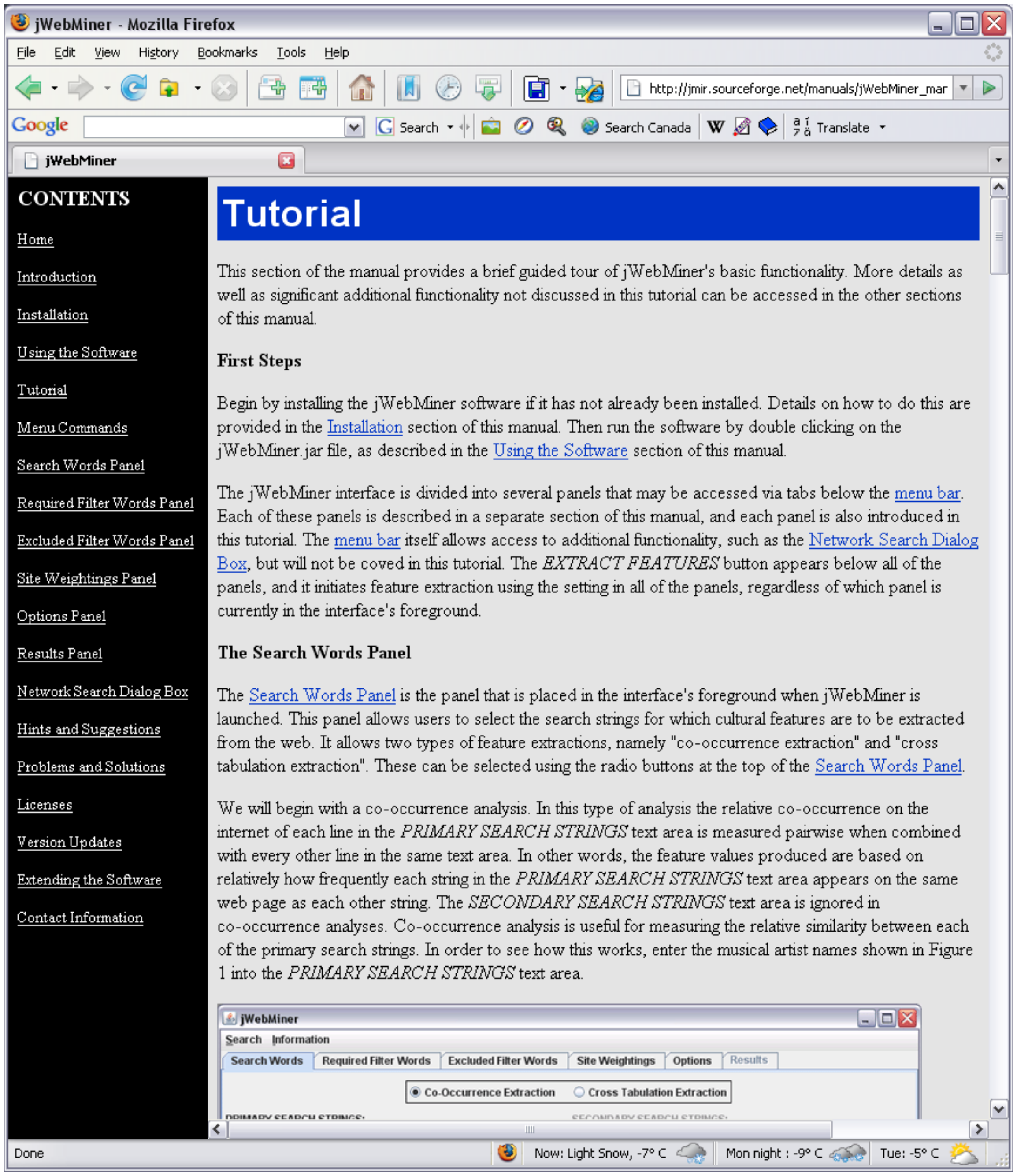
The *Options Panel* (Figure 5.7) allows additional options to be selected. These include which web searches are to be used, special instructions to submit to search engines, details of the scoring functions used and what information is to be included in final reports.

jWebMiner is also designed to encourage both experimentation with different extraction parameterizations and functionality expansion by the MIR community. It can be useful in both of these cases to see specific search results in addition to hit counts themselves. jWebMiner therefore includes a dialog box allowing users to perform test queries and see search results for different web services side by side (Figure 5.8).

## 5.6 Summary of original contributions

jWebMiner is, to the best of the author's knowledge, the first and only out-of-the-box cultural feature extractor designed for general-purpose MIR research. It has the advantages of having a well-documented GUI interface that makes it accessible to researchers of all technical backgrounds.

jWebMiner also offers the only polished and well-documented open-source MIR cultural feature extraction code base. The software is especially designed with an easily extensible architecture intended to encourage collaborative development within the MIR research community. jWebMiner also includes helpful debugging functionality.


jWebMiner - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://jmir.sourceforge.net/manuals/jWebMiner_mar
Google Search Search Canada Translate
jWebMiner

CONTENTS

Tutorial

This section of the manual provides a brief guided tour of jWebMiner's basic functionality. More details as well as significant additional functionality not discussed in this tutorial can be accessed in the other sections of this manual.

**First Steps**

Begin by installing the jWebMiner software if it has not already been installed. Details on how to do this are provided in the Installation section of this manual. Then run the software by double clicking on the jWebMiner.jar file, as described in the Using the Software section of this manual.

The jWebMiner interface is divided into several panels that may be accessed via tabs below the menu bar. Each of these panels is described in a separate section of this manual, and each panel is also introduced in this tutorial. The menu bar itself allows access to additional functionality, such as the Network Search Dialog Box, but will not be coved in this tutorial. The *EXTRACT FEATURES* button appears below all of the panels, and it initiates feature extraction using the setting in all of the panels, regardless of which panel is currently in the interface's foreground.

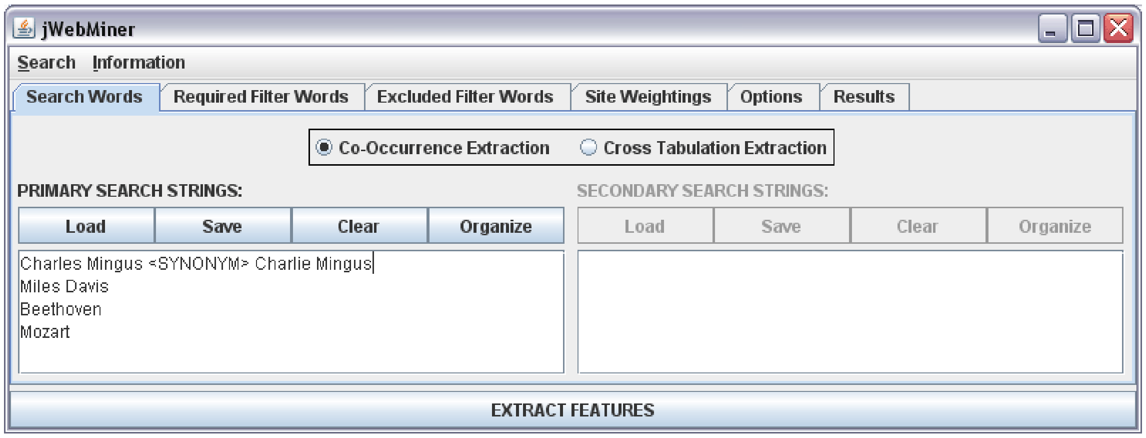
**The Search Words Panel**

The Search Words Panel is the panel that is placed in the interface's foreground when jWebMiner is launched. This panel allows users to select the search strings for which cultural features are to be extracted from the web. It allows two types of feature extractions, namely "co-occurrence extraction" and "cross tabulation extraction". These can be selected using the radio buttons at the top of the Search Words Panel.

We will begin with a co-occurrence analysis. In this type of analysis the relative co-occurrence on the internet of each line in the *PRIMARY SEARCH STRINGS* text area is measured pairwise when combined with every other line in the same text area. In other words, the feature values produced are based on relatively how frequently each string in the *PRIMARY SEARCH STRINGS* text area appears on the same web page as each other string. The *SECONDARY SEARCH STRINGS* text area is ignored in co-occurrence analyses. Co-occurrence analysis is useful for measuring the relative similarity between each of the primary search strings. In order to see how this works, enter the musical artist names shown in Figure 1 into the *PRIMARY SEARCH STRINGS* text area.

jWebMiner
Search Information
Search Words | Required Filter Words | Excluded Filter Words | Site Weightings | Options | Results
Co-Occurrence Extraction
Cross Tabulation Extraction
Done
Now: Light Snow, -7° C
Mon night : -9° C
Tue: -5° C

**Figure 5.1:** jWebMiner's HTML manual. This manual can be viewed either within the jWebMiner interface or via a web browsing application.

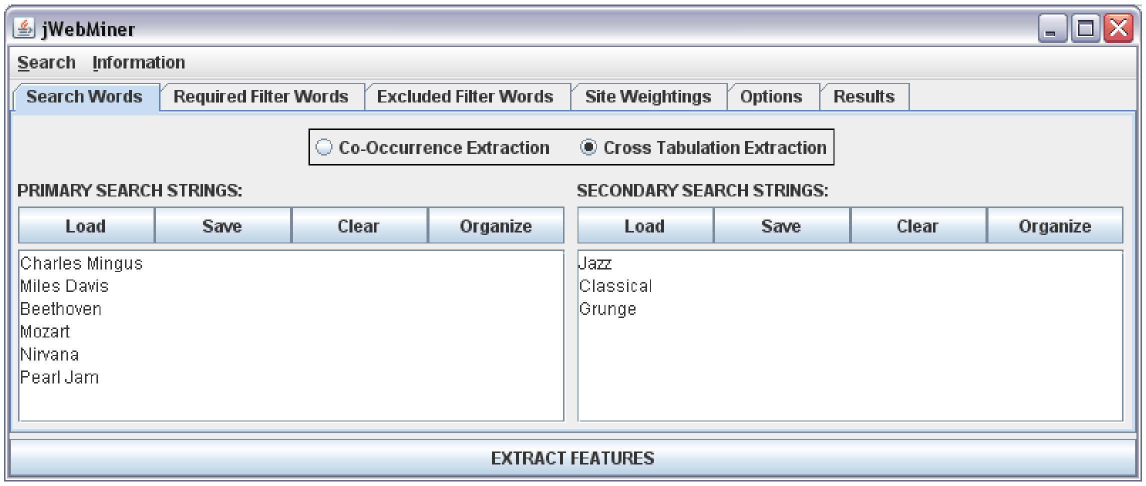


Co-Occurrence Extraction [✓] / Cross Tabulation Extraction

PRIMARY SEARCH STRINGS:

```
Charles Mingus <SYNONYM> Charlie Mingus
Miles Davis
Beethoven
Mozart
```

**Figure 5.2:** Query terms for a sample co-occurrence feature extraction that will measure the similarity between four musical artists. A synonym for *Charles Mingus* is included.



Co-Occurrence Extraction / Cross Tabulation Extraction [✓]

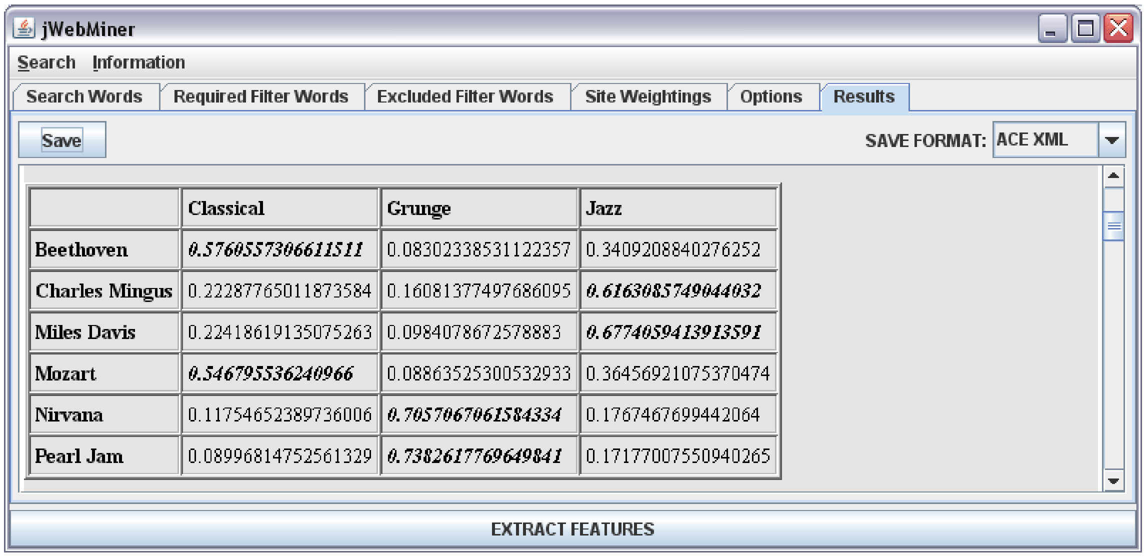
PRIMARY SEARCH STRINGS:

```
Charles Mingus
Miles Davis
Beethoven
Mozart
Nirvana
Pearl Jam
```

SECONDARY SEARCH STRINGS:

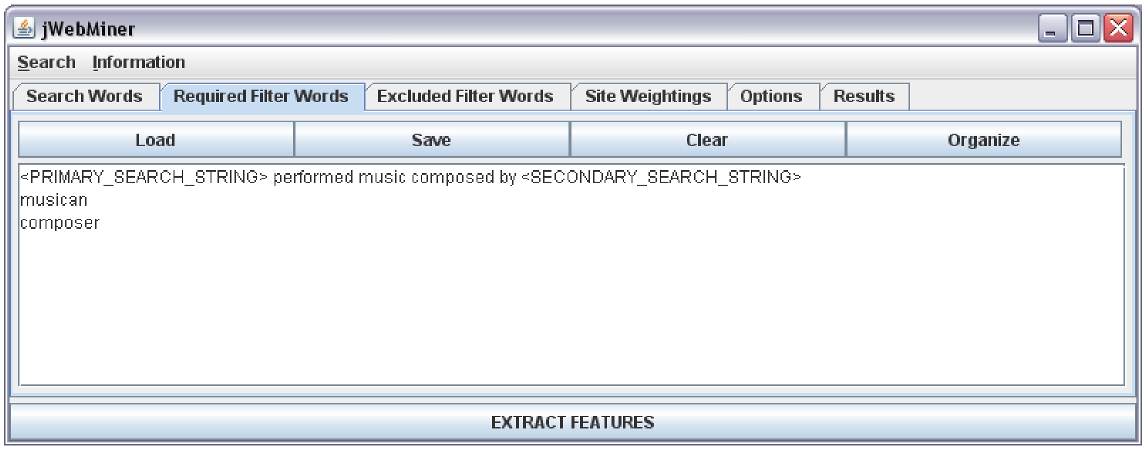
```
Jazz
Classical
Grunge
```

**Figure 5.3:** Query terms for a sample cross tabulation feature extraction that will measure class membership of six artists in three musical genres.



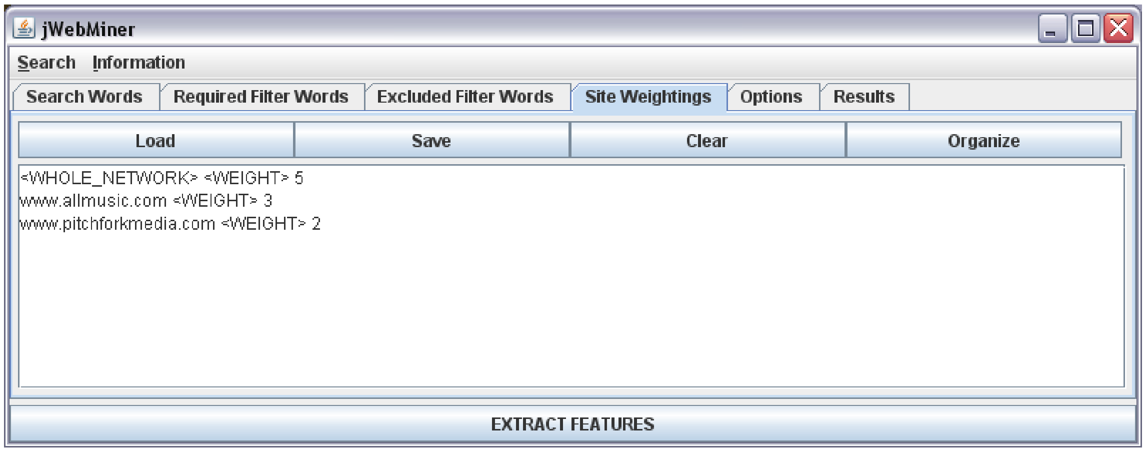
| | Classical | Grunge | Jazz |
|---|---|---|---|
| **Beethoven** | ***0.5760557306611511*** | 0.08302338531122357 | 0.3409208840276252 |
| **Charles Mingus** | 0.22287765011873584 | 0.16081377497686095 | ***0.6163085749044032*** |
| **Miles Davis** | 0.22418619135075263 | 0.0984078672578883 | ***0.6774059413913591*** |
| **Mozart** | ***0.546795536240966*** | 0.08863525300532933 | 0.36456921075370474 |
| **Nirvana** | 0.11754652389736006 | ***0.7057067061584334*** | 0.1767467699442064 |
| **Pearl Jam** | 0.08996814752561329 | ***0.7382617769649841*** | 0.17177007550940265 |

**Figure 5.4**: Sample results from the queries from Figure 5.3. The values represent the normalized weighted relative frequencies with which each artist name appears on the same web page with each genre name. The highest value for each artist is automatically bolded.



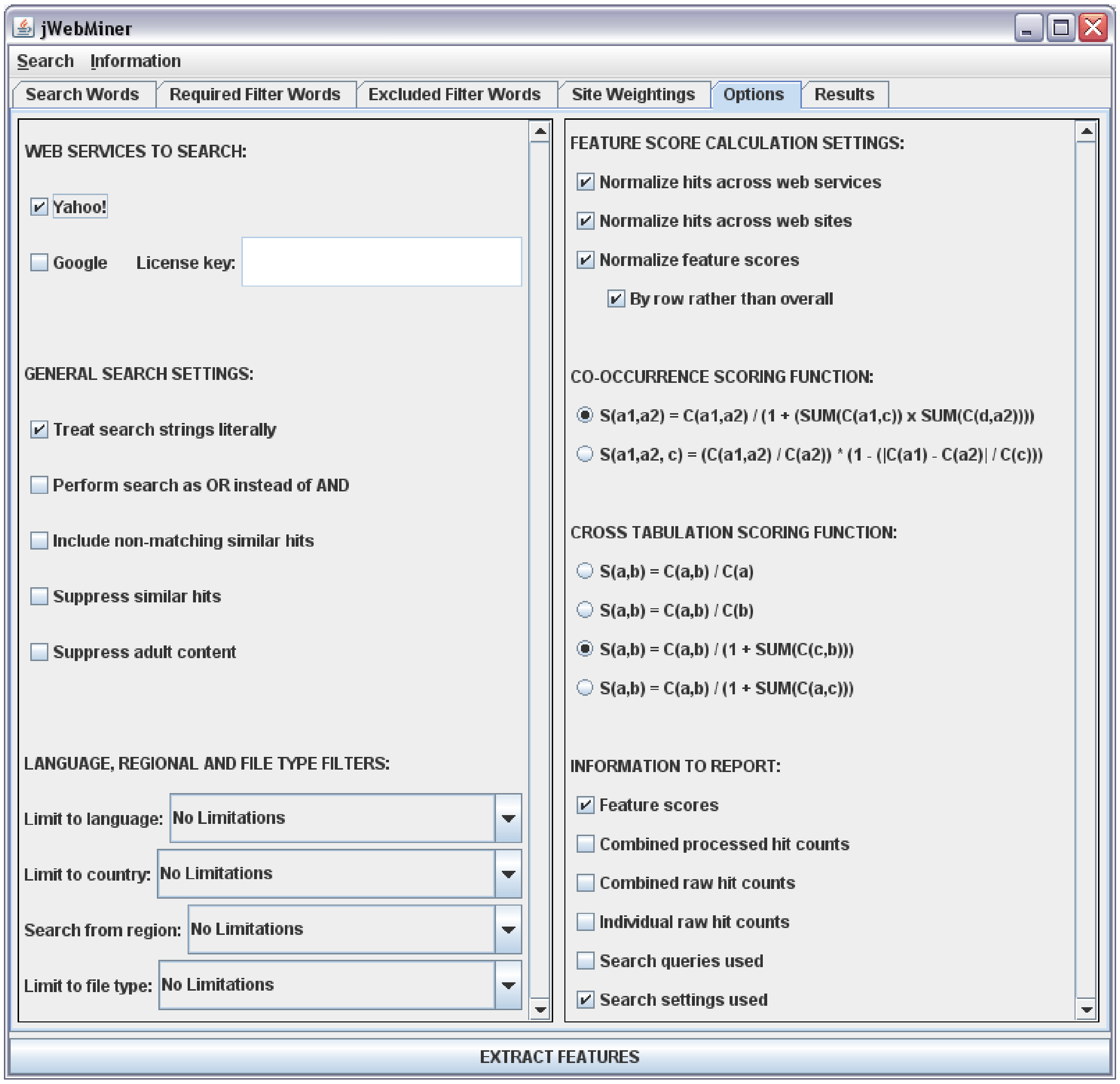
```
<PRIMARY_SEARCH_STRING> performed music composed by <SECONDARY_SEARCH_STRING>
musican
composer
```

**Figure 5.5:** The jWebMiner *Required Filter Words Panel* with one pattern-based filter string and two basic filter strings designed for measuring performer / composer relationships. The <PRIMARY_SEARCH_STRING> and <SECONDARY_SEARCH_STRING> keywords are wildcards that will be replaced in queries by search strings from the Search Words Panel.



```
<WHOLE_NETWORK> <WEIGHT> 5
www.allmusic.com <WEIGHT> 3
www.pitchforkmedia.com <WEIGHT> 2
```

**Figure 5.6:** The jWebMiner *Site Weightings Panel* with both the whole network and specifically the AllMusic and Pitchfork web sites selected to be individually searched separately. The results from the whole network will be assigned 50% of the weighting on the hit counts used to calculate features, AllMusic will be assigned 30% and Pitchfork will be assigned 20%.



WEB SERVICES TO SEARCH:

[✓ Yahoo!]

[ ] Google License key:

GENERAL SEARCH SETTINGS:

[✓ Treat search strings literally]

[ ] Perform search as OR instead of AND

[ ] Include non-matching similar hits

[ ] Suppress similar hits

[ ] Suppress adult content

LANGUAGE, REGIONAL AND FILE TYPE FILTERS:

| | |
|---|---|
| Limit to language: | No Limitations |
| Limit to country: | No Limitations |
| Search from region: | No Limitations |
| Limit to file type: | No Limitations |

FEATURE SCORE CALCULATION SETTINGS:

[✓ Normalize hits across web services]

[✓ Normalize hits across web sites]

[✓ Normalize feature scores]

[✓ By row rather than overall]

CO-OCCURRENCE SCORING FUNCTION:

[✓ S(a1,a2) = C(a1,a2) / (1 + (SUM(C(a1,c)) x SUM(C(d,a2))))]

( ) S(a1,a2, c) = (C(a1,a2) / C(a2)) * (1 - (|C(a1) - C(a2)| / C(c)))

CROSS TABULATION SCORING FUNCTION:

( ) S(a,b) = C(a,b) / C(a)

( ) S(a,b) = C(a,b) / C(b)

[✓ S(a,b) = C(a,b) / (1 + SUM(C(c,b)))]

( ) S(a,b) = C(a,b) / (1 + SUM(C(a,c)))

INFORMATION TO REPORT:

[✓ Feature scores]

[ ] Combined processed hit counts

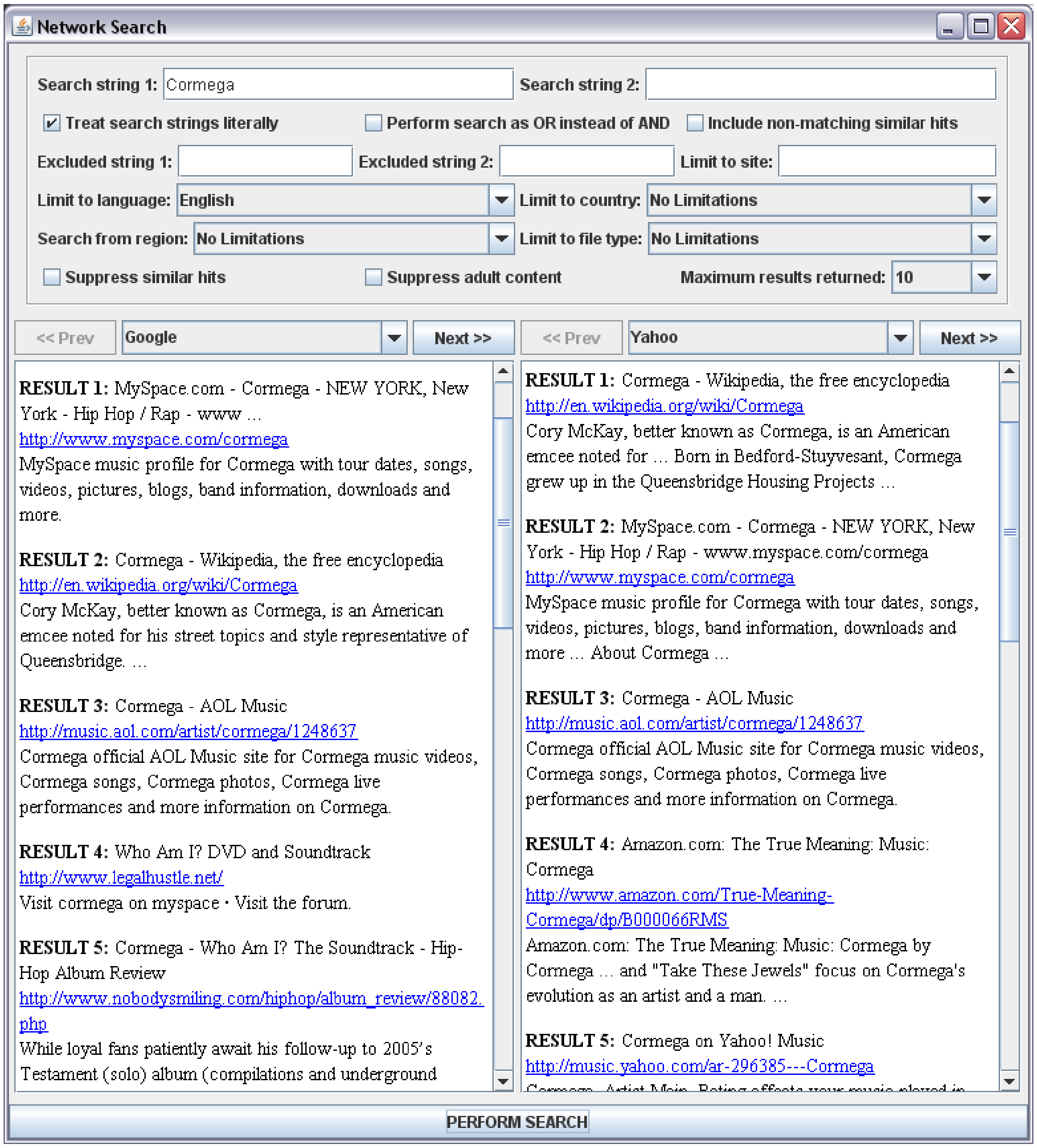
[ ] Combined raw hit counts

[ ] Individual raw hit counts

[ ] Search queries used

[✓ Search settings used]

**Figure 5.7:** Additional ways of customizing jWebMiner feature extractions offered by the *Options Panel*.



Search string 1: Cormega
Search string 2:

[✓] Treat search strings literally
☐ Perform search as OR instead of AND
☐ Include non-matching similar hits

Excluded string 1:
Excluded string 2:
Limit to site:

Limit to language: English
Limit to country: No Limitations

Search from region: No Limitations
Limit to file type: No Limitations

☐ Suppress similar hits
☐ Suppress adult content
Maximum results returned: 10

**RESULT 1:** MySpace.com - Cormega - NEW YORK, New York - Hip Hop / Rap - www ...
http://www.myspace.com/cormega
MySpace music profile for Cormega with tour dates, songs, videos, pictures, blogs, band information, downloads and more.

**RESULT 2:** Cormega - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Cormega
Cory McKay, better known as Cormega, is an American emcee noted for his street topics and style representative of Queensbridge. ...

**RESULT 3:** Cormega - AOL Music
http://music.aol.com/artist/cormega/1248637
Cormega official AOL Music site for Cormega music videos, Cormega songs, Cormega photos, Cormega live performances and more information on Cormega.

**RESULT 4:** Who Am I? DVD and Soundtrack
http://www.legalhustle.net/
Visit cormega on myspace · Visit the forum.

**RESULT 5:** Cormega - Who Am I? The Soundtrack - Hip-Hop Album Review
http://www.nobodysmiling.com/hiphop/album_review/88082.php
While loyal fans patiently await his follow-up to 2005's Testament (solo) album (compilations and underground

**RESULT 1:** Cormega - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Cormega
Cory McKay, better known as Cormega, is an American emcee noted for ... Born in Bedford-Stuyvesant, Cormega grew up in the Queensbridge Housing Projects ...

**RESULT 2:** MySpace.com - Cormega - NEW YORK, New York - Hip Hop / Rap - www.myspace.com/cormega
http://www.myspace.com/cormega
MySpace music profile for Cormega with tour dates, songs, videos, pictures, blogs, band information, downloads and more ... About Cormega ...

**RESULT 3:** Cormega - AOL Music
http://music.aol.com/artist/cormega/1248637
Cormega official AOL Music site for Cormega music videos, Cormega songs, Cormega photos, Cormega live performances and more information on Cormega.

**RESULT 4:** Amazon.com: The True Meaning: Music: Cormega
http://www.amazon.com/True-Meaning-Cormega/dp/B000066RMS
Amazon.com: The True Meaning: Music: Cormega by Cormega ... and "Take These Jewels" focus on Cormega's evolution as an artist and a man. ...

**RESULT 5:** Cormega on Yahoo! Music
http://music.yahoo.com/ar-296385---Cormega

**Figure 5.8:** The jWebMiner *Network Search Dialog Box,* which allows users to see results of test queries in addition to hit counts themselves. This is for the purposes of experimentation and debugging. A sample search for the artist *Cormega* is shown here.

## *5.7 Future research*

The flexibility of jWebMiner's interface encourages experimentation with various techniques and feature extraction parameterizations. It is still poorly understood which ones work best in various different circumstances, so an important emphasis of future research will involve comparing the relative efficacy of various different filter strings, synonyms, site weightings, scoring metrics and other configurations for a variety of tasks. Even small changes to such settings have shown the potential to dramatically affect results during informal experiments, and methodical experimentation could provide useful guidelines for using jWebMiner and other cultural feature extractors. An important caveat, however, is that the web is changing constantly, as are the algorithms used by each search engine, so a configuration that is superior one month may be inferior the next.

Another important priority is to design a simpler API for the jWebMiner code so that others can access it from their own software even more easily. Although jWebMiner is already open-source and was implemented using well-documented and modular code, the emphasis was on providing an effective GUI. It is hoped that an improved API will further encourage collaborative development of jWebMiner and of cultural feature extraction algorithms in general by the MIR community.

There are a number of excellent web services that remain to be taken advantage of by jWebMiner, such as those offered by Last.FM and Amazon. Yahoo! Music also offers specialized functionality that remains to be exploited by jWebMiner. These services could be incorporated into jWebMiner's existing search engine co-occurrence and cross tabulation framework, or they could be used to extract entirely different types of features. Building an API and GUI facilitating the development of specialized screen scrapers would also be a helpful addition, as there are a number of useful sites, such as AllMusic or Pandora, that do not make web services publicly available.

There are also a number of techniques described in Sections 5.3 that have yet to be implemented in jWebMiner. Examples include using the *define:* Google keyword to gain information on query synonyms, limiting hits to sites that include query strings specifically in page titles, wildcard phrase-based searching (e.g., as done by Geleijnse and Korst (2006b)) and an expanded number of feature metrics. It would also be useful to take greater advantage of search engines' spelling recommendation functionality to detect misspelled queries. jWebMiner also currently considers only page counts, not the number of occurrences of terms on each page. More sophisticated natural language processing could also be incorporated into jWebMiner, such as n-gram analysis and per-page string proximity measures. Specialized features with a narrower focus also deserve study, and there are many sources of cultural information available that warrant targeted investigation, as seen in Section 5.2.3.

Some important first steps have been taken on using text processing techniques to extract features from libretti / song lyrics, and there remains much to be done. Content (e.g., love, political messages, etc.), rhyming scheme, vocabulary, clichéd phrases, characteristic slang and other characteristics can all provide useful data. Although it is true, strictly speaking, that features extracted from lyrics are content-based rather than cultural, there are similarities in the text processing strategies that can be applied to both, and parallel research in the two areas could be mutually informative.

Still and video images such as album art, band home pages, promotional photographs or music videos can also yield significant amounts of useful cultural features. Simply looking at how three musicians are dressed, for example, can make it immediately possible to identify which ones play rap, classical and metal, for example. Meaningful information can be extracted by looking at a performer's appearance and actions on stage (facial expressions, ritual gestures, types of dancing, etc.) as well as from looking at an audience's dress and behaviour (clapping, shouting, sitting quietly, dancing, etc.). Automatically extracting musically useful features from such images is an area which researchers have only begun to explore.

# 6. ACE: Applying machine learning to music

## *6.1 Introduction to ACE and machine learning*

The previous three chapters introduced jAudio, jSymbolic and jWebMiner, three software applications designed to extract features from audio, symbolic and textual cultural musical data, respectively. The step that follows feature extraction in the automatic music classification workflow is to utilize methodologies that can automatically recognize patterns in feature values that are characteristic of the particular categories that one wishes to classify music into. This then makes it possible to have a computer use feature values extracted from music to automatically associate music with an appropriate category or categories.

This mapping between feature values and categories is the purview of *machine learning* and *pattern recognition,* which utilize a range of techniques that often include algorithms that train themselves on feature values, a process that is at least somewhat analogous to how humans learn from exemplars. *ACE (Autonomous Classification Engine)* is the jMIR component that applies machine learning to feature values in order to automatically learn to associate them with particular categories.

As discussed in Chapter 1, there are many important areas of MIR research in which automatic music classification plays an essential role, both academic and commercial. These include direct classification problems like genre, style, mood, composer or performer classification; similarity-related classification problems like music recommendation, playlist generation and hit prediction; and tasks associated with audio to symbolic transcription like onset detection, pitch tracking and instrument identification. Tools such as ACE can therefore have a very broad breadth of application in the MIR research community.

### 6.1.1 Why use machine learning?

Given that computers can process huge quantities of data much more quickly, cheaply and, potentially, consistently, than humans, it is clear that automatic music classification can have important advantages over manual human classification. However, it might be argued that it would be better to encode expert knowledge into music classification systems rather than to use machine learning. For example, a genre classifier designed to identify Blues music might be explicitly programmed to look for a swung rhythm, predominance of pentatonic scales, a twelve bar blues chord progression, an AAB structure and so on when. So, why use machine learning instead of such an approach?

Unfortunately, it can be very difficult to impossible in practice to manually encode such information into automatic classification systems in any kind of general sense. Musical classification problems often involve multiple competing and incompatible theoretical models. Even when there is general music theoretical agreement, the number of variables and considerations to be taken into account can have such a great scope that it can be very difficult to properly and sufficiently encode them into a heuristics-based classification system. In addition, expert humans can sometimes have unconscious biases associated with their individual theoretical predispositions, whereas data-driven machine learning is not biased by such potential prejudices.[143]

Machine learning, in contrast, requires minimal or no explicit human specification of a problem, or even understanding of it. This is because machine learning operates empirically on statistical patterns in the data that is being learned rather than relying on pre-existing theoretical models and heuristics, although some algorithms do allow these to be incorporated as well.

Machine learning-based music classification has the additional advantages that learned models can be automatically retrained when needed with minimal direct human involvement. This is also particularly advantageous when classifying music, which changes constantly. A genre classifier trained in the 1980's could easily be retrained to recognize Grunge when it arose in the 1990's, for example, just as a classifier trained in the early 1990's could be retrained in the early 2000's to recognize Emo. Music classifiers based on machine learning can thus not only process music much more quickly and cheaply than human classifiers, but can also be updated via retraining to keep abreast of the changing musical climate.

[143] The particular choice of features that are provided to machine learning algorithms can potentially introduce theoretical bias, however. If one assumes that only harmony and not rhythm, for example, is relevant to solving  a given classification problem, and as a result only extracts features related to harmony, then one is effectively introducing theoretical bias into the process by preventing the machine learning system from considering all potentially relevant information.

Yet another advantage of machine learning is that it can be taken advantage of by users to process types of music that they may know little about themselves. This can help facilitate large-scale global music processing systems that deal with a breadth of music that exceeds what any individual human could reasonably hope to become expert in.

### 6.1.2 Overview of ACE and meta learning

Machine learning and pattern recognition are subtle and sophisticated areas that require a high level of technical knowledge and experience in order to be exploited to their full potential. Choosing the best algorithm(s) to use for a particular application and effectively determining which values to use for the parameters of each algorithm are not tasks that can be optimally performed by researchers inexperienced in machine learning, and can be as much of an art as a science even for experienced experts. As a consequence, the unfortunate status quo in MIR research has often been to simply use only a few often arbitrarily chosen and parameterized classification algorithms, something that can result in significantly reduced performance.

As noted above, ACE is the jMIR component that performs machine learning. It is designed to increase classification success rates not only for machine learning experts, but also for users with little or no background in machine learning. ACE can be used to train new classification models on labelled data, to use previously learned models to classify novel musical (or other) data and to automatically evaluate the suitability of different machine learning configurations for a particular classification problem. There are a variety of machine learning strategies that may be applied to any given problem, as noted in Section 6.2, and ACE not only allows users to choose manually between them, but also provides users with the option of having ACE automatically perform experiments to automatically find which strategy appears to most suitable for the given problem based on empirical evidence.

This process of comparing and evaluating different machine learning algorithms with respect to a particular problem is the basis of a process called *meta learning.* Each machine learning algorithm has its own strengths and weaknesses, and a particular algorithm may be very well-suited to solving a certain classification problem, but less effective with respect to other problem domains. The experiment shown in Table 9.2, for example, as well as several of the other experiments described in Chapter 9, demonstrate empirically how the best classification algorithm for a problem can depend on the particular problem. This issue is also discussed theoretically in Section 6.2.

Not only can different algorithms vary in terms of classification accuracies, but they can also vary in consistency and in the time that it takes to train new models or to perform classifications. Another important difference is that some algorithms produce classification models that can be relatively easily examined and understood by humans once they have been learned, and other algorithms produce models that are essentially black boxes.

Since an essential goal of jMIR in general and ACE in particular is to make effective machine learning accessible to users with potentially no background in it, it is desirable to automate the process of choosing the algorithm to apply to a particular classification problem. ACE, as noted above, therefore uses meta learning to automatically perform experiments on a given dataset and its associated categories using a variety of different machine learning algorithms and parameterizations in order to automatically determine which are better suited to the problem at hand. Each algorithm is evaluated in a variety of ways so that users may choose one (or more) that is appropriate for their particular needs. For example, one might find that a given algorithm classifies data with a slightly higher accuracy than a second algorithm on average, but might still choose the second algorithm because it is more consistent or performs classifications more quickly.

It is important to reemphasize that meta learning, at least as it is implemented in ACE, is a purely empirical process, so users do not need any background whatsoever in machine learning in order to take advantage of it. This is essential in fulfilling jMIR's goal of making powerful machine learning technology available to researchers in fields such as the humanities or the library sciences who have very valuable musical knowledge but less background in machine learning. Of course, it may be true in some cases that a pattern recognition expert could recommend a specialized solution to a given problem that is as good or better than one found experimentally by ACE. ACE is not intended to replace such experts, but rather to automatically provide good solutions relatively quickly and effortlessly to users with diverse skill levels. If a user has a particular reason for using a particular algorithm, then ACE will certainly allow them to use it. Having noted this, even those researchers with a great deal of experience in pattern recognition often resort to experimentation in order to find good algorithms, and the meta learning approach used by ACE automates this process for them.

Of additional interest to expert users, ACE is utilizes the well-established Weka machine learning library (Witten and Frank 2005), which makes it relatively easy to develop new algorithms within the powerful Weka framework and then add them directly into the ACE meta learning framework. This also means that new algorithms implemented by the very active Weka community can be incorporated into ACE as they are released. Both ACE and Weka are implemented in Java, and both are entirely open-source and freely distributed.

Although ACE and the ACE XML file formats (see Chapter 7) are designed to meet the special needs of music, and include a number of important advantages with respect to music that are not available in alternative platforms such as Weka (see Section 6.3.2), there is nothing about ACE that prevents it from being applied to classification problems that do not involve music. ACE can in fact be used to apply meta learning to any application domain to which Weka could also be applied, and can read and write Weka's native ARFF file format in addition to ACE XML.

The original prototype version of ACE, ACE 1.0 (McKay and Fujinaga 2005a; McKay et al. 2005; McKay and Fujinaga 2007b), was originally developed in 2005, prior to the formulation of the jMIR project as an integrated whole. This version of ACE has since been used successfully in several experimental studies (Fiebrink, McKay and Fujinaga 2005; Sinyor et al. 2005; McKay and Fujinaga 2008). Although this original version 1.0 was found to be very effective, the need for improvements nonetheless became apparent, particularly with respect to the user interface and API structure. As a result, the improved ACE 2.0 (Thompson et al. 2009) is now available as an alpha release, although ACE 1.0 remains the stable release version at the time of this writing. However, given that much of ACE 2.0 is already complete, this chapter includes information relating to this updated version as well as ACE 1.0. The differences between the two versions are described in Section 6.6.

The ACE Java source code and associated bytecode are freely available at jmir.sourceforge.net/index_ACE.html. The distributions use and include the Xerces XML Parser[144] and the Weka Data Mining Software,[145] which are both also distributed for free.

### 6.1.3 Chapter outline

Although it is in no way necessary to have a background in machine learning in order to use ACE, a strong underlying understanding of machine learning is necessary to extend ACE, and it is of course always beneficial to have some such knowledge of machine learning if one is to apply it to problems. A relatively detailed background section on machine learning is therefore provided in Section 6.2. This includes a general overview of some of the fundamental procedures and problems of machine learning as well as a partial survey of some of the most commonly used machine learning algorithms, including those used by ACE.

Although ACE is currently the only meta learning software system designed specifically for music information retrieval research, there are some excellent more general machine learning environments available, as well as musical systems that do not include meta learning. Section 6.3 briefly overviews several frameworks of both kinds.

ACE itself is the subject of the remainder of this chapter. Section 6.4 describes the functionality offered by the software and itemizes the algorithms that it utilizes. The additional functionality that has and is currently being added to ACE as part of Version 2.0 is described in Section 6.6, and Section 6.5 provides an overview of ACE's user interfaces. Section 6.7 summarizes the original research contributions associated with the ACE component of jMIR, and Section 6.8 highlights long-term research goals.

## *6.2 Overview of machine learning*

### 6.2.1 Fundamentals of machine learning

*Machine learning* can be understood as referring to algorithms that allow computers to automatically construct abstract representations of problems by learning them from data of some kind. Machine learning can be used for many purposes, such as artificial

[144] xerces.apache.org
[145] www.cs.waikato.ac.nz/ml/weka/

intelligence or general regression problems, but from the perspective of automatic music classification it is used to learn mappings from musical data that can be used to classify the music. The particular algorithms that learn classification mappings are referred to as *classifiers* or *learners*.

The use of machine learning for the purpose of classification falls under the disciplinary umbrella of *pattern recognition,* or *discriminant analysis,* as statisticians refer to it. *Data mining,* or the application of machine learning to large data collections, is another strongly related discipline, although it is most often associated with a business context.

The individual entities classified by classifiers are variously referred to in the literature as *instances, data points, samples, observations, examples, exemplars* or *feature vectors*. Instances are typically only samples drawn from a population. The classification of an instance by a classifier into a particular class, or set of classes, is called a *prediction, classification* or *hypothesis*.

The categories that instances are classified into by classifiers are typically called *classes*. Abstract classes and the relationships between them can be organized into *class ontologies*.

As noted above, the process of classification, or labelling of individual instances with class names, typically first requires the extraction of *features* from instances. Just to review for the purpose of completeness, features are pieces of descriptive information that can be extracted from each instance and then used by classifiers to associate each instance with appropriate class labels. Features are also sometimes referred to as *attributes, inputs* or *variables,* and sets of features extracted from individual instances are sometimes referred to as *patterns* or as the *input representation.*

Features can be either *numeric* or *nominal.* Numeric features, also sometimes referred to as *continuous*, consist of numeric values such as integers or real numbers, and may thus have values corresponding to any one of the infinite number of numerical values available. Nominal features, also sometimes called *categorical,* must take one of a finite number of values, each of which is denoted with a name or symbol. Nominal features may be *ordinal features,* which is to say that they may be ranked but do not have precise inter-value distances (e.g., hot > warm > cold). Alternatively, some nominal features may not have any notion of ordering, or they may be *interval* features, which means that inter-value distances may be calculated more precisely. In practice, interval features are typically numeric rather than nominal, although this is not always the case.

The process of learning classification mappings from a dataset is called *training.* This training process optimizes the parameters of the classifier's internal representation of the mapping between feature values and class labels. Although the term *model* is sometimes used to refer to some particular classification algorithm (e.g., a multilayer perceptron[146]), it is more often used to refer to the particular mapping function learned by some given trained classifier. This latter use of the term *model* is the one that is used in this document. This learned model can be represented as:

$$y = g(x \mid \theta) \qquad (6.1)$$

where $y$ is the output class, $x$ is the feature set and $\theta$ is the set of model parameters (e.g., the architecture and weights of a multilayer perceptron). Learned models can be viewed in a variety of ways, one of which is to express them as of sets of *discriminants,* which is to say functions or rules that separate instances of different classes based on their features. In practice, it can be very difficult to find models that perfectly map all instances to appropriate classes. Those instances that do not obey the rules of the model are referred to as *outliers*.

A model may be *predictive,* which is to say that it can be used simply to predict unknown class labels from feature values, or it may be *descriptive,* meaning that it can be used to gain additional knowledge from the data, or both. The practice of acquiring simple rules from a model that explain the underlying nature and behaviour of the data is called *knowledge extraction.*

The particular data that is used to train and test classifiers is referred to as the *ground-truth.* This data is often labelled with model class labels that are assumed to be canonical. The choice of this data can have a very significant impact on the quality of the learned model, as will be discussed in the sub-sections below.

It is now useful to provide a musical genre classification example with the goal of more fully clarifying the terms described above. One would begin by constructing a ground-truth data set consisting of musical recordings associated with genre class labels

[146] See **Section 6.2.6**.

that are assumed to be reliable. Each of these recordings could be seen as an instance or, alternatively, each recording could be divided into analysis windows and each such window could be seen as an instance. One would than extract features from each ground-truth instance and give the features to the classifier to train on. The average tempo of a piece or the spectral flux of an analysis window are examples of possible features. The classifier would then generate a model by training on the features extracted from the ground-truth, and would then be able to use this model to predict the class labels of novel instances based on features extracted from them.

Although machine learning and automatic classification are described in some depth in the following sub-sections, many additional details are available in the vast machine learning literature. To give just a few of the many available resources, the books of Bishop (2007), Alpaydin (2004), Duda, Hart and Stork's (2001), Russell and Norvig's (2002) and Mitchell (1997) are excellent places to start. Jain et al. (1999) and Briscoe and Caelli (1996) offer excellent reviews of some of the earlier machine learning algorithms, many of which are still very effective. Hastie, Tibshirani, and Friedman (2001) provide a very good statistical perspective on machine learning.

Rowe's book (2001) is a good resource for those wishing to apply algorithms associated with artificial intelligence to musical tasks. Although this book touches on the actual techniques in less detail and much less rigorously than some of the other resources mentioned above, it is a very good introduction to artificial intelligence from a musical perspective and provides some excellent insights on how music can be conceptualized and represented in ways that computers can deal with effectively.

### 6.2.2 Automatic classification paradigms

There are primary overall paradigms associated with automatic classification: *expert systems, supervised learning* and *unsupervised learning.*

Expert systems use pre-defined heuristics to process features and arrive at class predictions. These heuristics are typically specified manually by humans based on pre-existing knowledge about the problem domain and, as such, expert systems do not typically use machine learning. Expert systems can be very effective for simple, easily defined and well understood problems, and can result in very high classification success rates by taking advantage of facts that are known to be true. Machine learning is much more suitable for more complex problems, however, since the manual formalization of many complex and interrelated heuristics can be very difficult and time consuming to implement properly. An additional major problem with expert systems is that models can be very difficult to modify so as to reflect changes in the problem domain.

As discussed in Section 6.1.1, expert systems tend to be a poor choice for problems related to music, with the exception of a few limited and specialized applications. This is because of the sophistication of musical problems and because pre-existing theoretical knowledge of precise heuristics that can be formalized into discriminants is often too sparse, inconsistent and contradictory for expert systems to be viably applied. This is readily apparent when one considers the range of popular, art and folk musics of the world and the variety of theoretical frameworks or lack thereof relating to each corpus.

These weaknesses of expert systems emphasize one of the key advantages of machine learning, which is the lack of a need to manually specify any details of the model to be learned or to have any *a priori* knowledge of the model at all.[147] This is because the learning algorithms construct models automatically. This can be useful not only in performing actual instance classifications, but also with respect to analyses of the learned models themselves, which can offer useful practical and theoretical insights into a problem domain.

Classifiers that utilize supervised learning attempt to formulate their own features to class mappings by using machine learning techniques to train on labelled ground-truth. Put more formally, they minimize the classification error by optimizing $\theta$ from Equation 6.1 as they learn from labelled training instances. The key point here is that the ground-truth instances are annotated with class labels, so that the supervised learning algorithms can learn by example. Once a model is learned, the classifier can use it to predict the class labels of unlabelled instances.

Supervised learning can be very useful in classifying instances into the particular classes that are of interest to users, but for it to be useful users must know precisely which classes they are interested in. Users must also have labelled ground-truth, which can be expensive to acquire in large quantities.

[147] Although some pre-existing knowledge can sometimes be useful with respect to certain machine learning algorithms, it is certainly not necessary.

Unsupervised learning, in contrast, does not require training data that is annotated with canonical class labels. This is because unsupervised learning groups instances into groups, or *clusters,* based on similarities and differences purely in the feature data itself, without reference to any pre-defined class labels. In statistics, this finding of a structure in the input space is referred to as *density estimation*.

Although each resulting instance cluster can certainly be interpreted as a class, there is no guarantee that these self-learned classes will be useful or meaningful with respect to specific problem domains. For example, unsupervised learning is arguably poorly suited to musical genre classification. This is because, even the clusters that are produced will likely legitimately indicate similarity on some level, this similarity will likely not be of the kind that causes the clusters to be associated with the particular genre labels that humans use.

The main focus of this chapter is therefore on supervised rather than unsupervised learning, since supervised learning is typically much better suited to classification problems. This does not mean that unsupervised learning can never be useful with respect to music classification, however, as it can perform very well in classification problems where one is only interested in a few quite dissimilar classes. Unsupervised learning can also be very useful for problems where one is more interested in similarity in general than in pre-defined class groupings, such as automatic playlist generation or music recommendation.

The example of an imagined musicological research project involving the attribution of a set of historical pieces whose composers are unknown can be used to illustrate the relative merits of the three pattern recognition paradigms described above. An expert system would be appropriate if the stylistic practices of each candidate composer are well understood and can be easily formalized into heuristics (e.g., Figure 6.1). A supervised learning approach would be suitable if one has a number of additional pieces known to be by each of the candidate composers, as these could be used to train the system so that it could learn to automatically recognize the characteristics of each composer (e.g., Figure 6.2). Finally, an unsupervised approach would be suitable if one does not have a set of candidate composers, but would like to segment the music into groups that are likely to each correspond to a different composer (e.g., Figure 6.3).

```
if ( parallel_fifths == 0 &&
     landini_cadences == 0 )
   then composer → Palestrina
else composer → Machaut
```

**Figure 6.1:** Pseudocode for a simple expert system designed to distinguish between the music of Guillaume de Machaut and Giovanni Pierluigi da Palestrina. If a piece has neither parallel fifths nor Landini cadences, then the system concludes that it is by Palestrina. If there are either parallel fifths or Landini cadences, then the system concludes that the piece is by Machaut. In practice, of course, this heuristic is a dramatically oversimplified discriminator, and would likely perform quite poorly.

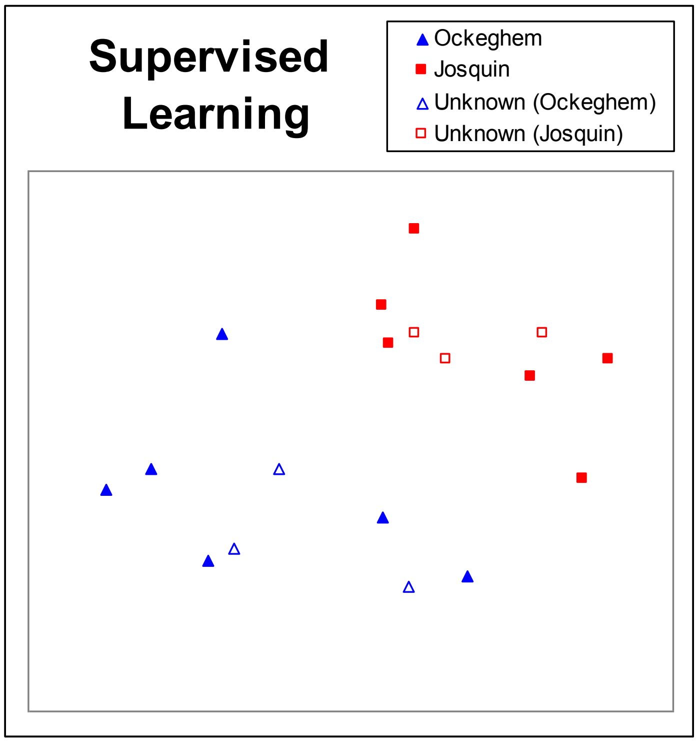


**Figure 6.2:** An example of supervised learning. In this case, many features are projected into two dimensions so that they can be more easily displayed. The problem is to teach the system to distinguish between compositions by Johannes Ockeghem (triangles) and Josquin Desprez (squares). The system is first trained by providing it with labelled compositions that are known to be by each composer (the filled in triangles and squares). The system is then given six unlabelled compositions (the empty triangles and squares). Based on the examples that the system was trained on, the system identifies three compositions as being by Ockeghem and three as being by Josquin. Note that, unlike the expert system in Figure 6.1, it is not necessary to explicitly specify any of the characteristics of the composers themselves when training the system, since the system learns these characteristics directly from the training examples.

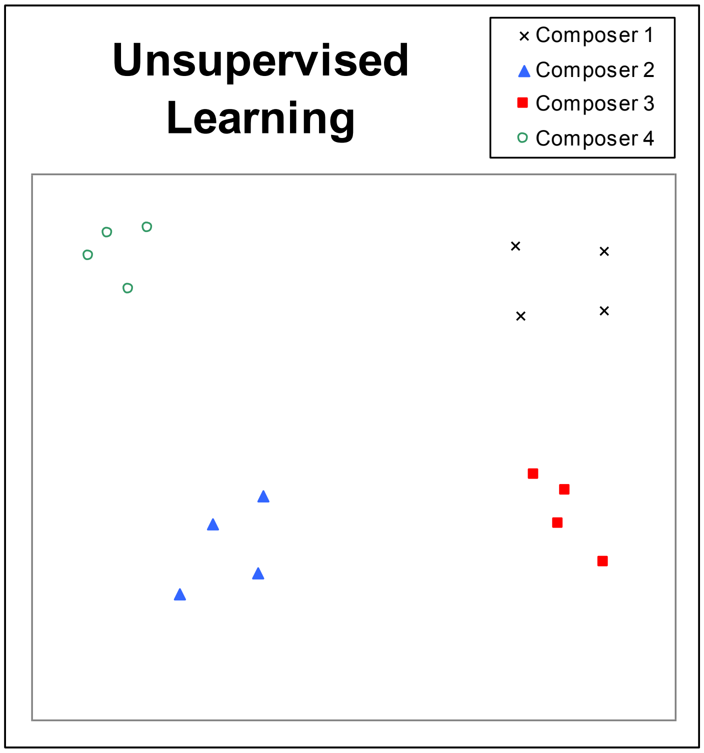


**Figure 6.3:** An example of unsupervised learning. As in Figure 6.2, many features are projected into two dimensions so that they can be more easily displayed. The problem is to teach the system to separate a body of sixteen anonymous pieces for which one has no reliable information at all about attribution. This means that supervised learning is not an option, because no labelled training samples are available. The unsupervised learning algorithm examines the sixteen pieces and separates them into four groups based on their relative differences and similarities, with each group hopefully corresponding to a different composer.

As a side note, there is another large class of machine learning called *reinforcement learning* that is used when one needs to assess the goodness of a sequence of actions. A single action is not necessarily important in itself, but a sequence of actions such as a sequence of moves in a chess game is. The reinforcement learning research community is largely distinct from the automatic classification community, and reinforcement learning problems are typically formulated in terms of an *agent* moving through an *environment* and learning a *policy* of actions to take as it goes that maximizes its *reward.* Having noted this, reinforcement learning could potentially be applied to certain sequential musical problems, such as automatic harmonic analysis of chord progressions. Reinforcement learning is not appropriate for most supervised learning oriented problems such as genre classification, however.

### 6.2.3 Classification error and common difficulties that must be overcome

The classification models that classifiers build during training are often imperfect, and will not necessarily correctly predict the classes of all of the instances that they are used to classify. This can be true not only of instances not included in the training set, but even of instances that are in fact in the training data. The proportion of the training instances that are incorrectly classified after a given training iteration is referred to as the *empirical error,* and this is what is typically minimized as training proceeds. The empirical error will ideally be zero after training is complete, although this will not always be the case. A particular classification algorithm may simply be inherently unable to correctly model a particular problem, or it may get caught in a local rather than global minimum in the error space even if it could theoretically model a correct mapping.

*Underfitting* is one of the important problems that one must account for in automatic classification. Underfitting occurs when the mapping function underlying the model is insufficiently complex compared to the function underlying the data. For example, a classifier that uses only linear discriminants will be unable to correctly model a problem where the discriminants are actually higher order. A non-zero empirical error after training is complete can be one indication of underfitting, although a zero empirical error is not necessarily sufficient proof that underfitting has been avoided.

Another important problem is related to the fact that training instances are typically only a sample drawn from a much larger population, and one wishes not only to train a model that performs well on the training data, but one that will perform well on any instance in the population. *Generalization* refers to the ability of a trained model to classify instances not in the training set, something that is of essential importance. A measure of generalization can be found by calculating the *generalization error,* which reflects the classification performance on instances in some *validation set* (also sometimes called a *test set*) that contains instances not in the training set. Evaluation methodologies for helping to ensure generalization are discussed in Section 6.2.4.

*Overfitting* is a problem associated with insufficient generalization. The risk of overfitting is increased when an overly complex model is used, such as one with high degree polynomial discriminants when the data's inherent model has simple linear discriminants. Although a complex model in such a scenario can of course sometimes simply model a less complex model without problem, there is the risk that the complex model will instead learn noise in the training data in order to minimize empirical error that does not generalize to the overall instance population and in fact increases generalization error.

Underfitting and overfitting are often associated with *bias error* and *variance error,* respectively.[148] A high bias error means that the model that a classifier has learned is insufficiently complex to properly discriminate between classes, and this learned model is unable to properly classify instances. A high variance error means that the learned model is too complex and has modeled noise in the training sample that does not generalize to the population as a whole. One indication that variance error is a problem can be models that change significantly when the training data is varied, although it can be beneficial under certain conditions to use unstable models in general (see Section 6.2.7).

Since one does not typically know the model underlying the instance population of interest, one does not necessarily have *a piori* knowledge of the best classifier complexity to choose. This makes it difficult to choose a model that is sufficiently complex to avoid underfitting but not so complex as to promote overfitting. One must therefore resort to intuition or experimentation, the latter of which can be automated via meta learning (see Section 6.2.10).

It is also important to note that the risk of overfitting the data when using a given classification algorithm decreases as the size of the training data increases. Unfortunately, large training sets are often expensive to acquire and annotate with class labels, and increasing the number of instances can also increase training and/or classification times significantly, depending on the classification algorithm used.

The particular choice of instance to be included in the ground-truth is also important, as one wishes to have typical exemplars in order to help avoid overfitting, but unusual instances should also be present to help avoid underfitting. Instances of all classes of interest must be present as well, ideally in roughly equal proportions, since some

[148] Formally speaking, bias and variance are associated specifically with parametric classifiers (see Section 6.2.6), as they are calculated using the parameters learned for a particular distribution. In practice, however, the two terms are also used informally with reference to classifiers of any kind.

classifiers may minimize error in complex problems by simply learning to never map feature patterns to classes for which only a few instances are proportionally present in the training data. A scenario where the training data does not contain sufficiently diverse instances for the model to correctly generalize is referred to as an *ill-posed problem*. This is not always easy to anticipate and avoid due to the potentially high cost of ground-truth and the lack of *a priori* knowledge about the population, however, so one must often make assumptions when choosing the training data, something that is called the *inductive bias.*

*Noise* refers to the problem of unwanted anomalies in training or validation data. Noise can be due to improperly labelled ground-truth (called *teacher noise*), improperly extracted or recorded feature values or features that are important to the data's inherent model but that have not been extracted (called *latent features* or *unobserved variables*).

Latent features are modeled as a random component by the classifier unless they are well correlated with other features that are in fact extracted. One can therefore be tempted to extract a wide variety of features from instances in order to guard against the presence of such unobserved variables. Unfortunately, too many features can in fact degrade classifier performance, a problem known as the *curse of dimensionality*. Increasing the size of a feature set increases the complexity of the model that must be learned, and thus increases the risks of overfitting or of never converging to a good solution, and can also be problematic with respect to the resources needed to extract, store and learn features. As a rule of thumb, the number of training instances needed to properly train a model tends to increase exponentially with the number of features.

As with other aspects of machine learning, the choice of the feature set to use can be as much of an art as it is a science. One needs a feature set that is not too small and not too big, without knowing ahead of time what the ideal size is for the particular problem at hand. The choice of the particular features to use is particularly important, as they must encapsulate sufficient information to train a model that sufficiently approximates the underlying model of the data. It is generally wise to try to avoid redundant features, and it can also be very useful to consult experts in the given problem domain who can provide insights that can be invaluable in choosing the features to use. *Dimensionality reduction* techniques can also be used to automatically reduce the size of excessively large feature sets, as described in Section 6.2.5.

### 6.2.4 Validating and comparing classification models

As noted above, a low classification error on the training data itself is not a sufficiently reliable indicator that the learned model is well generalized to the population from which the training sample was drawn. The problem of evaluating generalization is addressed by reserving part of the ground-truth for use in evaluating generalization. Such validation data must not be used to train the model in any way.

Although good performance on a validation set still does not prove generalization, it is certainly a better indicator than empirical error. A generalization error that is similar to the empirical error is a good sign that overfitting has been avoided, although it is not a guarantee. For example, if the ground-truth data was selected from the data population using a selective bias, then overfitting due to this bias will not be detected by validation data drawn from this same ground-truth.

Although the best ratio of training to validation instances can vary, 10% to 20% of ground-truth instances are typically reserved for validation. A greater number of total available instances usually necessitates a smaller proportion of instances reserved for model validation in order to achieve statistical significance in the validation.

It is also important to be able to statistically anticipate the expected error rates of different classification algorithms on a given problem in order to choose which approach to use. This is also necessary when experimentally choosing model hyperparameters to use for a particular algorithm, such as the value of *k* for a k-NN classifier (see Section 6.2.6.4). Both of these problems are examples of meta learning, as described in Sections 6.1.1 and 6.2.10.

There are several ways of calculating the expected error rate of a classifier so that it can be compared with those of other classifiers, as described below. It is important to first stress, however, that when comparing multiple models a third partition of the ground-truth should be reserved to evaluate the classifier that is finally chosen. This *publication* *set*[149] must consist entirely of instances that have not been used to train or validate any of the individual classification algorithms up until this final stage. This is necessary to help ensure that the choice of classifier is not overfitted, just as it is necessary to use a validation set to ensure that the parameters of a learned model are not overfitted.

It can also be important to go beyond simply ensuring that the instances chosen for training, validation and publication do not overlap directly. For example, a genre classification experiment that includes recordings by the same performer in both the training set and the validation set may arguable effectively be contaminating the validation set because of the particular similarity of recordings by the same performer. As a result, some music classification experiments have used *artist filters* to avoid such situations (Flexer 2007).

*Cross-validation* is one common method that is used to evaluate how well a classifier is expected to perform. This involves first randomly dividing the ground-truth instances into *k* equally sized parts. The classification algorithm is then used to train *k* models that are each validated once. The training and validation of each of the *k* models is called a *fold*. Each fold involves using one of the *k* data partitions for validation and the other *k-1* partitions for training. Each of the *k* partitions is thus used for validation exactly once, and none of the *k* trained models is ever validated on instances that were used to train it. An indication of bias error can be found by calculating the average error rate across folds, and variance error is indicated by calculating the standard deviation of the error rate across all folds.

The best choice of *k* depends on the amount of ground-truth that is available and on the processing resources needed by the classification algorithm. A large *k* means that many models need to be trained, each on relatively many instances, which can be computationally expensive for some algorithms. A small *k* means that the number of training instances available for each training run is reduced, something that can be problematic if too few ground-truth instances are available in each training partition to train proper models. In general, five to thirty-fold cross-validation is commonly used. A

[149] Publication sets are sometimes also referred to as *validation sets* or *test sets* in the literature, but these overloaded terms are avoided here because they are also used to refer to sets used to evaluate individual models after training.

small *k* is acceptable if ground-truth is plentiful and training times are an issue, and a large *k* is best when only a small amount of ground-truth is available.

*Leave-one-out* is an extreme version of cross-validation that is appropriate when one has only a small ground-truth dataset. If there are *n* instances available in the ground-truth then $k = n$. This means that *n* models must be trained, each of which is validated on only one instance.

*Bootstrapping* is another approach that can be used when only very small datasets are available, although there is much less data independence than one would ideally like. This approach samples *n* instances *with replacement* from datasets containing *n* instances, meaning that a given instance may be chosen more than once. This process is repeated in order to generate multiple subsets, which are then grouped randomly into training and validation pairs.

*5 x 2* is another variant of cross-validation that involves using training and validation sets of equal size. The instances are first randomly divided into two equally sized partitions. Two folds are then performed, the first using one of the two partitions for training and the other for validation. The roles of the partitions are then reversed in the second fold. This process is repeated four more times, each using a different random partitioning, for a total of ten folds. The disadvantage of this approach is that there is less independence between the ten trials than there is in traditional ten-fold cross-validation, for example, something that can make it statistically difficult to achieve good confidence levels with the 5x2 approach. The advantage of this approach is that it allows larger amounts of training data to be used in each fold than if traditional cross-validation were applied, which can be necessary if there is only a small amount of ground-truth data available.

It is usually best to make sure that each class is represented in comparable proportions in each cross-validation data subset. For example, if 20% of all note durations encountered in an optical music recognition problem are quarter notes, then the durations of roughly 20% of the notes in each and every training and validation set should also be quarter note durations. Cross-validation that maintains class proportions in this way is called *stratified*.

There are a number of alternative validation techniques that can be used to help choose an appropriate classification algorithm for a particular problem, including *regularization, structural risk minimization, minimum description length* and *Bayesian model selection* (Alpaydin 2004). These approaches all make prior assumptions about the model, however, so cross-validation is usually a better general-purpose approach, assuming a large enough ground-truth. The other approaches can be better when training data is sparse, however.

A *confusion matrix* can be used to illustrate the types of misclassifications made by a classifier. This is a square table whose rows indicate true classes and whose columns indicate the classifier's predicted classes. Each entry of the table indicates the fraction of instances belonging to the class of the corresponding row that were classified as belonging to the corresponding column's class. A perfect classifier would thus have non-zero entries only on the diagonal.

Although techniques such as cross-validation allow one to make comparisons between the performance of different classifiers, there is still no guarantee that one classifier with a higher average classification performance across folds will in fact perform better than a second classifier with a lower average performance. Statistical *hypothesis testing* and *significance testing* tools based on *interval estimation* (Kendall and Stuart 1973) can help to increase one's confidence that a given classifier really is better than another based on their relative cross-validation performances.

Interval estimation allows one to make the claim that, given a set of data, a certain hypothesis is correct within a certain error range with a certain amount of confidence. For example, one might be able to say that, based on a study performed on a sample of the population, the population has an average height between 160 cm and 170 cm with 95% confidence. This means that the actual average height of the population will fall within the calculated range 95% of the time. Such statistical hypothesis testing makes it possible to judge the likeliness that a given average classification accuracy across cross-validation folds, for example, will be within a certain range of the true performance of the classifier on the population.

There are a number of different hypothesis testing techniques that can be used, each with their own sets of assumptions and correspondingly varying amounts of appropriateness for different classification problems. *Student's paired t test* can be used to examine the results of *k* folds in order to, with a chosen confidence level, accept or reject the hypothesis that the true error rate of a classifier is at or below some value.

A *contingency table* may be constructed in order to indicate how well two classifiers perform on the same validation set after being trained on the same training set. Such a table specifies the number of instances correctly classified by both classifiers, the number incorrectly classified by both, the number correctly classified by one but incorrectly by the other and vice versa. *McNemar's test* makes it possible to use a contingency table to test, with a given confidence level, the hypothesis that the two classifiers have statistically the same error rate.

In practice, one often wishes to test more than two classifiers in order to find which is truly the most accurate. For example, consider the case where *l* candidate algorithms are each trained on *k* datasets, such that there are *k* trained classifiers for each of the *l* algorithms, and one wishes to test the *l* algorithms for statistically significant differences in performance. *Analysis of variance (ANOVA)* provides a means for doing this, by testing the hypothesis that the mean error rates of the *l* groups are equal. If this hypothesis is rejected, then the procedure may be repeated on different subsets of the *l* groups in order to find which have statistically significant differences in performance and which do not. Not all possible combinations need to be tested, since the algorithms can be ranked by mean error rates and only neighbours need to be tested. As a rule of thumb, if it is found that two algorithms are not significantly different in terms of error rate, then it is usually best to choose the one that is simpler (e.g., has fewer parameters) and/or faster.

Unfortunately, one can find many examples in the literature where significance testing is not used as rigorously as it could be. For example, the methods discussed above assume that the number of validation errors made by each classifier is binomial with an approximately normal distribution, assumptions that are not necessarily always valid. Fortunately, there are tests that do not make these assumptions. The *Wilcoxon signed-rank test* is a nonparametric alternative to Student's paired t-test that can be used when the population cannot be assumed to be normally distributed. There are also nonparametric alternatives to ANOVA, such as the *Kruskal-Wallis test* or the *Newman-Keuls test* (Motulsky 1999).

An additional issue is that most hypothesis tests test whether classifiers have the same error rates, not if the particular errors that they make are the same. In practice, different misclassifications can have varying costs, and it may be appropriate to use loss functions so that this may be taken into account. Unfortunately, hypothesis testing that takes such cost functions into account is much more difficult.

In addition, there are a variety of factors to consider when making the final decision on the classification algorithm or algorithms to apply to a given problem beyond bias and variance error. These include:

- Training time and space complexity of the algorithm.
- Classification time and space complexity of the algorithm.
- The number of features needed to train an effective model and how cheaply they can each be extracted.
- The number of ground-truth instances needed to train an effective model and how cheaply they can each be acquired and annotated.
- The amount of *a priori* knowledge one has about the population's instances and their features. For example, are features known to follow a normal probability distribution?
- The class ontology. How many classes are there? How similar are classes to one another? Is there a formal class structuring? May instances belong to more than one class?
- Ability of the algorithm to perform structured learning (see Section 6.2.8).
- Interpretability of the algorithm (i.e., whether or not knowledge can be extracted from the trained model in order to gain insights into the problem domain).
- Ability of the algorithm to perform cost-sensitive learning (i.e., whether or not it is possible to model situations where some misclassifications are more serious than others).

- Ability of the algorithm to improve learned models as additional data becomes available.
- Ease of implementation of the algorithm.

As a final point, it should be noted that classification accuracy is sometimes evaluated in terms of *precision* and *recall.* The precision associated with a class is the number of instances correctly classified as belonging to that class divided by the total number of instances labelled by the classifier with predicted class labels corresponding to that class (i.e. the number of true positives plus the number of false positives). The recall for a class, in contrast, is the number of instances correctly classified as belonging to that class divided by the total number of instances with that ground-truth class label (i.e. the number of true positives plus the number of false negatives). Precision can therefore be seen as a measure of classification fidelity and recall can be seen as a measure of classification completeness. Ideally, one would have values of 1 for both precision and recall, but in practice there can often be a trade-off between one and the other.

### 6.2.5 Dimensionality reduction

As discussed in Section 6.2.3, it is essential that features provide enough information to machine learning algorithms for them to be able to properly train models that can properly discriminate between classes. Since one typically does not have *a priori* knowledge of which features are needed to do this, one is tempted to simply extract as many features as possible as insurance against unobserved variables. The use of more features increases model complexity, however, which in turn increases time complexity, space complexity, the number of training instances needed and, overall, the risk of overfitting.

*Dimensionality reduction* techniques address this trade-off by taking a given feature space and automatically mapping it to a feature space with fewer dimensions. The ideal is to do this in a way that retains the most important information in the original feature space while only discarding information that is redundant or irrelevant to the particular classification problem at hand.

In addition to advantages with respect to classification performance, reducing the number of features can make it easier for humans to visualize and study feature data in order to understand it and gain insights about its underlying nature. It can be difficult for humans to visualize data in more than three to five dimensions (e.g., x, y, z, time and colour).

Dimensionality reduction techniques can be very effective in reducing the risk of overfitting due to too many input features, but they do not offer a guarantee that some useful information will not be discarded, although they do try to minimize the likelihood of this occurring. So, while these techniques are without a doubt often very useful, one should avoid making the mistake that they are always the best solution, or to reflexively apply a favourite dimensionality reduction technique without considering alternative dimensionality reduction techniques that might be more appropriate for the particular problem at hand. For example, one should not discard the option of consulting experts in particular problem domains for their views on which features will be useful in favour of simply applying off-the-shelf dimensionality reduction techniques.

Certain classification algorithms, such as decision trees, automatically do feature selection internally. Some, such as multilayer perceptrons, even do non-linear feature weighting internally. Dimensionality reduction pre-processing can still be useful even with such algorithms, however, as a greater number of features can still increase model complexity in such cases, thereby requiring a larger ground-truth to avoid overfitting. For example, multilayer perceptrons require a greater number of input units if there are a greater number of input features. Furthermore, depending on the algorithms in question, dimensionality reduction pre-processing can result in efficiency gains in terms of feature extraction, training and classifying costs.

One very simple, and typically unsuccessful, type of dimensionality reduction is to evaluate how well each feature can be used to classify instances individually, and then choose the *n* best performing features. Unfortunately, this approach fails to take into account the fact that the ways in which different features vary together can have powerful discriminating power. This approach can therefore result in the rejection of a very useful feature subset simply because a single feature in it that is very effective when combined with the other features in the subset does not perform well individually. A feature evaluation system must therefore consider the discriminating power of features operating collectively, not just individually.

Before proceeding to discuss better dimensionality reduction algorithms, it should first be emphasized that dimensionality reduction should be trained and validated with separate partitions of the ground-truth data, just as one does with classification algorithms. This helps to avoid overfitting the feature selection to the particular training data (Fiebrink and Fujinaga 2006).

It is also useful to note that are several ways to numerically estimate the usefulness of various feature sets. *Information gain,* for example, provides a way of calculating the increase in information that is gained by adding a new feature to a feature set via entropy calculations. Although information gain is typically associated specifically with decision trees, it may be generalized.

There are two primary overall approaches to dimensionality reduction. The first, *feature extraction,*[150] consists of deriving a smaller set of new features from the original features. The second overall approach, *feature selection,* involves simply choosing a subset of the original features and discarding the remainder.

The primary advantages of feature extraction over feature selection are that it is usually easier to implement and can produce results much more quickly. However, most feature extraction algorithms have the disadvantage that they operate entirely on the feature data itself without reference to the particular classes in question.[151] Most, but not all, feature extraction techniques tend to emphasise those features that have the greatest variability in general, which will not necessarily coincide with the particular variability that one needs when classifying instances into a particular class ontology.

An additional disadvantage of many feature extraction approaches is that they can still require the calculation of a large feature set in order to perform the mapping to the lower dimensional feature space, something that can be problematic if features are expensive to calculate. Furthermore, it is sometimes useful to retain information on how well features perform in their original form in order to gain theoretical knowledge about a problem. For example, one might be interested in gaining musicological and music theoretical insights

[150] This terminology can be confusing, as the overloaded term *feature extraction* is also used to refer to simply calculating the values of features from instances.
[151] Such techniques are, in other words, unsupervised approaches that ignore potentially valuable class labels even when it is available.

from how well different musically meaningful features can distinguish between different musical genres (McKay and Fujinaga 2005).

There are a variety of commonly used feature extraction algorithms, including *principle component analysis (PCA), linear discriminant analysis (LDA), factor analysis (FA)* and *multidimensional scaling (MDS)*. All of these methods are unsupervised, with the exception of LDA, and thus do not take class labels into account. Particular attention is given here to PCA because it is very often used in MIR and classification research in general.

The goal of PCA, as with other feature extraction dimensionality reduction algorithms, is to find a mapping from a $d$-dimensional feature space to a new $k$-dimensional space where $k<d$ such that one loses as little useful information as possible. PCA finds a linear combination of the original features that projects them onto a new feature space in a way that maximizes variance. Each dimension of the new feature space is called a *principle component,* and each principle component is chosen so as to have a maximum amount of spread of instances across its dimension. The principle components can then be ranked based on their variance and those principle components with the lowest variance can be dropped in order to arrive at a dimensionality reduction. It should be noted, however, that if the dimensions of the original feature space are not correlated, then PCA will not lead to any reduction in dimensionality.

It is important to note that if the variances of the original features vary considerably then this will affect the direction of the principal components more than feature correlations. In order to avoid this problem the data is often pre-processed so that each original dimension has a mean of 0 and a unit variance before applying PCA. This combination of pre-processing and PCA therefore essentially centers the instance features and then rotates the axes to line up with the directions of highest variance.

PCA is sensitive to outliers, as a few points distant from the center can have a large effect on the variances and thus the choice of principle components. It is therefore common to discard outliers before applying PCA. One simple method for doing this is to calculate the Mahalanobis distance of data points and to then discard the isolated data points.

Linear discriminant analysis also bears special mention because it is a supervised method. Given instances from various classes, LDA tries to find a projection that separates the classes from each other as much as possible within the new feature space. This is done by using Fisher's linear discriminant to keep the means of each class as far apart as possible. Fisher's linear discriminant is optimal if the classes are normally distributed, although it can still often be used effectively even if this is not the case.

A general problem with linear projection methods like PCA, FA, MS and LDA is they assume features interact in a linear manner when in fact features often interact in a nonlinear manner. One solution is to use *principle curves* to find curves, as opposed to lines, that pass through the "middle" of a group of instances. In general, however, such non-linear approaches are possible but difficult.

One final point to mention before proceeding to discuss feature selection techniques is that many classification approaches use Euclidian distances. However, strictly speaking, Euclidian distances are only appropriate if all features have the same variance and are not correlated. If this is not the case then a distance metric such as the Mahalanobis distance should be used instead. Alternatively, the Euclidean distance could still be used by the classifier, but suitable feature normalization should first be performed using a technique such as principle component analysis.

Feature selection typically involves iteratively training and testing a chosen classification algorithm with different subsets of the features available. The feature subset that is found to produce the best classification performance can then be selected as the subset to use.

A variant of feature selection, *feature weighting,* is performed by associating weights with each of the selected features in order to indicate their importance relative to one another. Simple feature selection is often preferred to feature weighting because it is simpler to perform and because many classification algorithms effectively perform feature weighting as part of their model training anyway. Processing complexity is a particular problem with feature weighting, as the search space grows with $\Theta(nd)$, where $d$ is the number of features and $n$ is the (potentially infinite) number of allowable values for each weight (Punch et al. 1993). Having noted this, feature weighting can still be useful in some cases, such as in certain specialized classifier ensembles (McKay 2004).

Ideally, one would like to exhaustively test every possible feature subset. The simplest type of feature selection, *exhaustive feature selection,* does exactly this. Although this can work for small feature sets and with classification algorithms that do not require long training times, it is otherwise not usually feasible. For a set of $d$ features there will be $2^d-1$ possible experiments, each involving the training and testing of a classification model. Although techniques such as branch-and-bound searching can be used to improve speed, exhaustive searches are often effectively intractable in practice.

Although exhaustive selection does ensure that the features selected will be optimal for the ground-truth data available and the classification algorithm chosen, it in no way guarantees that the selected features will be optimal for the overall instance population. The processing complexity of exhaustive searches therefore becomes particularly unattractive given this lack of assured generalization optimality. Fortunately, there are a variety of alternative feature selection techniques available that, while still not ensuring an optimal feature set, can arrive at solutions much more quickly than exhaustive searches.

*Forward selection* and *backward selection* are well-known approaches of this kind. Forward selection operates by starting with an empty feature set and adding features one by one. This process begins by first testing all possible feature subsets consisting of one feature only and choosing the one that best classifies the test data. Next, all possible feature subsets consisting of this feature and one other feature are tested, and the best performing pair is chosen. Features continue to be iteratively added one by one in this way until some maximum number of features are selected or until performance stops improving. Backward selection, in contrast, starts with the full set of features, and iteratively removes features one by one. Forward and backward selection have the same technical complexity, but forward selection will often be faster in practice because smaller feature subsets are usually faster to evaluate.

A problem with these two techniques is that there is no way to remove (or restore) a feature once it has been added (or taken away). This problem, called *nesting,* can be significant, since a feature that performs well early on in the feature selection process may actually not be one of the best features to choose. A technique called forward *plus l take away r* overcomes nesting by first applying forward selection *l* times and then applying backward selection *r* times. A variation of this technique, called *sequential floating selection,* dynamically assigns the values of *l* and *r* instead of fixing them (Pudil et al. 1994).

*Genetic algorithms,* or *GAs,* are another approach that can be used to search for a good feature subset. GAs can be used for a wide variety of purposes, including optimization problems where exhaustive searches are not practical. Siedlecki and Sklansky (1989) pioneered the use of genetic algorithms for feature selection, and they have been used successfully in the past with respect to music classification (e.g., Fujinaga 1996). There is some evidence that, in general, GAs perform feature selection better but slower than greedy search algorithms (Vafaie and Imam 1994). Additionally, Minaei-Bidgoli et al. (2004) have shown that GAs are also powerful tools for feature weighting in multiple classifier systems. Fiebrink, McKay and Fujinaga (2005) found that genetic algorithms applied to feature selection and weighting can result in improved performance in a specifically musical context, particularly when utilized with a parallel architecture.

GAs are inspired by the biological process of evolution. They make use of data structures called *chromosomes* that iteratively *breed* in order to *evolve* a (hopefully) good solution to a problem. Each chromosome consists of a bit string that encodes a potential solution to the problem that the GA is being used to solve. This bit string is somewhat analogous to the DNA of a biological chromosome, and is combined with the bit strings of other chromosomes when breeding occurs. Each bit string has a *fitness* associated with it that indicates how well its associated solution solves the problem at hand.

In the case of feature selection, each bit in the bit string can be used to represent a feature, with a value of 1 implying that the feature is to be used and a value of 0 implying that it is not to be used. Alternatively, each bit string can be used to encode segmented multi-bit words, each of which represents a numerical value indicating the weighting of a feature.

A GA begins with a population of many chromosomes whose bit strings are randomly generated. Reproduction, and hence the evolution of potential solutions, occurs through a process called *crossover*. Some fraction of the chromosomes, based on a GA parameter called the *crossover rate,* is selected for reproduction, and the remaining chromosomes are discarded. One way of selecting the chromosomes that will reproduce, called the *roulette method,* assigns a probability of selection for reproduction to each chromosome based directly on its fitness. An alternative, called *rank selection,* ranks the chromosomes based on fitness and then bases the probability of selection for crossover on this ranking. This latter approach prevents one or a few chromosomes with very high relative fitnesses from dominating early on, as this could lead to a local minimum in error space that is far from the global minimum.

The actual process of crossover most often involves taking two of the chromosomes selected for breeding and breaking the bit string of each one into two or more parts at randomly selected locations. The resulting has with a bit string constructed from the pieces of the bit strings of its parents. An alternative approach involves going through the bit strings of the parents one bit at a time and copying the bits to the child when the values for the parents correspond and randomly selecting a bit's value when the corresponding bit of the two parents differs. Each parent chromosome can reproduce multiple times, either polygamously or monogamously.

There are several additional characteristics that are sometimes incorporated into GAs. *Mutation* involves assigning a probability that is usually very small to every bit of every child's bit string that the bit will be flipped. *Elitism* involves automatically cloning the chromosome with the highest fitness from one generation to the next. *Villages* or *islands* involve segregating the chromosomes into different groups that evolve independently for the most part, but can occasionally exchange a few chromosomes.

A simplified version of genetic algorithms, known as *random mutation hill climbing,* is also occasionally used. This involves eliminating the crossover step, and having the population improve through mutation only.

Genetic algorithms have been shown to find good solutions to many problems, although not necessary the optimal ones. Their success is highly dependant on the quality of the fitness function that is used. In the case of feature selection, this fitness function is related to how well the selected feature subsets perform in classifying test instances.

GAs can be very computationally expensive, particularly if fitness evaluation is expensive, since fitness must be calculated for each chromosome at each generation. This makes GAs suitable for classification algorithms that require little or no training time, but less appropriate for approaches that are computationally expensive to train.

There are a number of alternative feature selection techniques, such as *simulated annealing*. Jain and Zongker (1997) have written a good overview of experimental feature selection methods as well as an empirical study of their relative performance. Kirby (2001) offers a good resource for those looking for more specialized techniques than those discussed here.

As a final note, dimensionality reduction is just one among many types of pre-processing and post-processing that may be applied to data in the context of machine learning. Bruha (2001), for example, provides a good overview of such pre- and post-processing techniques.

### 6.2.6 Commonly used machine learning algorithms

There are a great variety of algorithms that implement machine learning in various ways, each with its own strengths and weaknesses. For example, Bayesian classifiers are efficient and can perform classifications with theoretical optimality, but require pre-existing statistical knowledge of the data that one is dealing with in order to do so. Nearest neighbour classifiers are simple and fast to train, but cannot infer sophisticated logical relationships between features. Multilayer perceptrons can model such relationships, but take a relatively long time to train. Tree induction algorithms are not always as effective as some other methods, but the models that they train are more easily interpretable by humans than the models trained by most other algorithms. Hidden Markov models can be good at modelling sequential data, but are less appropriate when dealing with independent feature sets.

These are just a few examples of the many differences between various algorithms. Further complicating matters, there are often different versions of each algorithm with different hyperparameters (e.g., the value of *k* in a k-nearest neighbour classifier). Different algorithms can also be combined into classifier ensembles (see Section 6.2.7), which can operate using a variety of different coordination algorithms, each of which have their own strengths and weaknesses.

This sub-section focuses on some of the most commonly used algorithms that have been shown to be successful in a variety of problem domains. There are certainly many other excellent algorithms available, however, and new ones are continually being developed. This sub-section also places a strong emphasis on supervised learning algorithms in particular, as they tend to be the best choice for most automatic music classification tasks.

The overview of classifiers presented here emphasizes an intuitive and practical discussion of the algorithms. Rigorous theoretical proofs and detailed mathematical descriptions of learning schemes are largely omitted, as they are documented at length elsewhere and the goal of this sub-section is primarily to emphasize the more applied aspects of automatic classification so that existing implementations may be more easily understood and used effectively.

Although there are many alternative ways that one might organize classification algorithms, the distinction between *parametric* and *nonparametric* classification methodologies is given particular emphasis in the literature. The parametric approach assumes that the training sample is drawn from a population whose feature values obey known statistical distributions, such as Gaussian distributions. Parametric algorithms essentially attempt to find the parameters of these distributions during training. So, in the case of a Gaussian distribution, training would involve inferring the mean and variance of the Gaussian from the training data. Nonparametric classifiers, in contrast, do not assume any particular statistical distributions, and local models are estimated based only on the training data.

#### 6.2.6.1 Overview of parametric methods

Parametric classifiers have the advantages of being efficient and of tending to perform very well when the features do in fact obey the assumed distribution. They can be particularly useful in helping to smooth noisy training data, particularly since many nonparametric methods have a tendency to generate overly complex models that risk overfitting noisy training data. The major disadvantage of the parametric approach is that the true distributions of feature values may not in fact match the assumed distributions. Although parametric classifiers can sometimes still achieve adequate classification success rates even in such cases, the success rates do tend to degrade significantly in such circumstances.

Bayesian decision theory is a common point of departure for discussing automatic classification algorithms. If $x$ is a feature pattern corresponding to some instance and $y$ is a class belonging to the class ontology *Y,* then one is interested in the following quantities:

- *P(y|x):* The probability that an instance belongs to class *y* given that it has a feature pattern *x.* This is referred to at the *posterior probability.*
- *P(y):* The probability that any given instance belongs to a given class *y,* regardless of its feature pattern. This is referred to as the *prior probability*. It is useful to note that, if one assumes that each instance may only belong to a single class, then for all classes $y_1$ through $y_n$ in a class ontology:

  $$1 = P(y_1) + \ldots + P(y_n) \tag{6.2}$$

- *P(x|y):* The probability that an instance has a feature pattern *x* given that it belongs to class y. This is referred to as the *likelihood.*
- *P(x):* The marginal probability than an instance will have the feature pattern *x,* regardless of the class that it belongs to. This is referred to as the *evidence.*

So, for example, if *P(y)* is the probability that any individual plays the guitar and *P(x)* is the probability that any person has calluses on their fingers, whether or not that person plays the guitar, then *P(y|x=callused)* is the probability that a person who is known to have callused fingers does in fact play the guitar.

These basic quantities are associated with *Bayes' Rule,* the fundamental equation of Bayesian decision theory*:*

$$P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)} = \frac{P(x \mid y)P(y)}{\sum_y P(x \mid y)P\,(y)} \tag{6.3}$$

An important qualification to Equation 6.3 is that the second equality is only valid if *x* and *y* are independent.

Bayes' Rule is very useful for classification. Given a feature pattern $x_i$ extracted from an instance *i,* one can use Bayes' Rule to calculate the posterior probability $P(y_j|x_i)$ for each class $y_j \varepsilon Y$. These posteriors in effect indicate the probabilities with which the instance *i* belongs to each class in *Y.* One can then simply classify the instance *i* into the class $y_j$ with the highest posterior probability, or one can treat the posteriors as class membership weightings. A classifier that use Bayes' Rule in such a manner is called a *Bayes' classifier.*

It is also possible to account for cases where certain types of misclassifications are more costly than others. For example, misclassifying a symphony by Mozart as being by Haydn is likely better than misclassifying it as being by Schoenberg. This can be done by calculating the *expected risk, R,* associated with classifying an instance as class $y_j$:

$$R(y_j \mid x) = \sum_{k=1}^{K} \lambda_{jk} P(y_k \mid x) \qquad (6.4)$$

Of course, this necessitates knowledge of $\lambda_{jk}$, which is some given measure of loss incurred by potentially misclassifying an instance as class $y_j$ when it actually belongs to class $y_k$. If $\lambda_{jk}$ is known, then one can calculate the expected risk for all classes $y \varepsilon Y$, and then classify an instance as belonging to the class $y$ with the lowest expected risk. There are also alternatives solutions to this type of problem, such as *utility functions.*

The greatest advantage of Bayes' classifiers is that their classification performance is optimal in theory. This is a claim that can be made for very few classifiers, since the norm for many classifiers is simply to hope that the learned model converges to a good local minimum in the error space, without any guarantee that it will in fact be the global minimum.

Unfortunately, in reality one is rarely able to actually achieve optimality with Bayes' classifiers, since optimality requires perfectly accurate knowledge of the prior, likelihood and evidence, which must in practice usually be estimated. Such estimations serve as the foundation of many parametric models, which use statistical distributions such as the Gaussian distribution to estimate these probabilities. Estimation techniques such as this do not work well when the instances belonging to each class do not each form coherent groups in feature space, however, since if instances of a single class are located in multiple discrete clusters then this implies multiple distributions (e.g., multiple Gaussians, each with their own mean and variance, or even multiple distributions of different kinds). Fortunately, this problem can be addressed with *mixture models,* which are described below.

Before doing so, however, it is useful to first discuss simpler cases. The Bernoulli distribution is often used to estimate Bayesian probabilities for two-class problems, and can be generalized to the multinomial distribution for problems involving more than two classes.

The Gaussian distribution is often used for modelling class-conditional probability densities. In practice, a multivariate normal probability density is often assumed, even when the true distribution may well not be normal. It is useful in the multivariate situation, which is to say when there are multiple features, to represent the instances as a matrix where each column is a feature and each row is an instance. One can then calculate the *mean vector,* which has elements consisting of the mean of each column (i.e., of each feature) across all known instances.

It can also be helpful when approaching classification problems to calculate how each feature varies compared to each other feature via a *covariance matrix*. The entries on the diagonal give the variance of each feature and all other entries give the covariance of each pair of features. This matrix can be used to calculate the correlation between features. If two features are independent then their covariance must be zero, although a covariance of zero does not necessarily mean that the features are independent.

The calculation of the mean vector and the covariance matrix are useful because they can be used as parameters of a multivariate distribution. Also, the assumption of a shared covariance matrix can be a very useful in simplifying problems. Although this assumption can increase bias, it also tends to reduce the variance of estimators. This assumption of a shared covariance matrix leads to linear discriminants, which is to say linear decision boundaries between classes in feature space, something that can be a convenient simplification but which should also be treated with caution, since the true discriminants may not be linear. *Linear discriminant analysis* and the related *Fisher's linear discriminant* (see Section 6.2.5)*,* for example, are techniques for finding the linear combination of features that best separate two or more classes.

*Bayesian networks,* which are also called *belief networks* or *probabilistic networks,* are visual models for representing the interactions between features. These networks are composed of acyclic graphs consisting of nodes and directed arcs connecting the nodes. Each node is associated with a random variable*, x,* and has a value corresponding to the probability of the random variable, *P(x)*. An arc from a node *X* to another node *Y* indicates that the node *Y* is influenced, although not necessarily caused, by *X*. Arcs are associated with the probability *P(y|x)*. The relevance of this to classification is that Bayesian networks may be converted to trees and belief propagation may be used for inference.

The naïve *Bayes' classifier* is a simplified variant of Bayes' classifier that ignores possible correlations among the input features. In other words, it is assumed that the absence or presence of any feature is unrelated to the absence or presence of any other feature. For example, a musical instrument might be (potentially incorrectly) classified as a member of the violin family because it has strings and is bowed. Even though the feature of being bowed depends on the feature of there being strings, a naïve Bayes' classifier considers these two features to contribute independently to the probability that the instrument is a member of the violin family. Keeping this assumption of the naïve Bayes' classifier in mind, a multi-feature problem can be reduced to a group of single feature problems, so the likelihood and posterior can respectively be expressed as:

$$p(x \mid y) = \prod_j p(x_j \mid y) \qquad (6.5)$$

$$p(y \mid x) = p(y) \prod_j p(x_j \mid y) \qquad (6.6)$$

Although the assumption of feature independence is often invalid, the naïve Bayes' algorithm can in fact nonetheless perform surprisingly well. The naïve Bayes' classifier is also relatively efficient, because the assumption of independence is equivalent to setting all off-diagonal entries of the covariance matrix to 0, so they do not need to be calculated.

As noted above, well-known statistical distributions are often used to estimate the Bayesian likelihood, estimate and prior so that Bayes' rule can be applied. In order to do this, one must first choose a statistical distribution and then find its parameters.

If the type of statistical distribution is known, then *maximum likelihood estimation* is a statistical method for fitting the distribution to the data and for finding estimates for the distribution's parameters. If the feature patterns *X* of all known instances *I* are likely to occur in a population with some probability density *p(x|θ),* where *θ* represents the distribution's parameters, then the process of maximum likelihood estimation attempts to find the *θ* that gives the highest *p(x|θ)* for *X*. In other words, maximum likelihood estimation picks the model parameters that make the given data more likely than any other parameters would make them.

### 6.2.6.2 Distribution estimation

Purely parametric approaches assume a single statistical distribution. *Semiparametric density estimation* techniques are also available for empirically estimating a mixture of distributions from the training data. Since one often encounters instances that are drawn from multiple sub-groups, the use of multiple distributions is often advisable. For example, a pitch classification system should be able to recognize a middle C played on a piano as well as a middle C played on a guitar. Semiparametric methods are typically applied by assuming a type of distribution, such as Gaussian, but not the number of co-existing distributions of this type, since one does not have *a priori* knowledge of how many sub-groups instances are drawn from or which instances are associated with which sub-groups.

Any of the range of available unsupervised clustering algorithms can be used to group instances into clusters as a pre-processing stage before applying parametric estimation to each of the clusters. It is also often useful in practice to apply dimensionality reduction to each of the clusters separately, before estimating parameters.

The *mixture density* of a mixture model with *k* distributions can be expressed as:

$$p(x) = \sum_{i=1}^{k} p(x \mid G_i) P(G_i) \qquad (6.7)$$

where the $G_i$ are the *k* mixture components and each *x* represents the feature pattern of a training instance. The mixture components are also called *groups* or *clusters*. The $p(x|G_i)$ are the *component densities* and $P(G_i)$ are the *mixture proportions*. The number of mixture components, *k,* and the instances, *x,* are the inputs, and the learner estimates the component densities and mixture proportions. If it is assumed that the component densities all obey a parametric model, then it is only necessary to find their parameters.

Although unsupervised clustering algorithms are outside the particular focus of this chapter, it is appropriate in the context of supervised parametric and semiparametric estimation to briefly explain one simple algorithm, *k-means clustering,* as an example of how clustering can be performed. In k-means, one first specifies the number of clusters being sought, *k,* after which *k* cluster centers are chosen in feature space. All instances are then assigned to their nearest cluster center, where "nearest" is determined by measuring the distance between the feature pattern of each instance and the cluster center. The mean center of each cluster is then calculated, and these centers are taken to be the new cluster centers. Instances are then each reassigned to the clusters whose centers are closest to each of them. This process continues iteratively until no more changes are made.

This algorithm is very sensitive to the initial placement of the *k* centers, and there are a number of ways of choosing this initial configuration. For example, the centers can correspond to the feature coordinates of *k* randomly selected instances, the centers can be placed random distances from the overall centre for all instances or the centers can be evenly spaced along the principal component. There are also a variety of ways to choose the *k* hyperparameter, such as the needs of an application, manual choice after plotting the data in two or three principal component dimensions or simply experimentation with different values.

The k-means algorithm is a special case of the *expectation-maximization algorithm* that assumes a Gaussian mixture and that assumes features are independent with equal and shared variances. This is why k-means uses hyper-circles to form clusters, while the more general expectation-maximization uses hyper-ellipses of arbitrary shapes and orientations. There are also many alternative clustering algorithms that are more sophisticated than k-means, such as *self-organizing maps* and *adaptive resonance theory.*

Despite the improvement offered by allowing for multiple distributions, semiparametric approaches still assume one or more particular distribution types, such as Gaussian. This is a potentially incorrect assumption that can lead to classifier bias. Fortunately, other estimators are available that estimate the distributions themselves empirically, although the curse of dimensionality causes them to be less effective when dealing with instances with many extracted features.

*Histogram estimators* offer one such approach. These divide the feature space into equally sized intervals called *bins* and associate magnitudes to each bin based on the number of instances in the training data that fall within its region in feature space. This results in a histogram that models the probability density.

The choice of bin width is an important hyperparameter to consider when using histogram estimators. Wide bins lead to smoother, less noisy probability density estimates, but can also lead to loss of detail. Conversely, narrow bins are undesirably sensitive to noisy data, but are better at capturing details in feature space. One solution to this tradeoff is to use *kernel estimators,* also called *Parzen windows,* which use overlapping kernel function, such as a Gaussian kernel, to smooth the input in order to permit relatively narrow bins. There are also methods for dynamically varying window widths intelligently (Gentle 2009).

#### 6.2.6.3 Overview of nonparametric methods

Histogram estimators are an example of a nonparametric approach. Although in the application described above the nonparametric algorithm is used as a kind of pre-processing for a parametric algorithm, nonparametric algorithms can certainly be used alone. This means that no assumptions are made about data distribution(s) and that classification models are trained purely empirically based on available data. Nonparametric algorithms can also be called *instance-based* or *memory-based* algorithms.

The fundamental advantage of nonparametric methods over parametric and semiparametric approaches is that they avoid model bias due to assumptions about probability distributions that may be invalid, although with the trade-off that the risk of overfitting may be higher. Also, parametric and semiparametric approaches have the advantage over nonparametric approaches of reducing classification problems to the estimation of only a small number of simple parameters. Nonparametric methods are therefore often less efficient than parametric methods. Although this does not apply to all algorithms, parametric methods are often $O(d)$ or $O(d^2)$ in computational complexity during classification, where $d$ is the number of parameters, and many nonparametric methods are $O(N)$, where $N$ is the number of training instances. Notably, usually $d^2 << N$.

This relative classification inefficiency is most notably true of those nonparametric algorithms that store training data and postpone model computation until they are required to perform a classification. Such algorithms are called *lazy learners,* as opposed to *eager learners* that compute models during training, such as parametric algorithms. The upside of lazy learners is that their training time is often much shorter than that of eager learners, although in practice one typically prefers fast classification over fast training.

### 6.2.6.4 Nearest neighbour classifiers

One of the most commonly used lazy learning algorithms is *k-nearest neighbour*, also called *k-NN*. Training is performed simply by storing the feature space coordinates of each training instance. A test instance may then be classified by calculating the distance between its feature space coordinates and the coordinates of the surrounding training instances that were memorized during training. The test instance is then associated with the class labels of the *k* nearest training instances. Finally, there are a number of ways of determining the final class or classes of the test instance, such as simply assigning the class that occurs most frequently among the *k* nearest neighbours or by performing a vote weighted by the feature space distances to each of the neighbours. If multiple classes may be associated with instances in the given problem domain then no such operations are necessary.

Considered slightly differently, k-NN classification involves growing a cell in feature space around a test point until the *k* nearest training points are captured. A variety of tools developed for computational geometry can therefore be used to improve classification performance. Of particular interest, *condensing* or *thinning* methods can be used to reduce classification complexity and memory requirements by reducing the number of stored training instances. These methods can also remove noisy outliers that could degrade classification accuracy. The idea is to select a subset of the training instances that lowest error on the testing data. *Gabriel thinning* is one particularly appropriate approach (Bhattacharya, Mukherjee and Toussaint 2005). Varieties of the nearest neighbour algorithm that use data thinning are called *condensed nearest neighbour* algorithms.

There are a variety of distance measures that can be used to calculate the distances between points in feature space. Euclidean distance is often used, although it is sometimes appropriate to use alternatives such as Manhattan distance, Tanimoto distance or tangent distance (Duda, Hart and Stork 2001). As with many other nonparametric algorithms, it is best to normalize feature values before storing them in order to ensure that all features have the same numerical range, otherwise features with large values may inappropriately dominate distance measurements relative to features with small values. Since Euclidean distance measurements assume uncorrelated inputs with equal variances, when this is not the case it can be advisable to use techniques that locally estimate optimal distance to separate classes, such as the *discriminant adaptive nearest neighbour*.

The choice of the *k* hyperparameter of k-NN is important, as a *k* that is too large risks inappropriately including points belonging to instances of non-corresponding classes in the selection region even if the training points are well clustered. A *k* that is too small, on the other hand, can make the classifier too sensitive to noisy training points. Higher values of *k* are more appropriate when the value of *n,* the number of training samples, is higher. One common rule of thumb is to set *k* to the square root of *n,* and the value of *k* is often chosen to be odd in order to avoid ties. When dealing with unstratified training data, which is to say when there are different numbers of training instances per class, it is appropriate to also consider the number of training instances belonging to each class, otherwise classes with too few training samples relative to *k* will be overwhelmed even if the training data is well clustered.

The *nearest neighbour classifier* is a simplified version of k-NN where *k* is set to 1. This approach divides the feature space into a *Voronoi tessellation*. It has been shown that, as the number of training instances approaches infinity, the misclassification rate of 1-NN is never worse than twice the Bayes' rate, which is to say the optimal rate (Duda, Hart and Stork 2001).

k-NN classifiers have a number of important advantages. They are very good at dealing with scenarios where instances belonging to a single class are distributed in feature space in several distant sub-clusters, as demonstrated in Figure 6.4. Their models are also easy to update even after they have been trained, since new training points can simply be added to the model. Also, the k-NN algorithm can be trained almost instantaneously since all it must do is memorize points (assuming no thinning has been applied).

The disadvantage of k-NN is that, like other lazy learners, classifications can be computationally expensive to perform. A naïve implementation of k-NN will have a classification complexity of $O(dn^2)$, where $d$ is the number of features and $n$ is the number of training instances. Fortunately, a variety of more computationally efficient k-NN implementations are available, although eager algorithms do still tend to perform classifications significantly more efficiently.

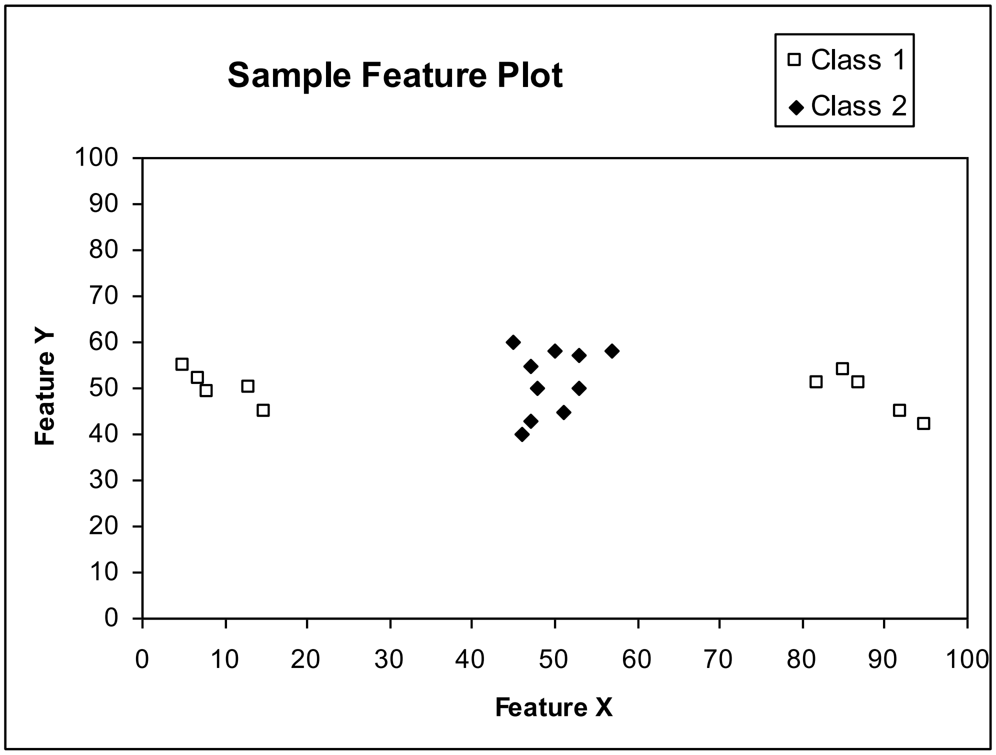


**Figure 6.4:** An artificial example demonstrating one type of scenario where k-NN classifiers are particularly advantageous. Here the instances belonging to Class 1 are divided into two different clusters separated with other instances belonging to Class 2, such that there is no single linear discriminant that could segment this feature space in such a way that all instances of Class 1 would be found on one side of it and all instances of Class 2 on the other side. k-NN classifiers, in contrast, would be unaffected by the fact that the Class 1 instances are in two different clusters, as long as *k* is small enough compared to the number of instances in each cluster.

k-NN classifiers are also limited with respect to the types of relationships between features that they can model. Classifications are based purely on distances, with no contingency for modelling conditional relationships between features. For example, consider the case of an experiment where one is attempting to classify musical recordings by genre, and the only feature that is available is the fraction of notes played by each different instrument. Ideally, a classifier will note from the training data that if even one note is played by an electric guitar then one can categorically eliminate all traditionally performed pre-20th century genres from contention. k-NN classifiers cannot learn this, however, as they are incapable of modelling such logical relationships between features. Consequently, the extensive use of a flute in a progressive rock recording also containing a very small amount of electric guitar might cause a k-NN classifier to falsely conclude that the piece is a classical flute sonata, for example.

#### 6.2.6.5 Feedforward neural networks

*Feedforward neural networks (FNNs)* are a type of nonparametric classifier than can model more sophisticated relationships between features than purely distance-based classifiers. They can also typically perform classifications more quickly than classifiers like k-NN, although training them usually takes much longer. FNNs are a member of a class of classifiers called *artificial neural networks* that are inspired by the organic brain, although they do not actually simulate neurological processes.

FNNs are composed of units called *nodes* that are also sometimes called *neurons*. These nodes are interconnected by unidirectional links, each with an associated weight. Learning in FNNs takes place by iteratively modifying the values of these weights.

The nodes of FNNs are organized into layers. The *input layer* is composed of nodes that are provided with input data directly. For the purposes of classification, each node in the input layer is typically given a value corresponding to a single feature value. There is also an *output layer,* whose nodes provide the output of the FNN in response to each input pattern placed on the input nodes. A typical approach to using networks for classification involves associating one output node with each possible class, although there are alternative approaches.

A FNN consisting of only an input layer and an output layer is called a *perceptron.* An important limitation of perceptrons is that they are only capable of learning linearly separable patterns. This means, for example, that they cannot learn a Boolean XOR function. In order to address this issue, one or more *hidden layers* of nodes may be added in between the input and output layers. The kind of FNN that results from doing this is called a *multilayer perceptron.* The *universal approximation theorem* for neural networks states that any continuous function that maps some interval of real numbers to some other interval of real numbers can be approximated arbitrarily closely by a multilayer perceptron with only one hidden layer, assuming an appropriate activation function such as the sigmoid function is used (see below).

Multilayer perceptrons are perhaps the most commonly used artificial neural network architecture. The presence of one or more hidden layers results in a classifier that essentially acts as a black box, since only the input and output layers are interacted with directly.

Each node in a FNN has a link to every node in the previous layer (if any) and to every node in the subsequent layer (if any). Each node also has an *activation level,* which is the value propagated down the links leading to nodes in the subsequent layer, or to the output sensors in the case of nodes in the output layer.

The activation level of a node is calculated by adding the values on all of the links coming into the node from nodes in the previous layer and processing the sum through an *activation function*. The value on each link is calculated by multiplying the link's weight and the value placed on the link by the node preceding it. Some FNNs also include a *bias unit,* or *bias node,* for each hidden or output node that outputs a constant value through a weighted link.

The activation function must be continuous, as its derivative is taken during training. A commonly used activation function is the sigmoid function:

$$sigmoid(x) = \frac{1}{1+e^{-x}} \tag{6.8}$$

As a side note, in practice it is usually best to scale model outputs to between 0.1 and 0.9 when using the sigmoid activation function, as using a range between 0 and 1 can overtax the network.

Stated more formally, the net input into an input node is simply the input value placed on it. The net input $net_j$ into a hidden or output node $j$ is given by:

$$net_j = w_b b + \sum_{i=1,n} w_{ij} o_i \tag{6.9}$$

where $w_b$ is the weight of the bias link, $b$ is the bias constant (0 if no bias node are used), each $i$ corresponds to a node in the preceding layer, $n$ is the number of nodes in the preceding layer, $w_{ij}$ is the weight on the link from node $i$ to node $j,$ and $o_i$ is the output of node $i$. The output of node $j$ with net input $net_j$ and a sigmoidal activation function is then:

$$o_j = \frac{1}{1+e^{-net_j}} \qquad (6.10)$$

FNNs are trained by putting training patterns on their inputs and observing how the outputs of the network differs from the model outputs. Weights are then iteratively adjusted in order to make the output closer to the model output using *gradient descent*. The specific process of first adjusting the weights leading into the output nodes and then successively adjusting the weights in each preceding layer is known as *backpropagation*.

The training modifications to the weights leading into an output node *k* are adjusted by first calculating $\delta_k$, the *error signal:*

$$\delta_k = (t_K - o_k) f'(net_k) \qquad (6.11)$$

where $t_k$ is the target or model activation value for the unit *k*, $o_k$ is the actual activation value, $net_k$ is the net input into *k,* and *f'* is the derivative of the activation function. For the sigmoid activation function:

$$f'(net_k) = o_k(1-o_k) \qquad (6.12)$$

The weights leading into output unit *k* from each unit *j* in the preceding layer, $w_{jk}$, are adjusted as follows from iteration *t* to iteration *t+1*:

$$w_{jk}(t+1) = w_{jk}(t) + \eta\delta_k o_j + \alpha\Delta w_{jk}(t) \qquad (6.13)$$

where

$$\Delta w_{jk}(t) = w_{jk}(t) - w_{jk}(t-1) \qquad (6.14)$$

and $\eta$ and $\alpha$ are neural network parameters known as the *learning rate* and the *momentum,* respectively. The momentum term is sometimes omitted. These hyperparameters control how quickly a neural net is likely to converge to a stable solution and how likely it is to get stuck in a poor local minimum in error space far from the global minimum. The learning rate controls how large the adjustments to the weights are during each training iteration (i.e., how large the steps are in error space towards a solution), and the momentum stops the network from ineffectually oscillating back and forth in error space from iteration to iteration by taking into account the previous adjustment to each weight. Increasing the learning rate can cause a network to converge faster, but a value that is too high may also cause the network to jump around so much that it does not converge. Increasing the momentum also usually increases the speed of convergence, but can cause poor classification performance if it is set too high. As rules of thumb, a value of 0.2 or less is often best for the learning rate, and values between 0.5 and 1 are usually best for the momentum. The initial values of the weights are randomly determined.

The error signal for a hidden unit, *j,* behind unit *k* in a subsequent layer is given by:

$$\delta_j = f'(net_j)\sum_k \delta_k w_{kj} \qquad (6.15)$$

The weight change formula used to modify the weights of the links pointing to a hidden unit is the same as that for the links pointing to an output unit (equation 6.14).

There are a variety of ways of going about the training process, all of which for the most part produce similar results. One can simply feed the training samples into the network one by one, and adjust the weights after each sample, a process called *online learning*. If one is inclined to be more mathematically orthodox, one can instead only actually implement the weight adjustments after all of the adjustments have been calculated for a particular training sample in its entirety, a process called *offline learning*. In practice, either approach can work well, but it is generally a good idea to randomly order the training samples if online learning is used in order to avoid receiving many similar patterns consecutively, which could be conducive to falling into a poor local minimum in error space.

The process of classifying the entire training set once and updating the weights based on the error signal is called an *epoch* or a *training iteration*. Many epochs are usually needed before the network converges to a good solution. An indication of convergence can be measured by observing the rate of change from epoch to epoch of the sum of squares error between the model output activation values and the actual values, *E*:

$$E = \frac{1}{2}\sum_k (t_k - o_k)^2 \qquad (6.16)$$

Convergence can generally be said to have occurred when this error stops changing significantly from epoch to epoch, although it is not unknown for a network to seemingly converge for a large number of epochs before suddenly starting to change significantly again.

There is some disagreement in the literature as to the best number of hidden layers to use and how many hidden nodes should be used in each. de Villiers and Barnard (1992) offer some convincing arguments that more than one hidden layer can actually degrade performance rather than improve it, for example. There have also been a number of competing formulas proposed as to the ideal number of hidden units to use. For example, Wanas et al. (1998) claim that the best performance, in terms of both performance and computation time, occurs when the number of hidden nodes is equal to *log(n)*, where *n* represents the number of training samples. There is no real consensus on this matter in the literature, however, and the optimal number of hidden nodes likely depends on the particularities of each individual problem.

A good experimental approach is to test performance by gradually adding hidden nodes one by one until the performance fails to improve by a certain amount. There are many variations of this approach, including Ash's pioneering work (1989). Another approach is to use techniques such as genetic algorithms to optimize network architecture. Alternatively, one can in some cases reflect perceived structure in the application domain in the network architecture. In any case, it is true in general that increasing the number of hidden units increases the complexity of the functions that can be modelled, but also increases the training time and, potentially, the probability that the network will not converge or will overfit the data. In practice, it can be helpful to pre-process features so that the input is normalized for factors such as size and rotation in order to simplify the job of the learner.

Feedforward neural networks are just one of the many varieties of artificial neural networks that exist, each with its own strengths and weaknesses. For example, *recurrent neural networks* include connections from the output nodes back to nodes in earlier layers, something that incorporates memory into the network so that present classification results can influence the classification of subsequent instances. This is useful in classifying data whose sequence is meaningful, such as chord progressions. *Kohonen self-organizing networks* perform unsupervised learning, to give another example. These are just a few of the many varieties of artificial neural networks that have been developed.

### 6.2.6.6 Discriminant-based classifiers: Linear classifiers, SVMs and decision trees

*Discriminant-based classifiers* are the last type of classifier that will be discussed in this sub-section. Such classifiers focus on finding the discriminants that separate instances from different classes by estimating the parameters of these discriminants directly. This differs from *likelihood-based classifiers,* such as many of the classifiers discussed above, which attempt to estimate the parameters of feature probability distributions instead of directly focusing on the discriminants themselves. One can therefore avoid needing to make potentially faulty assumptions about whether features are correlated or about what kind of statistical distributions the features follow when using discriminant-based classifiers.

The trade-off is that discriminant-based classifiers must often make potentially incorrect assumptions about the discriminants themselves. Linear discriminant-based classifiers, for example, assume that instances of a class are linearly separable from instances of other classes, something that is often not in fact true. In practice, however, good classification performance can still often, but not always, be attained even when the true discriminants are not in fact linear. Although approaches can certainly be used where higher-order discriminates are calculated (*quadratic classifiers* in particular have some limited popularity), the use of linear discriminants have several significant benefits: training and classification speed tend to be very high,[152] classification performance tends to be high even when there are many input features and the risk of overfitting is reduced. It is therefore often a good idea to try a linear discriminant-based classifier to see if it is sufficient before using more sophisticated techniques.

Put more formally, one can define the $i^{th}$ discriminant as $g_i(x|\Phi_i)$, where $x$ represents the input features and $\Phi_i$ represents the discriminant parameters. Learning is thus an estimation of $\Phi_i$. Each of a set of $d$ linear discriminants can then be written as:

$$g_i(\mathbf{x} \mid \mathbf{w}_i, w_{i0}) = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} \qquad (6.17)$$

where $w_i$ are the weights that must be learned.

[152] Efficiency of linear classifiers is *O(d),* where *d* is the number of features. Quadratic classifiers, in contrast, are $O(d^2)$.

Such sets of linear discriminants define hyperplanes in feature space. If all classes are linearly separable, then there must exist a hyperplane such that all instances belonging to any one class will be found on one side of the hyperplane and all instances belonging to all other classes will be found on the other side of the hyperplane. So, assuming once again that a problem is linearly separable, one need only find a discriminant defining such a hyperplane for each class in order to arrive at an optimal classifier.

If the classes are not in fact linearly separable, then one imperfect but often reasonably effective solution is to divide the problem into a set of linear sub-problems using a technique called *pairwise separation of classes*. This approach requires that a separate hyperplane discriminant be found for each pair of classes, which is to say that $K(K-1)/2$ linear discriminants must be found if there are $K$ classes..

An alternative approach for dealing with problems that are not linearly separable is to substitute in terms that are non-linear functions of the original features in order to perform a mapping into a higher-dimensional space where the problem may in fact be linearly separable. Higher-order terms, sinusoids or logarithms are all examples of functions that might be used in such substitutions. Stated more formally, one can express a discriminant as a linear sum of nonlinear *basis functions,* also called *kernel functions*. For example, for two input features, $x_1$ and $x_2$, one can define five new variables, $z_1=x_1$, $z_2=x_2$, $z_3=x_1^2$, $z_4=x_2^2$ and $z_5=x_1x_2$, which can then themselves be combined in an entirely linear function. A linear discriminant may thus be defined in the five-dimensional $z$-space that corresponds to a nonlinear discriminant in the two-dimensional $x$-space.

Returning to the training of linear discriminants in the form of Equation 6.17, regardless of whether the $x$ are the original features or kernel functions, the weights $w$ are typically learned using gradient descent. Much as described above for backpropagation artificial neural networks, this involves iteratively reducing classification errors on training data by finding the gradient of the error function with respect to the discriminant weights, which are initialized randomly.

It is desirable to not only find a hyperplane that separates instances belonging to a class from instances not belonging to it, but to find the particular hyperplane that maximizes the distance between itself and training instances in order to improve its ability to generalize and thus not overfit the training data. The distance from the hyperplane to the instances closest to it on either side is called the *margin,* and the *optimal separating hyperplane* is the hyperplane that maximizes the margin. Those particular training instances whose points in feature space constrain the width of the margin are called *support vectors.*

*Support vector machines (SVMs)* are a popular and very effective type of classifier designed to find optimal separating hyperplanes. With respect to a simple two-class and two-feature problem, an SVM finds the line (or hyperplane, for feature spaces with a greater dimensionality) that maximizes the margin between the *support vectors.* Support vectors can be defined as the instances in feature space that lie on the maximum margin hyperplanes. The weights defining the discriminant are calculated by taking the average across all support vectors.

It is useful to note that this approach in effect discards all training instances that are not close to the decision boundary by only using the support vectors to calculate discriminants. The efficiency of SVMs can therefore be improved by using a more efficient method to remove training instances on the interior before training the SVM in order to reduce the work that needs to be done during SVM optimization.

Although SVMs are designed specifically for two-class problems, they can be generalized to *K*-class problems by defining *K* two-class problems, each one separating one class from all others. An alternative approach is to use pairwise separation of classes, as described above for linear classifiers in general, which requires *K(K-1)/2* discriminants.

Kernel functions are often used in SVMs in order to be able to deal with problems that are not linearly separable by, as discussed above, mapping lower-dimensional spaces to higher-dimensional spaces where hyperplanes can in fact be used to effectively discriminate between instances. SVMs that use kernel functions are sometimes called *kernel machines.* The most popular kernel functions for SVMs are polynomials of various degrees, radial-basis functions and sigmoid functions.

SVMs also sometimes use *slack variables* to deal with problems that are not linearly separable. Slack variables indicate the deviation from the margin of problem instances, which is to say instances whose feature coordinates fall on the wrong side of the discriminant or are on the correct side but are too close to it. These slack variables can be used to improve performance by using them in a penalty term. This in effect creates a *soft margin* that permits some misclassifications, and a *cost parameter* can be used to control the balance between forcing rigid margins and allowing training errors. It is often best to err on the side of not making the cost penalty too large, as otherwise the risk of overfitting the training data can become too large.

It is important to note that classifiers framed specifically as discriminant-based classifiers are by no means the only classifiers that can be used to estimate discriminants. For example, computational geometry can be used to easily estimate discriminants from a k-NN classifier, as noted earlier. In fact, SVMs are very similar to condensed k-NN classifiers that throw away training points that are not necessary to find discriminants. SVMs are also closely related to artificial neural networks, and an SVM using a sigmoid kernel function is in fact mathematically equivalent to a two-layer perceptron. The difference between discriminant-based classifiers like SVMs and classifiers such as feedforward neural networks and k-NN, however, is that discriminant-based classifiers estimate discriminants directly, rather than making them available as a by-product of other processing. Even most nonparametric classifiers effectively estimate feature probability densities, something that is avoided when using discriminant-based classifiers.

*Decision tree classifiers* are another type of discriminant-based classifier. A decision tree is a hierarchical data structure that performs classifications using a divide-and-conquer strategy that breaks complex decision functions into series of simple decisions. Decision trees consist of internal *decision nodes* and *terminal leaves*. Each decision node implements a *test function* that takes as input the features of an instance and outputs a branch of the node that points to a particular subordinate decision node, a process called a *split*. The instance's feature values are then in turn processed by the test function of the decision node that is selected by the previous split. This process starts at the root of the tree and is repeated recursively until a terminal leaf node is reached, whereupon a final class label is assigned to the instance.

The test function of a given decision node can take a variety of forms. If the features being considered are discrete, then there is a branch for different possible combinations of features. If the features are numeric, then inequalities involving thresholds are used as decision rules. In either case, each test function effectively implements a discriminant.

*Univariate* decision trees are a simple variant that only process a single feature in each decision node, although trees as a whole can still certainly process multiple features by processing different features in different decision nodes. The test functions of *multivariate* trees, in contrast, may take as input any of the available features. *Omnivariate* trees are a hybrid approach where some decision nodes process single features and some process multiple features.

Decision trees have a number of important strengths. Some varieties, especially univariate trees, process data very quickly, even faster than linear discriminant-based classifiers in some cases. Also, decision trees are very compact in terms of space efficiency, as only discriminants need to be encoded, unlike some nonparametric methods like k-NN that must memorize feature values. Perhaps the greatest advantage of decision trees, however, is their *interpretability.* Unlike many other classification methods, like feedforward neural nets, for example, decision trees are not just black boxes that can be used to perform classifications. Instead, the decision rules implemented by the test functions of decision nodes can be examined in order to gain information about the solution to the problem that can be theoretically meaningful for the particular problem domain. This process is called *knowledge extraction,* and yields a hierarchy of if-then rules that can be applied to features in order to arrive at classifications. This is, of course, exactly what a decision tree actually does when performing classifications, and it is the resemblance to a naturally human intuitive way of making decisions that makes the discriminants learned by decision trees much more human-meaningful than the models learned by most other classification algorithms.

It can be useful to calculate *rule support* when using decision trees, which is to say the percentage of training data *covered* by each rule, where a rule is said to be covered by an instance if the instance satisfies all conditions of the rule. Rule support can help show which decisions and which features are most important for arriving at classifications.

Even though multivariate trees can solve more complex problems than univariate trees, the rules that they learn tend to be more complex and more difficult for humans to interpret. Univariate trees are often preferable to multivariate trees because of this and because of their efficiency, and in practice they can still often perform quite well even when applied to moderately complex problems. An additional advantage of univariate trees is that they essentially perform their own feature selection, as they ignore features that they do not need.

The disadvantage of decision trees, especially univariate decision trees, is that they are not as effective as many other algorithms at learning complex models. Nonetheless, the interpretability of decision trees and their speed has caused them to remain reasonably popular, at least for problems that are not too complex. They are especially popular for use in classifier ensembles, as discussed in Section 6.2.7.

Decision trees learn their models from training data using a process called *tree induction.* There are several learning algorithms that can be used to implement this, including the well-known C4.5 algorithm (Quinlan 1993). In most cases, the goal of the process is to find the smallest (and therefore simplest, least susceptible to overfitting and easiest to interpret) tree that can be constructed with no error on the given training data.

There are typically many possible such trees and, since finding the smallest tree is NP complete, one is forced to use local search procedures, most commonly greedy algorithms. The most common approach is start at the root, look for the best split, and then proceed recursively down the tree until no more splits are needed and a leaf can be created. The goodness of a split is measured using a metric called the *impurity measure*. A split is pure if, after the split, all instances in each branch belong to the same class. If a split cannot be pure, then the instances are split such that the split minimizes impurity.

Unfortunately, noisy training data can easily lead to very large trees that overfit the data. Furthermore, splitting tends to favour those features with many possible values, which can also lead to overly large and complex trees. In order to minimize such problems, tree construction often ends when nodes become pure beyond some threshold, rather than strictly requiring an impurity of zero. *Entropy, Gini index* and *misclassification error* are three different ways of measuring impurity that for the most part perform similarly. An alternative approach is to introduce a metric that penalizes too much branching. In practice, it is generally best not to split a node if the number of instances reaching it is smaller than a certain percentage of the training set, such as 5%, regardless of impurity.

The process of reducing the size of a decision tree is called *pruning.* Stopping tree construction before an impurity of 0 is reached, as discussed above, is called *prepruning* the tree. *Postpruning* occurs after a full tree has been grown such that all leaves are pure and there is no training error. Postpruning involves setting aside a portion of the ground-truth data into a *pruning set* and using it to test the full tree for overfitting so that it can then be pruned appropriately.

### 6.2.7 Ensemble learning

*Ensemble learning* involves using multiple classifiers to solve individual classification problems. Each of the component classifiers is called a *base learner*. In principle, any learning algorithm could be used by any given base learner, and the various base learners in an ensemble might each use either the same or different algorithms. There are a wide variety of techniques that can be used to coordinate the base learners in order to output overall classification results from the ensemble, and there are also a variety of ways of training ensembles.

The practice of combining classifiers into ensembles is inspired by the notion that the combined opinions of a number of experts is more likely to be correct than the opinion of a single expert. Based on this, one might intuitively posit that an ensemble of classifiers will likely similarly perform better than one of its component classifiers operating individually. In practice this can often in fact be the case, assuming the use of a well-designed ensemble, although it is certainly not guaranteed.

One might reasonably question whether the increased computational demands and implementation complexity that typically accompany ensemble classification are justified if one is not guaranteed an increase in performance. Dietterich (2000) has proposed three reasons why classifier ensembles are worthwhile:

- *The statistical reason:* Suppose one has a number of trained classifiers. Although it is easy enough to find how well they each perform on the training and validation samples, this knowledge still offers only an estimate of how well the classifiers will each generalize to the instance population as a whole. If several of the classifiers performed similarly on the validation data, then there is no way of knowing which classifier is in fact the best with respect to the population. If one chooses a single classifier, one runs the risk of accidentally choosing one of the poorer ones. This statistical argument is particularly strong in cases where only limited training and validation data is available, as the evaluation of individual classifiers using validation sets is likely to have a potentially high error.

- *The computational reason:* This argument applies to classifiers that train using hill-climbing or random search techniques. Training several multilayer perceptrons, for example, on the same training data might result in significantly different learned models, depending on the randomly generated initial weights. Aggregating these classifiers into an ensemble can take advantage of the multiplicity of solutions offered by each of the classifiers. The computational argument also highlights the particular appropriateness of *unstable learners* for ensemble classification, which is to say algorithms like decision trees, condensed k-NN[153] or multilayer perceptrons that are very sensitive to differences in training data, in terms of factors such as the instances present or the features extracted, and can potentially converge to very different trained models when this information is changed only slightly.
- *The referential reason:* This argument is based on the fact that there is no guarantee that the types of classifiers that one is using for a particular problem could ever converge to a theoretically optimal solution. To provide a simple example, consider a case where a researcher mistakenly believes that a given problem is linearly separable, and decides to use a linear classification algorithm. In reality, the optimal discriminants will be non-linear, so it is not possible for any linear classifier to perform optimally individually. However, an ensemble of linear classifiers could approximate non-linear decision boundaries, and could therefore potentially perform better than any single linear classifier could.

An essential element in the effectiveness of classifier ensembles is their *diversity*. If all of the classifiers in an ensemble tend to misclassify the same instances, then combining their results will have little benefit. In contrast, a greater amount of independence between the classifiers can result in errors by individual classifiers being overlooked when the aggregate results of the ensemble are combined. So, even if the

[153] Only condensed nearest neighbour classifiers are unstable, as a nearest neighbour classifier without data thinning is in fact a good example of a stable learner.

component base learners of two different ensembles all have the same average classification error rate, the ensemble that has the greater base learner diversity will likely have a better overall ensemble classification error rate, which is what really matters. Many of the most successful ensemble approaches are therefore based on increasing classifier diversity.

Classifier ensembles can achieve diversity in a variety of ways:

- Each base learner may use a different learning algorithm. Different algorithms tend to be based upon different assumptions, so it is hoped that the bias error due to incorrect assumptions will be filtered out through the use of multiple algorithms. For example, one base learner might be parametric and another nonparametric, or one might use linear discriminants and another non-linear discriminants.

- Different base learners may utilize the same learning algorithm, but with different hyperparameter settings. For example, one might vary *k* in a k-NN classifier or the number of hidden nodes in a multilayer perceptron. Randomly generated initial conditions of classifiers can also be varied, such as the initial weights of an artificial neural network.

- Different base learners may use different representations of the input data, namely different features. This can be particularly useful when dealing with very different types of input, such as speech recognition based on video of the speaker's lips as well as the audio signal. This approach can also be applied to classification problems in general, where different feature subsets are provided to different base learners based perhaps on random partitioning or on the output of different dimensionality reduction algorithms.

- Different subsets of the available training data may be used to train the different base learners. As noted above, this is particularly appropriate when using unstable learning algorithms.

*Weak learners,* which are learners that are likely to have only poor classification accuracy when applied individually, are often used in classifier ensembles. Weak learners can have classification accuracies even as low as 55% on two-class problem, for example. The primary advantage of weak learners is that they tend to use simple classification models that are good at avoiding overfitting, something that can otherwise be a particular problem with classifier ensembles due to the increase in model complexity when multiple classifiers are combined together. Of course, even when using weak learners one must still be careful to avoid situations where the ensemble coordination methodology reintroduces a high risk of overfitting.

An additional advantage of weak learners is that they are usually quick to train, an important consideration if one is using an ensemble consisting of many base learners that must each be trained. *Decision stumps* (decision trees with very short depths imposed on them) are one of the most commonly used weak learners in classifier ensembles, although there are certainly other alternatives as well.

The obvious trade-off of using weak learners is that they are prone to underfitting, hence their poor individual performance. The assumption of classifier ensembles that use weak learners is that the bias error of individual learners is averaged out in the ensemble as a whole since many weak learners are used.

Methodologies for combining the classifications made by individual base learners can be divided amongst several general groups. The first, *classifier fusion,* involves merging the classification results of all of the base learners via a process such as voting. The second, *classifier selection,* involves using some system to dynamically select which classifiers to use for classifying each particular given instance. The *mixture of experts* or *stacking* methods, as described below, are examples of how these two approaches can be combined together.

There are also several basic approaches to training base learners and processing features when classifying. *Multiexpert combination* approaches involve base learners operating in parallel, such that the features of each instance to be classified are given to every learner. *Multistage combination* methods, in contrast, utilize serial approaches where each base learner is trained only or primarily on instances that learners earlier in the serial classification chain performed badly on. Then, during classification, the base learners can be sorted in increasing complexity so that a complex learner might only be used if the preceding simpler learners are not confident in their classification result for a given instance. There is a rough but not universal correspondence between classifier fusion and multiexpert combination approaches, and between classifier selection and multistage combination approaches.

Voting is perhaps the simplest approach to coordinating classifier ensembles. An average is compiled of the classification outputs of each base learner, and the class with the most votes wins. Each base learner may output a simple binary yes/no for each class, or it may associate certainty value with each class, depending on the algorithm used by each base learner. Each learner can also have a weight associated with it in weighted voting schemes, based on some measure of its accuracy relative to the other base learners. Voting essentially operates as an averaging filter that smoothes out errors made by individual base learners, and is particularly appropriate for base learners with a small bias and a small variance.

*Bagging,* which is short for *bootstrap aggregating,* is a voting method where each base learner is trained using a different training set. The training sets are generated by *bootstrapping,* which means that, given *N* training instances, *N* instances are drawn with replacement for each base learner. This means that some instances may not be drawn at all for any given base learner's training subset, and some may be drawn more than once.

Bagging tends to work best when the base learners are unstable algorithms, which is to say that they are sensitive to the potentially small differences in training data that bootstrapping produces. As one might expect from the discussion above, weak learners are usually used because this is helpful in overcoming variance error, and the diversity of learners helps to overcome the bias error. Decision stumps are a popular choice of base learner in bagging ensembles, as they are both weak and unstable.

Bagging is a good approach to use when only a limited amount of training data is available, since it enables the data to be reused by multiple base learners. However, if *N* is in fact very large then it is best to train each base learner on a subset of the training data consisting of fewer than *N* instances in order to help increase classifier diversity. In practice, though, bagging is rarely used when large amounts of training data are available, since alternative ensemble coordination techniques tend to perform better when training data is plentiful.

*Boosting* is an example of an ensemble coordination methodology that works better than bagging when reasonably large amounts of training data are available. Boosting involves serially training base learners on the training instances that previous base learners are unable to classify correctly. This means that successive base learners make up for the weaknesses of previous base learners, with the consequence that classifiers further along in the classification change correct the bias error of previous classifiers. Boosting can often result in exceptionally high classification accuracies, but it requires more training data than bagging in order to be effective, and it can be more susceptible to noisy training data and outliers.

It is also generally best to use weak learners in boosting ensembles, particularly near the end of the classifier chain. This is because only relatively few training samples will be available late in the chain, and one must therefore be especially careful not to overfit them. Unstable base learners are also often a good choice, but less essentially so than with bagging.

*AdaBoost,* short for *adaptive boosting,* is a boosting variant that is currently one of the most popular and effective ensemble classification algorithms. Although it does still typically require more training data than bagging in order to achieve good results, it does not need as much as more conventional boosting methodologies. Decision stumps are once again a particularly common choice of base learner for AdaBoost.

There are several different versions of AdaBoost. *Resampling* variants probabilistically draw training instances for a given base learner, where the probability of a given training instance being chosen is proportional to the difficulty learners earlier in the chain had in correctly classifying it. *Reweighting* variants of AdaBoost perform final classifications using a type of voting where each base learner is given a voting weight based on its performance during training.

The *mixture of experts* approach to classifier ensembles involves using a *gating learner* that adaptively learns voting weights to assign to base learners when voting is performed to arrive at the final class labels to assign to instances. The weights that are assigned to each learner by the gating learner depend on the particular instance being classified, since the gating learner is trained using feature values extracted from instances. This approach in effect encourages each base learner to become an expert specializing in classifying particular kinds of instances, and the gating learner learns which such experts are best to consult for any particular instance.

The *stacking* approach is similar to the mixture of experts approach, except that the coordinating learner is a *combiner learner* instead of a *gating learner.* Whereas a gating learner takes feature values as inputs and outputs voting weights associated with each base learner, the combiner learner takes in the output of each of the base learners as input and outputs the final classifications of the ensemble. The combiner therefore does not process feature values directly, and the base learners do not vote directly. The combiner instead learns to map base learner outputs directly to class labels, which means that it effectively learns how the base learners make errors relative to one another.

The base learners in both the mixture of expert and stacking approaches are usually weak, but the coordinating learners are not. A multilayer perceptron is a common choice for the gating or combiner learner, as multilayer perceptrons naturally models weightings. Diversity can be incorporated into the base learners through varied features, training data or hyperparameters and, as always, one must be careful to avoid reintroducing variance error by overfitting the coordinating classifier. For example, the combiner should be trained using data that was not used to train the base learners.

*Cascading* is another ensemble approach that involves a serial chain of base learners. For cascading to work, each base learner must be able to associate certainty scores with its classification outputs or, alternatively, some implementations use an additional coordinating classifier to judge the reliability of the base learners given particular feature inputs. In either case, if there is sufficient certainty in the classification output of a given base learner, then this classification is output as the final decision of the ensemble. If the ensemble is not sufficiently certain of the base learner's classification it then defers to the next base learner in the chain. This means that the bias error of each classifier is essentially estimated for each instance and this is used to decide if the learner is competent to make a decision.

Cascading can be sensitive to variance error that can result from inflated certainty estimations. It is therefore usually best not to use weak learners in cascading ensembles, as one wants the early classifiers to be correct most of the time, or at least competent enough to tell when they may be incorrect. The use of k-NN classifiers as base learners is fairly common, as they are typically not weak learners and because they output a certainty score, namely the fraction of *k* corresponding to the chosen class. Of course, sufficient training data must be available to allow diversity in the base learners if k-NN is used. Multilayer perceptrons offer a good alternative to k-NN when less training data is available, as they are unstable classifiers, unlike uncondensed k-NN classifiers.

Of course, the base learners used in cascading ensembles do not all need to be based on the same algorithm. In practice the base learners are usually sorted in terms of space or time complexity such that the learners earlier in the chain are faster. This can significantly reduce consumption of computing resources. In cases such as this, parametric classifiers such as the naïve Bayes' classifier can be used, since they are fast and the posterior probability provides a good certainty estimation. The last classifier in the chain is still often nonparametric, however, as it serves as the last resort if none of the other classifiers can produce a classification with sufficient confidence. Also, the combination of parametric with nonparametric methodologies allows the parametric methods to classify typical instances and the nonparametric method to classify the exceptions.

Classifier ensembles based on e*rror-correcting output codes (ECOC)* attempt to break a difficult classification problem into a set of simpler problems. Classifiers specialized in each of the simpler problems can then be trained and combined to arrive at final classifications.

Each base learner is a binary classifier with an output of -1/+1. These classifiers may simply distinguish one class from all other classes, they may distinguish between only two classes or they may be trained to output the same value for different classes in order to generate useful code words. In any case, the output of the ensemble in response to any instance is in effect a binary code word.

A code matrix of size *K x L* can then be formed, where *K* is the number of candidate classes and *L* is the number of base learners. The entries of the matrix are set to the ideal ground-truth classification results of each base learner. Each row (i.e., each class) is thereby assigned a code word uniquely identifying its class. ECOC ensembles always have many more base learners than candidate classes, as this incorporates redundancy into the code words, with the result that the Hamming distance between the code words increases. There are also a number of other methods that can be used to help increase the

Hamming distance. In any case, one of the many available binary error correction algorithms available can then be applied to the code words output by the classifier ensemble when classifying individual instances in order to flip incorrect bits generated due to errors made by individual base learners (Kuncheva 2004).

Those interested in finding further information on classifier ensembles in general may wish to consult Kuncheva's book (2004) or the book edited by Kandel and Bunke (2002).

### 6.2.8 Ontological learning and blackboard systems

Classification approaches that take advantage of structure in the class ontology of a problem can also be used. For example, one might organize musical genres into a hierarchical ontology where broad genres such as Jazz are at the root of the genre tree and narrower genres such as Dixieland, Swing and Bebop are nearer the tree's leaves. Since classification problems generally increase in difficulty with the number of candidate classes, it can be useful to reduce a complex and difficult classification problem involving many leaf genres to a series of easier sub-problems. To continue the above example, the classifier might first solve a simple three-class Jazz/Classical/Rock problem and then address a series of increasingly more specific sub-problems using specialized models that are each trained to classify only amongst the direct descendants of each node in the tree.

Such a hierarchical classification approach involving a hierarchically organized set of specialized classifiers has in fact been effectively used for musical genre classification in this manner (McKay 2004). This approach is also often applied to computer vision problems, where objects can be reduced to collections of smaller shapes, or otherwise organized hierarchically (e.g., Barutcuoglu and DeCoro 2006).

This hierarchical classification approach not only has the advantage of taking advantage of knowledge about the problem domain to improve classification success rates, but can also result in misclassifications that are less severe than they might otherwise be when misclassifications do occur. This is because serious misclassifications will usually only occur when they are made near the top of the hierarchical class ontology, since classes become increasingly similar as one descends down the tree. Furthermore, this approach facilitates weighted penalization during training because of the knowledge embedded in the representation of the class ontology, although training schemes that penalize misclassifications between dissimilar classes more than misclassifications between similar classes during training can certainly be used even when hierarchical classifier ensembles are not used.

*Blackboard systems* offer a particularly interesting approach that was popular in the past, but is less fashionable recently, perhaps unfairly so. Blackboard systems consist of *knowledge sources* that are each experts in different problem domains and that each write hypotheses about a problem to a shared data space called a *blackboard.* Each knowledge source can then access information written to the blackboard by other knowledge sources and use it to refine its own hypotheses. Blackboards provide an excellent infrastructure for combining expert systems with supervised and unsupervised learning methodologies such that known information about a given problem domain can be incorporated into overall ensembles that also take advantage of machine learning to make up for areas of the problem domain that are poorly understood. The book edited by Jagannathan, Dodhiawala and Baum (1989) is a good source of information on blackboard systems.

### 6.2.9 Sequential learning

The classification algorithms described above and in Section 6.2.6 are designed to process instances individually, which is to say that it is assumed that the instances are independent and their ordering is irrelevant. Although this is certainly a reasonable assumption for certain types of classification, such as classification of songs by composer or mood, there are also many other cases where the sequence of inputs is significant. Melody and harmony, for example, are inherently sequential concepts. It is much more likely that a ii chord will be followed by a V chord than by a I chord in tonal music, for instance and, to give another example, speech recognition involves learning to associate a sequence of phonemes with a particular word.

There are a number of classification algorithms designed specifically to deal with data where the sequence of inputs is meaningful. Although ACE in its current form does not yet include any such algorithms, their inclusion in a future version is a priority, so it is appropriate to briefly touch on a few highlights of sequential classification here.

*Recurrent neural networks,* as briefly discussed in Section 6.2.6, are one well-known algorithm for dealing with sequential data. They incorporate a memory of previously encountered data via links leading from output nodes to nodes in previous layers.

*Conditional random field* classifiers offer another alternative for classifying sequential data, in this case using a probabilistic approach. An undirected graphical model is constructed, in which each vertex represents a random variable whose distribution must be inferred and in which each edge represents a dependency between the random variables of the two associated vertices.

*Hidden Markov models (HMMs)* are perhaps the best-known sequential classifier, and are in effect a special case of conditional random fields with stricter assumptions about the input and output sequence distributions. Like conditional random fields, HMMs can be visualized as graphs, although in the case of HMMs the edges are directed, each vertex represents a state and the edges represent transition probabilities between the states. HMMs will be explained in some detail here, as they are a good illustration of how sequential classification can work.

In order to illustrate HMMs, it is useful to consider a system that can be in one of $N$ distinct states, $S_1, S_2, \ldots, S_N$. The state at some time or position, $t$, is denoted as $q_t$. The system may move to a new state with some probability dependant on the previous states of the system in time:

$$P(q_{t+1}=S_j \mid q_t=S_i, q_{t-1}=S_k, \ldots) \qquad (6.18)$$

A *first-order Markov model* makes the assumption that the state at time $t+1$ only depends on the state at time $t$, and not on time $t-1, t-2, \ldots, t_1$. This is equivalent to saying that the future depends on the present but not on the past. A problem can be further simplified by assuming that transition probabilities are independent of time, which is to say that the probability of going from $S_i$ to $S_j$ does not change as time goes by. Put more formally, these assumptions mean that the probability of a transition, $a_{ij}$, from state $i$ to state $j$ becomes:

$$a_{ij} = P(q_{t+1}=S_j \mid q_t=S_i) \qquad (6.19)$$

where

$$a_{ij} \geq 0 \text{ and } \sum_{j=1}^{N} a_{ij} = 1 \qquad (6.20)$$

and where $N$ is, once again, the number of possible states. The set of all $a_{ij}$ can then be combined into an $NxN$ matrix $A$ whose rows sum to 1:

$$A=[a_{ij}] \qquad (6.21)$$

The initial state, or first state, can be defined using the *initial probabilities,* $\pi_i$, which gives the probability that the first state in the sequence is $S_i$:

$$\pi_i = P(q_1 = S_i) \text{ where } \sum_{i=1}^{N} \pi_i = 1 \qquad (6.22)$$

The initial probabilities can then be expressed as a vector:

$$\Pi = [\pi_i] \qquad (6.23)$$

where the $N$ elements sum to 1.

In an *observable Markov model* the current state can be observed. Put more formally, the value of $q_t$ is known at time *t,* and one can observe the sequence of states that the model passes through over time, *O:*

$$O = Q = \{q_1 q_2 \ldots q_T\} \qquad (6.24)$$

whose probability is

$$P(O = Q \mid A, \Pi) = P(q_1) \prod_{t=2}^{T} P(q_t \mid q_{t-1}) = \pi_{q1} a_{q1q2} \ldots a_{qT-1qT} \qquad (6.25)$$

Here $\pi_{ai}$ is the probability that the first state is $q_1$ and $a_{q1q2}$ is the probability of going from state $q_1$ to $q_2$.

In hidden Markov models, however, the states are not directly observable, although an observation can be made whenever a state is visited that is a probabilistic function of the state. In other words, information is observed when each state is visited that can give hints as to what that state might be. If there are *M* possible discrete observations of such hints of what the state might be, $v_1, v_2, \ldots, v_M$, then:

$$b_j(m) = P(O_t = v_m \mid q_t = S_j) \qquad (6.26)$$

where $b_j(m)$ is the *emission probability* or *observation probability* that $v_m$ is observed in state $S_j$. These emission probabilities can be collected into the matrix $B=[b_j(m)]$. It is important to reemphasize that the probabilities are assumed to not depend on *t,* and that the values observed constitute the observed sequence *O*. The actual state sequence *Q* cannot be observed directly because the model is hidden, but it can be inferred from the observed sequence *O*. It is then useful to calculate the *Q* that is most likely to have generated *O*. The parameters of a hidden Markov model are thus:

$$\lambda = (A, B, \Pi) \qquad (6.27)$$

*N* and *M* are implicit in these.

HMMs are usually used to solve three basic kinds of problems:

- Given a model $\lambda$, what is the probability of a given observed sequence of events? In other words, what is $P(O|\lambda)$? There is an efficient recursive procedure for calculating $P(O|\lambda)$ called the *forward-backward* procedure.

- Given $\lambda$ and $O$, what is the most likely state sequence $Q$? In other words, what is the $Q^*$ that maximizes $P(Q|O,\lambda)$? The *Viterbi algorithm* can be used to find the most likely state sequence using dynamic programming.

- Given $X$, a set of $k$ observed training sequences, what is the model $\lambda$ that maximizes the probability of generating $X$? In other words, what is the $\lambda^*$ that maximizes $P(X| \lambda)$? The *Baum-Welch* algorithm can be used to solve this problem.

A typical approach to using HMMs in classification is to have a set of HMMs, each one modelling sequences associated with one class. For example, for a speech recognition problem one could train a separate HMM model, $\lambda_i$, for each word of interest, where each state would correspond to a different phoneme. Given an unknown word consisting of a sequence of phonemes to classify, $O$, the $P(O|\lambda_i)$ could be calculated for each $\lambda_i$. Bayes' rule can then be used to find the posterior probabilities $P(\lambda_i|O)$, and $O$ is classified as the word associated with the $\lambda_i$ with the highest posterior.

If the input observations to an HMM are continuous rather than discrete, then one possibility is to discretize them using a technique such as *vector quantization*. This converts continuous values to discrete units. For example, a continuous sound signal of someone speaking can be separated into discrete phonemes by a pre-processing stage, and the resulting phonemes can then be processed by an HMM as described above.

### 6.2.10 Meta learning

*Meta learning* is a term that is overloaded in machine learning. For example, it is sometimes used to simply refer to classifier ensembles in general, a usage that is not adopted here. Instead, meta learning is used here to refer to the process of automatically selecting a good classification methodology or, potentially, combination of classification methodologies, to apply to a given classification problem using a data-driven empirical process. The choice of classification configuration can be very important decision since, as discussed previously in this section, different algorithms make different assumptions and are more or less suitable for different kinds of problems.

Meta learning is often implemented by automatically performing experiments in the given problem domain where each of a variety of classification configurations are trained separately and tested comparatively in order to arrive at estimates of how well each performs. Different classification configurations can be evaluated with respect to classification error, classification consistency, training space and time complexity, classification space and time complexity, etc. Preference can also be given to algorithms that have desirable qualities for the particular problem domain, such as transparency, for example.

An alternative approach is to either replace or supplement this process of successive experimentation with data analysis techniques that can automatically examine feature statistics and domain needs in order to form hypotheses about which types of classifiers might be a best fit for a problem, given known characteristics of different classification algorithms. Van Someren (2001) and Kalousis, Gama and Hilario (2004), for example, provide some interesting background relevant to this latter approach.

Yet another approach to meta learning involves training a machine learning system to choose a classification configuration for particular problems, rather than just experimentally ranking the estimated performance of different configurations directly. Although some ensemble techniques such as stacking or mixture of experts do effectively do this, they tend to do so in a very limited sense, as they typically only choose between learners that are of the same type but trained differently, or between a few relatively simple algorithms. Work has also been done on training more sophisticated specialized meta learning coordinators (e.g., Dzeroski, Todorovski and Blockeel 2002).

Meta learning can be used to select not only different algorithms, but also different algorithm hyperparameters and, if one is willing to experiment with classifier ensembles as well as individual classifiers, different approaches to combining algorithms into classifier ensembles. Meta learning can also be used to select between different pre-processing or post-processing techniques, such as dimensionality reduction algorithms.

As noted in Section 6.1.2, there are a number of important advantages to meta learning. Perhaps most important among these is that meta learning can make effective pattern recognition techniques available to users who might otherwise lack sufficient background in machine learning to select appropriate algorithms, and even individuals with significant experience in machine learning can benefit from meta learning. The downside of meta learning, of course, is that it can be a computationally intensive and time consuming process to perform experiments with a wide variety of classifier configurations, particularly if many features and training instances are available to be processed.

As discussed in Section 6.2.4, one must be careful to use statistically valid methods when comparing classifiers. A reserved publication set should be used to validate the final classification configuration output by the meta learning system in order to help ensure that the meta learning system itself is not overfitting the training data. Work has also been done on dataset sub-sampling techniques that can be used to partition training data in ways that improve meta learning performance (e.g., Fürnkranz et al. 2002). Appropriate hypothesis testing methodologies must also be used to verify that performance differences between different classification methodologies are actually statistically significant.

## *6.3 Existing machine learning software libraries*

### 6.3.1 General machine learning software

There are many available implementations of algorithms like those described in Section 6.2. Some of these are simple code libraries that may be embedded in external applications and some are complete applications with sophisticated user interfaces. Although there are far too many implementations to survey with any completeness here, it is appropriate to provide an overview of some of the best-known and most flexible pattern recognition frameworks.

RapidMiner, formerly known as YALE (Ritthoff et al. 2001; Klinkenberg, Mierswa, and Ritthoff 2005; Mierswa et al. 2006; http://rapid-i.com), is one of the most popular, powerful and professional-looking data mining environments. RapidMiner is implemented in Java, and comes in an open-source version as well as an expanded enterprise edition. RapidMiner's source code is available under a GNU General Public License as well as a proprietary OEM commercial license. It includes powerful graphical and command-line interfaces as well as an API.

MatLab is a particularly popular framework for implementing and distributing machine learning implementations, and many of these toolboxes have become quite popular, such as the Neural Network Toolbox.[154] The disadvantages with all MatLab toolboxes, unfortunately, is that one must purchase and use the proprietary Matlab software.

PRTools (van der Heijden et al. 2004) is one particularly appealing MatLab toolbox because it includes implementations of many different algorithms. PRTools' license does not permit it to be redistributed freely, however. Octave (Eaton and Rawlings 2003) is a free alternative MatLab toolbox that includes a limited number of tools that can be used for classification.

Torch (Collobert, Bengio and Marithoz 2002) provides a MatLab-like environment for applying machine learning, although it is implemented in C. It places a particular emphasis on efficiency, and is distributed under a BSD license.

Orange (Demsar et al. 2004) is a set of C++ implementations of machine learning and data mining algorithms that also includes flexible data input and manipulation tools. Orange's functionality can be accessed via the Python scripting language or via a visual programming environment.

SciPy[155] is another open-source software package associated with Python. It may certainly be used for machine learning, although its main focus is on mathematical and scientific computing in general, not specifically machine learning.

The R Project (Ihaka and Gentleman 1996) is another alternative. Its particular focus is on statistical computing, and it is particularly well-known for its graphing capabilities.

SEASR, or the Software Environment for the Advancement of Scholarly Research (Llorà 2008), is a Java-based framework that is particularly attractive because it facilitates distributed processing of computationally intensive tasks and because it provides a flowchart-based interface. Although the focus of SEASR tends to be more application driven, it can still be used for heavyweight data mining.

There are many other less well-known machine learning frameworks available, such as Java-ML,[156] MLC++[157] and Tanagra.[158] There are also many commercial data mining

[154] www.mathworks.com/products/neuralnet/
[155] www.scipy.org
[156] java-ml.sourceforge.net
[157] www.sgi.com/tech/mlc/
[158] chirouble.univ-lyon2.fr/~ricco/tanagra/index.html

frameworks, such as XELOPES,[159] but these are almost always both proprietary and closed source, which significantly limits their academic research potential.

There are also several excellent frameworks designed with a specific focus on music, but they tend to implement only limited data mining functionality relative to general purpose systems like RapidMiner. Marsyas (Tzanetakis and Cook 2000), CLAM (Arumi and Amatriain 2005), MIRToolbox (Lartillot, Toiviainen and Eerola. 2008) and sMIRk (Fiebrink, Wang and Cook 2008b), for example, are particularly well-designed MIR frameworks, but tend to focus much more on audio processing and feature extraction than on more sophisticated machine learning techniques like ensemble classification and meta learning.

One particularly interesting exploratory system that is both music focused and commercial in nature is Sony EDS (Pachet and Aymeric, 2004). This framework has the advantages of allowing dynamic construction of audio features and of allowing the use of specifiable heuristics. However, Sony's approach focuses specifically on genetic searches rather than on experimenting with a diverse range of algorithms, and emphasizes the tasks of similarity and recommendation rather than music classification in general.

### 6.3.2 Weka

Weka (Witten and Frank 2005; http://www.cs.waikato.ac.nz/ml/weka/), or the Waikato Environment for Knowledge Analysis, is another well-known Java-based data mining package. It is given special attention here because it is the most popular machine learning library in the MIR research community and because ACE uses Weka's classifier implementations. Weka is not only useful as a source of off-the-shelf algorithm implementations, but also as a platform for developing and distributing new algorithms.

Weka is entirely open-source and is available under a GNU General Public License. It includes implementations of a wide variety of machine learning algorithms and related tools, and there is relatively good supporting documentation available for it. Weka reads and writes Weka ARFF files, which are essentially enriched delimited text files, and also provides access to SQL databases via JDC.

[159] www.prudsys.com/Produkte/Algorithmen/Xelopes

Weka provides several user interfaces. The “Explorer” interface, which is intended as the main interface, is a GUI that permits users to pre-process data via “filters,” identify relationships between features, apply dimensionality reduction algorithms, apply supervised classification and regression algorithms, apply clustering algorithms, estimate the accuracy of learned models and visualize results. As indicated by its name, the Explorer interface is largely intended for exploring and experimenting with data.

The other Weka user interfaces are perhaps more suited for heavyweight machine learning. The command-line interface is particularly powerful, and can be capitalized upon by using scripting to automate sequences of tasks. As a visual programming alternative to the command-line, the “Knowledge Flow” interface allows Weka’s functional components to be accessed via a component-based GUI. The “Experimenter” is another graphical interface that is primarily intended for comparing the performance of Weka’s machine learning algorithms on collections of datasets. The most useful of Weka’s interfaces, from the perspective of embedding Weka’s functionality into other software applications like ACE, is Weka’s well-documented API.

### 6.3.3 Meta learning software

There are a few available systems that implement experimental meta learning functionality. Many of these are excellent general-purpose systems, and are therefore briefly reviewed here, but many of them unfortunately fail to meet the particular needs of music classification (see Section 6.4.5).

Consultant (Sleeman, Oehlman, and Davidge 1990), an ESPIRIT project, is an early software tool designed to aid users in selecting appropriate data mining tools by comparing different algorithms. ESPIRIT later produced two other meta learning-related efforts, namely Statlog (Michie, Spiegelhalter, and Taylor 1994) and METAL (Widmer 1996). The Data Mining Advisor (Brazdil, Soares and Costa 2003), a descendant of the METAL project, is a web-based assistant that outputs a ranked list of algorithms according to a weighted combination of parameters such as time and accuracy.

The Algorithm Selection Tool (Lindner and Studer 1999), or AST, offers a means of selecting algorithms using a case-based reasoning approach. Users are given explanations for each algorithm recommendation in the form of past experiences that are available for the case base.

The Intelligent Discovery Electronic Assistant (Bernstein, Provost and Hill 2005), or IDEA, can be used to automatically choose appropriate data mining "processes," depending on user needs such as accuracy and time complexity. A process may involve data preparation, classification and post-processing. The system generates possible processing steps for a specific task and then ranks them according to user specifications. The speed estimation is pre-determined based on tests with UCI data and the accuracy estimation is "autoexperimented" using a subset of the dataset.

SwissAnalyst (Povel and Giraud-Carrier 2004) is an open-source Java data mining environment built upon Weka. It places a particular emphasis on exploratory business research. The GUI is divided into five basic sections: Data Sources (loading datasets), Pre-Processing (dataset transformations), Data Exploration (statistics and visualization), Model Definition (defining and executing mining models), and Trained Models (evaluating and applying models to new datasets). SwissAnalyst can pre-select classification schemes based on the type of input data.

TunedIT (tunedit.org) is a new application for comparing different machine learning algorithms. It is designed specifically to evaluate the performance of new machine learning algorithms relative to existing ones, but it can potentially be adapted to meta learning in general. TunedIT is also associated with a knowledge base to which experimental results can be published and a repository where resources such as algorithm implementations and data sets can be stored.

Giraud-Cartier and Keller (2002) provide an excellent historical background on meta learning systems.

## *6.4 ACE's functionality*

As noted in Section 6.1.2, ACE is a software application designed to make powerful machine learning techniques accessible even to researchers with little or no background in machine learning. This includes users such as musicologists, music theorists, librarians and psychologists who possess valuable insights into MIR problems but might not have training in machine learning. This is important, as experts in pattern recognition rarely have specialized knowledge in individual application areas like music, and experts in these application areas rarely have expertise in pattern recognition. ACE's meta learning functionality can be used to automatically experimentally evaluate the effectiveness of different machine learning algorithms with respect to any given problem, which means that ACE can automatically recommend the specific machine learning algorithms that are appropriate for a particular problem to a researcher without the researcher needing to understand how the machine learning algorithms actually work. ACE then allows users to train a classification model using the chosen algorithm(s), and then use this trained model to classify unidentified instances.

ACE is also intended to be a useful tool for users who do in fact have strong backgrounds in machine learning by providing a framework for developing, evaluating and applying machine learning algorithms. Of particular interest, expert users can use ACE to benchmark and compare the performance of new classifiers and features relative to the baseline provided by ACE's already implemented algorithms and features that can be extracted using jMIR's feature extractors. This process is facilitated by the fact that ACE uses Weka classifiers as plug-in modules, something that makes it easy to implement new algorithms in the Weka framework and then experiment with them in ACE.

ACE is designed specifically for music classification, and has a number of important strengths in this domain relative to alternative general machine learning systems, as explained below. However, ACE can certainly also be easily applied to non-musical classification problems as well.

Figure 6.5 provides an overview of ACE and its role in jMIR. The input to ACE consists of extracted feature values, ground-truth labels[160] and an optional structured class ontology.[161] This information can all be represented in ACE XML Feature Value, Instance Label and Class Ontology files, respectively (see Chapter 7). This is convenient in using ACE with other jMIR components, such as feature values saved in ACE XML Feature Value files by jAudio, jSymbolic or jWebMiner, or labelled ground-truth saved in

[160] Ground-truth labels are only needed if a model is to be trained or if cross-validation or meta learning are to be performed. They can of course be omitted if ACE is simply being used to classify instances using a previously trained model.

[161] Structured ontologies can be used by certain classification algorithms, such as hierarchical classifiers, and can also facilitate weighted training schemes that penalize misclassification into similar classes less severely than other types of misclassification. Such algorithms are not yet implemented in ACE, but its file formats and data structures are already designed to store and communicate structured class ontologies in anticipation of the addition of this functionality in the future.

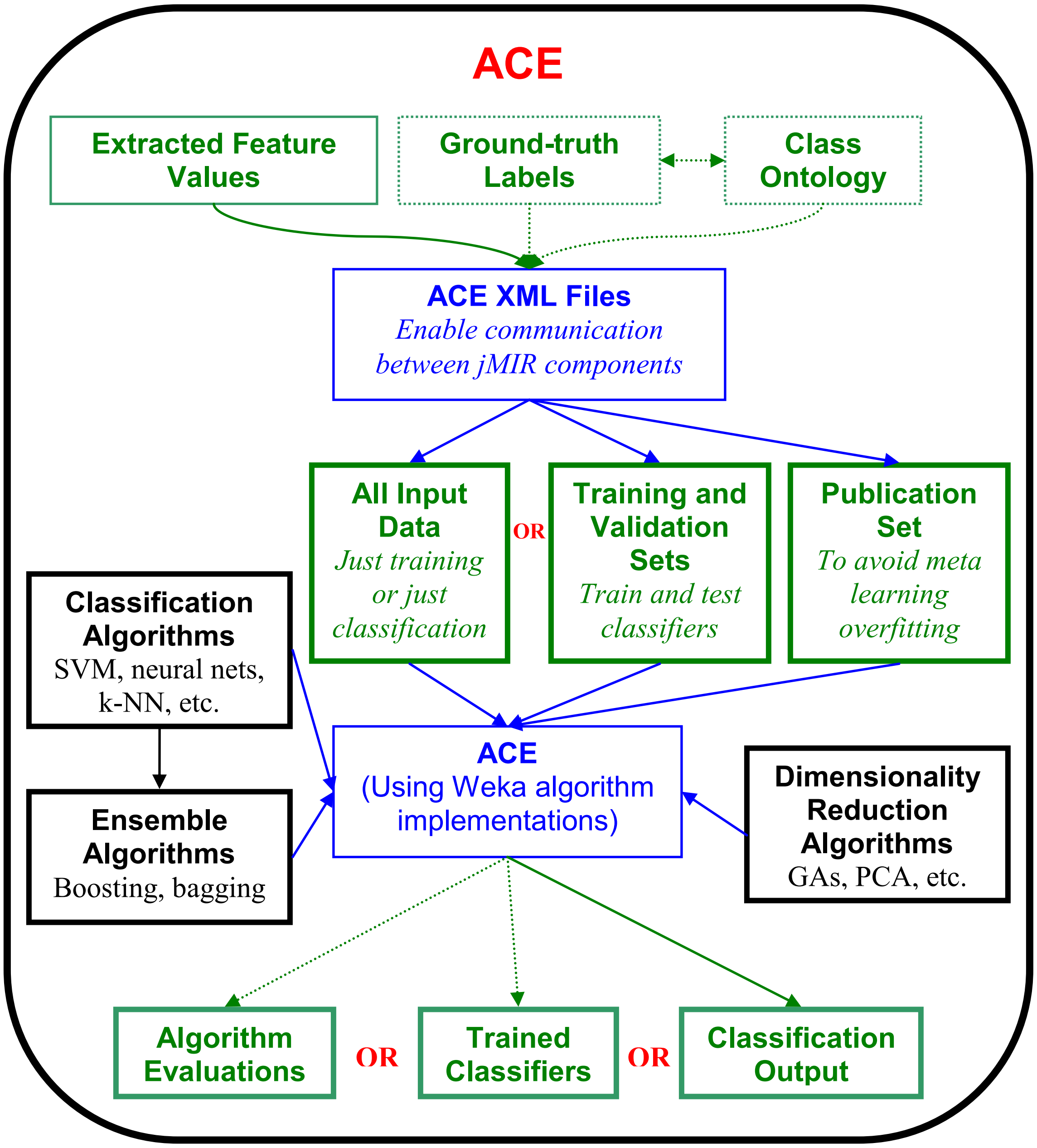

ACE
Extracted Feature Values
Ground-truth Labels
Class Ontology
ACE XML Files
Enable communication between jMIR components
All Input Data
Just training or just classification
OR
Training and Validation Sets
Train and test classifiers
Publication Set
To avoid meta learning overfitting
Classification Algorithms
SVM, neural nets, k-NN, etc.
Ensemble Algorithms
Boosting, bagging
ACE
(Using Weka algorithm implementations)
Dimensionality Reduction Algorithms
GAs, PCA, etc.
Algorithm Evaluations
OR
Trained Classifiers
OR
Classification Output

**Figure 6.5:** The flow of information in and out of ACE. ACE XML (or Weka ARFF) files are used to input feature values, ground-truth labels and/or a structured class ontology. ACE then automatically partitions the input instances based on whether classification, training, cross-validation or meta learning experimentation is to be performed. ACE then uses its catalogue of dimensionality reduction and classification algorithms, including classifier ensembles, to perform the desired task. It then outputs predicted instance labels, trained classification models or algorithm evaluations, depending upon the task selected by the user.

ACE XML Instance Label files using jMIRUtilities (see Section 8.8). Feature values and ground-truth may alternatively be input to ACE via Weka ARFF files (Witten and Frank 2005), although the use of ARFF instead of ACE XML imposes certain limitations on how ACE can be used, as certain types of information that can be taken advantage of by ACE cannot be represented in ARFF.

### 6.4.1 Core processing performed by ACE

ACE is designed to perform four basic types of tasks:

- **Training:** A classification model is trained and saved based on feature values and associated ground-truth class labels provided by the user. The particular dimensionality reduction and machine learning algorithms to be used are specified by the user, based either on the results of prior meta learning performed with ACE or on some other reasons that the user may have.
- **Classification:** A previously trained classification model specified by the user is used to classify instances based on features that have been extracted from them. ACE then outputs the resulting predicted class labels for each instance.
- **Cross-validation:** Cross-validation is performed on user-specified feature values and ground-truth using dimensionality reduction and classification algorithms also specified by the user. This can be useful in evaluating the performance of a particular feature set or a particular algorithm, for example. A variety of performance statistics are output by ACE.
- **Experimentation:** Meta learning is applied to the provided data in order to evaluate the appropriateness of different classification approaches to the given application domain. ACE experiments with different dimensionality reduction algorithms combined with different classification algorithms and, in some cases, different algorithm hyperparameters. ACE begins the experimentation process by generating several different feature subsets, one for each dimensionality reduction algorithm. Cross-validation is then performed once for each candidate classification algorithm on each such feature subset. Reports are then generated indicating the best performing dimensionality reduction and classification

algorithm pairing and providing associated statistics. Information is also provided on all other algorithm combinations that were experimented with as well.

ACE automatically partitions input data in ways that depend on the type of task that is to be performed. If only classification or only training is to be performed, then all of the input data will simply be processed in a single group. If cross-validation is to be performed, then the data is automatically partitioned into folds, such that each fold is appropriately divided into training and testing sets.

If meta learning experimentation is to be performed then, as of ACE 2.0, some of the input data is randomly reserved as a final publication set to be used in evaluating the best-performing dimensionality reduction and classification algorithm pair. The non-publication data is used in the cross-validation experiments that are performed in order to evaluate each algorithm pair. The algorithm pair that has the highest cross-validation performance is then used to train a new model that is trained on the entirety of the data that was previously used in the cross-validation experiments. Finally, the resulting model is validated on the reserved publication set, in order to test for overfitting by the meta learning process.

There are a number of ACE parameters and pre-processing options that may be selected by users, including:

- A maximum may be imposed on the percentage of instances that are permitted to belong to any single class in the data used for training and validation. This can be useful in helping to ensure that, in cases where far more instances are available belonging to one class than another, the classifiers do not train to a local performance maximum where instances are never classified to classes that are rare in the training data, or that the ability to classify to rare classes is not undervalued during evaluation. This option causes excess instances belonging to overly common classes to be randomly filtered out from the training, validation and publication datasets.

- A maximum may be imposed on the number of instances that may belong to each class in the training data. Instances beyond this number are removed from the training data. This can be useful for similar reasons as imposing a maximum on the class ratio, as described above, and can also be advantageous in speeding up training when very large datasets are available.

- The instances may be processed in the order that they appear in the input files, or they may be ordered randomly. The order in which training instances are processed can have an influence on the models trained by certain classification algorithms, such as feedforward neural networks.
- The number of folds to use during cross-validation.
- The maximum number of features beyond which exhaustive dimensionality reduction searches will not be experimented with.
- Whether basic or verbose reports are to be generated.

### 6.4.2 Information output by ACE

The output of ACE depends on which of the four basic tasks described in Section 6.4.1 is requested by the user. If basic training of a model is to be performed using a specified classification algorithm, then the output is simply the trained model itself saved as a Weka Java serialized object. Although there are long-term portability disadvantages with the use of such serialized objects (see Section 7.2.6), the classifier implementations used by ACE are all part of the Weka library, and Java serialized objects are the methodology used by Weka to save trained models. It is necessary for ACE to conform to this practice in order for the models that it trains to be distributable to others who may wish to use the trained models directly with Weka.

Users who wish to use a previously trained model to classify a set of instances may have ACE output predicted class labels for each instance to either ACE XML Instance Label files or Weka ARFF files. Additional useful information may also be printed to standard out. The nature of this additional information depends on whether model classifications are provided along with the trained model. If such model classifications are available, then the identifier of each instance is specified by ACE along with its predicted classes and model classes. Instances whose predicted classes do not correspond to their model classes are marked with asterisks, and summary statistics are provided indicating the total number of classifications and the percentage of predicted class labels that match the corresponding model class labels. If no model classifications are available, then the predicted class labels of each instance are simply listed. This additional information is provided primarily for quick and easy access during debugging, however, and it is the ACE XML or Weka ARFF files mentioned above that are intended as the primary output.

The basic output produced when cross-validation is performed consists of the classification success rate for each fold and a confusion matrix for each fold. Overall statistics are also provided, including the average success rate across folds, the standard deviation of the success rates across folds, a confusion matrix averaged across folds, the processing time and the name of the dimensionality reduction and classification algorithm configuration that were applied at the user's behest. Users may also select the verbose output option, in which case it is also indicated for each fold which instances were used for testing and which were used for training, as well as the predicted versus model class labels for each test instance.

As one might expect, the greatest volume of user feedback is produced when meta learning experimentation is performed. As noted above, ACE begins the experimentation process by generating several different feature subsets, one for each dimensionality reduction algorithm. Cross-validation is then performed once for each classification algorithm on each such feature subset. ACE generates a separate report for each such feature subset, specifying the features selected (or their projections, if appropriate) as well as, for each classification algorithm, the cross-validation average success rate, the standard deviation of the success rate across folds and the processing time. The best performing algorithm for the feature subset is also highlighted, and its confusion matrix is provided. If the *verbose output* option is selected, then detailed information is also provided on the he dimensionality reduction, with content that depends on the particular algorithm.

A summary meta learning experimentation report is also generated, indicating the classifier and dimensionality reduction algorithm pair that achieved the highest average cross-validation success rate. Its cross-validation performance during experimentation is repeated from its corresponding feature subset report, for ease of reference. More significantly, the classification results of the selected algorithm pair applied to the reserved publication set are also reported. A comparison of the classification results on the publication set with the cross-validation results from the meta learning comparison experiment can help to indicate if the meta learning algorithm overfitted the available data and, if not, what kind of performance can be anticipated on novel instances. If the *verbose output* option is selected, then the predicted and model class labels are indicated for each instance in the publication set, along with detailed information regarding the dimensionality reduction performed.

### 6.4.3 Algorithms used by ACE

The following algorithms are packaged with ACE:

- Dimensionality reduction algorithms:
  - Principal component analysis
  - Genetic search
  - Exhaustive search
- Base classifier algorithms:
  - C4.5 decision tree classifier
  - Naïve Bayes classifier
  - K-nearest neighbour classifier
  - Backpropagation neural network classifier
  - Support vector machine classifier
- Classifier ensemble algorithms:
  - Adaboost
  - Bagging

These particular algorithms were chosen because they are all well established and have a variety of relative advantages and disadvantages.

As mentioned above, ACE makes use of the Weka API, so the Weka implementations of all of these algorithms are used. It is a relatively simple matter to add additional algorithms to ACE once they are implemented in Weka. A reference to a new Weka algorithm need only be added to a single ACE Java class, after which the new algorithm's functionality will automatically become accessible to all other aspects of the software. This means that those wishing to experiment with algorithms beyond those already packaged with ACE have available to them the extensive library of algorithms already implemented as part of Weka. This library is also constantly expanding, as the Weka development community is quite active.

ACE not only performs experiments during meta learning with different classification algorithms, but also automatically experiments with different hyperparameterizations of some of these algorithms. The particular hyperparameterizations depend on the individual algorithms. For example, experiments with k-NN are automatically performed with a variety of values of *k,* as well as versions that are unweighted, weighted by distance and weighted by similarity.[162] To provide another example, the naïve Bayes' classifier is applied using both a basic Gaussian kernel and a kernel density estimator. By default, the classifier ensemble algorithms are seeded with decision tree stubs and the genetic search dimensionality reduction uses a naïve Bayes' classifier to evaluate chromosome fitness.

### 6.4.4 Administering jMIR projects

ACE can be used to organize and administrate jMIR projects. ACE includes functionality for generating, modifying and accessing jMIR projects via ACE XML Project and ZIP files (see Section 7.11.1), as well as for viewing, editing and saving component ACE XML Feature Value, Instance Label, Feature Description and Class Ontology files. ACE can also be used to translate Weka ARFF files to ACE XML, and vice versa.

jMIRUtilities (see in Section 8.8) is also a useful tool performing administrative tasks relating to integrating jMIR components with each other and with external sources of data, such as iTunes XML files. jMIRUtilities also automates some tasks that can also be performed with ACE as well. For example, the ACE GUI can be used to annotate instances with class labels, but jMIRUtilities' batch instance labelling tool can do so much faster if there are many instances to label.

[162] The Weka implementation of these weighting metrics are described at weka.sourceforge.net/doc.stable/weka/classifiers/lazy/IBk.html.

### 6.4.5 Advantages of ACE over alternative machine learning frameworks

As discussed in Section 6.3, there are a number of alternative machine learning frameworks, many of which are powerful and high-quality pieces of software. Why, then, should one use ACE instead of one of these alternatives?

One of the most important advantages of ACE is that it is one of the few frameworks that implement automated experimental meta learning. A related advantage of ACE is that its interface does not presuppose any background in machine learning on the part of users. Although several alternative systems do have excellent interfaces, including some with attractive and intuitive graphical user interfaces, like RadidMiner, these interfaces are typically designed for people who have at least some basic expertise in machine learning, and can require significant effort to learn and understand for users without such a background. ACE, in contrast, allows users to simply specify input files and have the system generate results. There is no need for users to specify variables such as the dimensionality reduction algorithms to use, the classification algorithms to use or the data partitioning approach to use unless, of course, they wish to. The simplicity of this approach helps to make the software much more accessible to general users.

Furthermore, ACE is implemented entirely in Java and does not require any external libraries beyond those that are packaged with it. This platform independence can significantly reduce installation difficulties, particularly compared to C or C++-based systems, for example, where installation can often entail lengthy and difficult debugging in order to resolve linking problems.

An additional advantage of ACE is that it is open-source, free and does not require any proprietary software to run. Although ACE is certainly not the only framework like this, many alternatives do in fact require the purchase of proprietary software such as MatLab in order to be used even if they are free themselves.

Perhaps the strongest advantage of ACE, however, is that it is designed to meet the particular needs of music information retrieval research. ACE is, in fact, the only meta learning framework designed specifically for music research. Music has a number of particularities and special problems that often fail to be properly and conveniently addressed by general-purpose machine learning frameworks.

Many of the limitations of general-purpose frameworks become apparent even at the representational level, as the file formats used by many alternative systems are incapable of representing types of information that can be essential to music research. The ACE XML file formats and the ACE data structures and methods that can be used to process the data stored in them are thus among the strongest advantages of ACE relative to alternative systems. Although the advantages of ACE XML with respect to representing data that is relevant to automatic music classification are discussed in much more detail in Chapter 7, some of highlights that are particularly relevant to ACE are listed below:

- The ability to associate multiple classes with a single instance. This is essential to many central MIR research areas, such as genre or mood classification.
- The ability to represent class labels and feature values for potentially overlapping sub-sections of instances as well as for instances as a whole. This is fundamental to dealing with audio, which is typically windowed during feature extraction, as well as for music segmentation. It can be very useful to maintain the logical relationship between a window of features and the recording that it was extracted from, rather than simply treating each window as an independent instance, as most classification frameworks do.
- The ability to maintain logical groupings between multi-dimensional features. Many important MIR-related features are multidimensional, such as MFCCs and beat histograms, and there can be important classification advantages to maintaining an association between the values of each such feature. Most alternative frameworks, in contrast, simply treat each value of a feature vector or array simply as an entirely separate feature.
- The ability to represent structured class ontologies. This information can be taken advantage of by classification algorithms such as hierarchical classifiers, for example. This can also be used to improve classifier training by enabling misclassifications between dissimilar classes to be penalized more severely than misclassifications between similar classes. This ability to take into account the seriousness of different misclassification can also be used to more fully compare

algorithm evaluations during meta learning. Most alternative frameworks do not provide any way of representing relationships between different classes, however.

- The ability to maintain various kinds of metadata about instances in a way that is packaged directly with the instances.

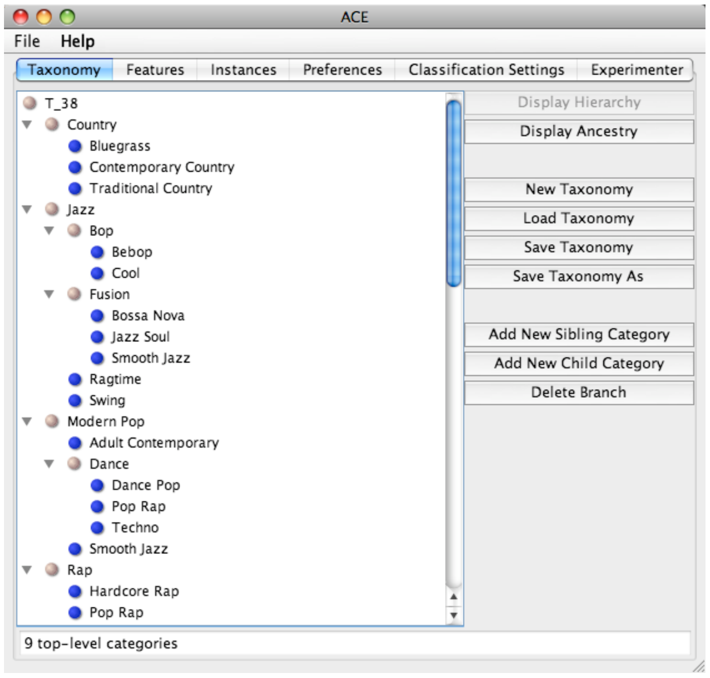
## 6.5 ACE's interface

ACE's functionality can be accessed via three different interfaces:

- **Command-line:** This is currently the primary means of using ACE as a meta learning and classification application. Training, instance classification, cross-validation and meta learning experimentation functionality can all be accessed via the command-line, and a number of settings and parameters, as discussed in Section 6.4, can be specified as well. Instructions on how to use the command-line are specified in the ACE 2.0 manual, and highlights can also be accessed by running ACE with the *-help* flag. The command-line interface is particularly useful for performing batch processing using various scripting tools or for quickly repeating experiments, with or without minor parameter variations. Very large datasets can take days or even weeks to process, and the automation of batch processing removes the need for human supervision during this period. The command-line is the only way to access ACE 1.0's core functionality, and is also improved significantly in ACE 2.0.

- **Graphical User Interface:** A command-line interface can be an obstacle to less technically oriented users, so efforts are ongoing to implement a GUI to facilitate access to ACE. As of the time of this writing, ACE 2.0's GUI may be used to display, edit and save ACE XML 1.1 files as well as perform various ACE XML Project and ZIP utility functions. It cannot yet be used to access ACE's core machine learning functionality, such as meta learning experimentation, but efforts are ongoing to implement this functionality. Figures 6.6 to 6.9 illustrate some of the GUI components that are already implemented. More details are available in the ACE 2.0 manual.

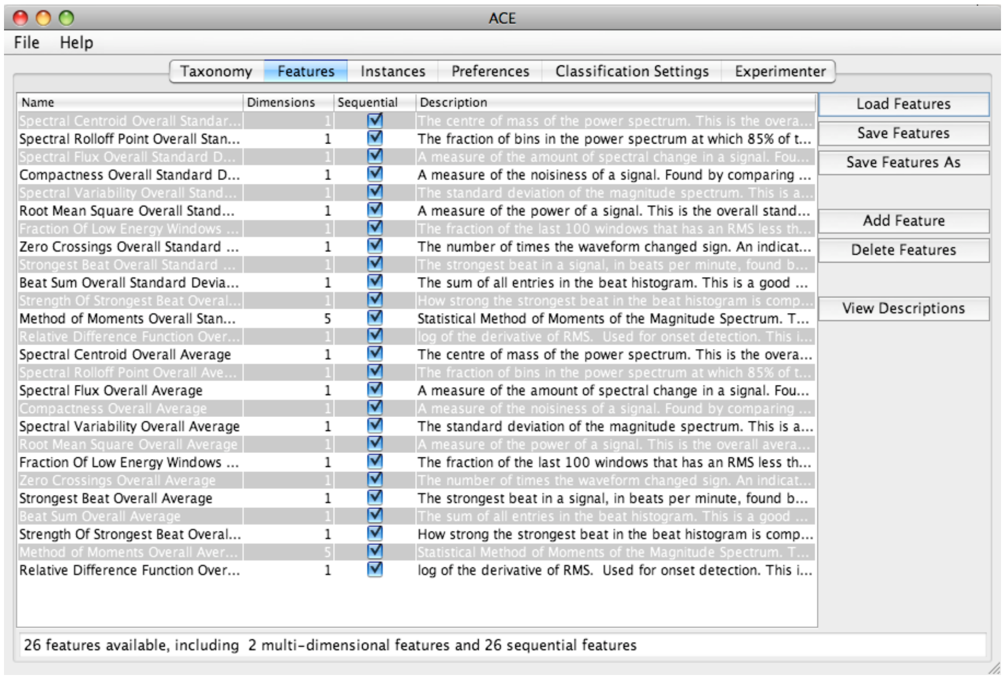
- **Java API:** ACE 2.0 is designed to be easy to integrate into other software frameworks, and its architecture is explicitly formulated to facilitate both this and extensibility. The code itself is open-source, and is well documented. As can be seen in Figure 6.10, all of ACE 2.0's functionality can be accessed via a *Coordinator* class. Tasks associated with dimensionality reduction, model training, instance classification, cross-validation and meta learning experimentation are each modularly segregated into, respectively, the *DimensionalityReducer, Trainer, InstanceClassifier, CrossValidator,* and *Experimenter* classes. Although these can certainly be extended by developers if desired, those wishing to simply incorporate ACE as is into their own systems never need to directly access them, or even be aware of their existence, as their functionality is entirely abstracted into the *Coordinator* class. More details are available in the ACE 2.0 manual and in the code itself.



- T_38
  - Country
    - Bluegrass
    - Contemporary Country
    - Traditional Country
  - Jazz
    - Bop
      - Bebop
      - Cool
    - Fusion
      - Bossa Nova
      - Jazz Soul
      - Smooth Jazz
    - Ragtime
    - Swing
  - Modern Pop
    - Adult Contemporary
    - Dance
      - Dance Pop
      - Pop Rap
      - Techno
    - Smooth Jazz
  - Rap
    - Hardcore Rap
    - Pop Rap

9 top-level categories

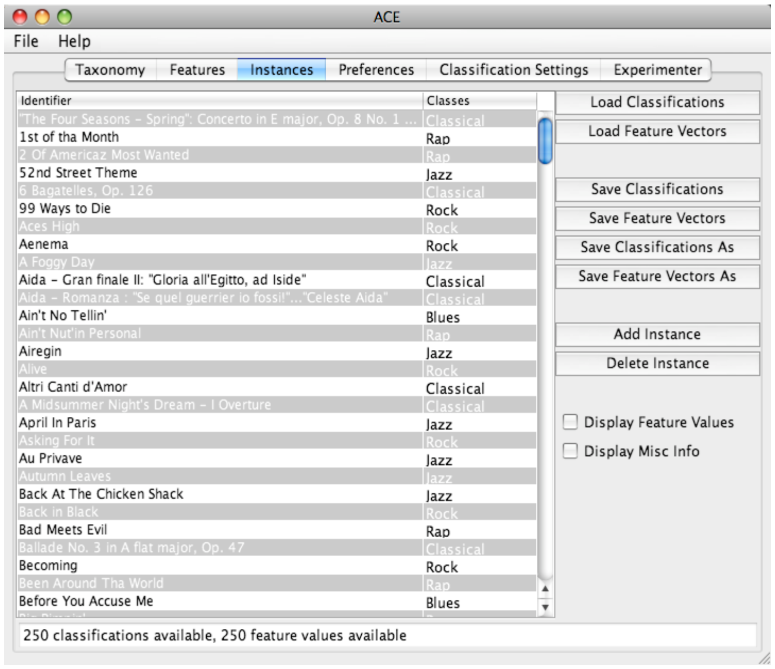
**Figure 6.6:** Sample genre taxonomy displayed in the *Taxonomy Pane* of the ACE 2.0 GUI. Users are able to create, load, view, edit and save class ontologies using this pane.



| Name | Dimensions | Sequential | Description |
|---|---|---|---|
| Spectral Centroid Overall Standar... | 1 | ✓ | The centre of mass of the power spectrum. This is the overa... |
| Spectral Rolloff Point Overall Stan... | 1 | ✓ | The fraction of bins in the power spectrum at which 85% of t... |
| Spectral Flux Overall Standard D... | 1 | ✓ | A measure of the amount of spectral change in a signal. Fou... |
| Compactness Overall Standard D... | 1 | ✓ | A measure of the noisiness of a signal. Found by comparing ... |
| Spectral Variability Overall Stand... | 1 | ✓ | The standard deviation of the magnitude spectrum. This is a... |
| Root Mean Square Overall Stand... | 1 | ✓ | A measure of the power of a signal. This is the overall stand... |
| Fraction Of Low Energy Windows ... | 1 | ✓ | The fraction of the last 100 windows that has an RMS less th... |
| Zero Crossings Overall Standard ... | 1 | ✓ | The number of times the waveform changed sign. An indicat... |
| Strongest Beat Overall Standard ... | 1 | ✓ | The strongest beat in a signal, in beats per minute, found b... |
| Beat Sum Overall Standard Devia... | 1 | ✓ | The sum of all entries in the beat histogram. This is a good ... |
| Strength Of Strongest Beat Overal... | 1 | ✓ | How strong the strongest beat in the beat histogram is comp... |
| Method of Moments Overall Stan... | 5 | ✓ | Statistical Method of Moments of the Magnitude Spectrum. T... |
| Relative Difference Function Over... | 1 | ✓ | log of the derivative of RMS.  Used for onset detection. This i... |
| Spectral Centroid Overall Average | 1 | ✓ | The centre of mass of the power spectrum. This is the overa... |
| Spectral Rolloff Point Overall Ave... | 1 | ✓ | The fraction of bins in the power spectrum at which 85% of t... |
| Spectral Flux Overall Average | 1 | ✓ | A measure of the amount of spectral change in a signal. Fou... |
| Compactness Overall Average | 1 | ✓ | A measure of the noisiness of a signal. Found by comparing ... |
| Spectral Variability Overall Average | 1 | ✓ | The standard deviation of the magnitude spectrum. This is a... |
| Root Mean Square Overall Average | 1 | ✓ | A measure of the power of a signal. This is the overall avera... |
| Fraction Of Low Energy Windows ... | 1 | ✓ | The fraction of the last 100 windows that has an RMS less th... |
| Zero Crossings Overall Average | 1 | ✓ | The number of times the waveform changed sign. An indicat... |
| Strongest Beat Overall Average | 1 | ✓ | The strongest beat in a signal, in beats per minute, found b... |
| Beat Sum Overall Average | 1 | ✓ | The sum of all entries in the beat histogram. This is a good ... |
| Strength Of Strongest Beat Overal... | 1 | ✓ | How strong the strongest beat in the beat histogram is comp... |
| Method of Moments Overall Aver... | 5 | ✓ | Statistical Method of Moments of the Magnitude Spectrum. T... |
| Relative Difference Function Over... | 1 | ✓ | log of the derivative of RMS.  Used for onset detection. This i... |

26 features available, including  2 multi-dimensional features and 26 sequential features

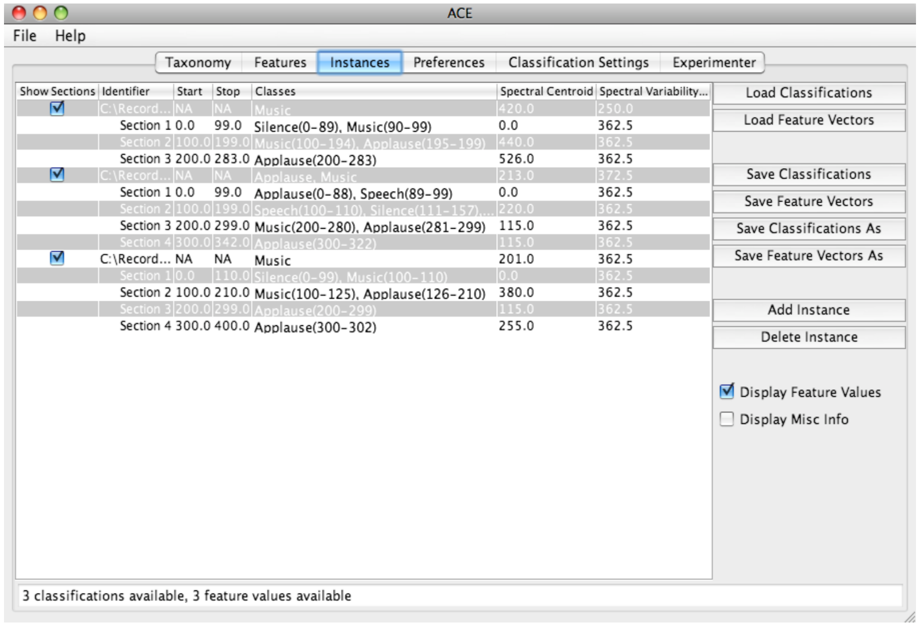
**Figure 6.7:** Sample feature metadata displayed in the *Feature Descriptions Pane* of the ACE 2.0 GUI. This information relates to features themselves, not to actual feature values extracted from features. Users are able to create, load, view, edit and save feature metadata using this pane.



| Identifier | Classes |
|---|---|
| "The Four Seasons – Spring": Concerto in E major, Op. 8 No. 1 ... | Classical |
| 1st of tha Month | Rap |
| 2 Of Americaz Most Wanted | Rap |
| 52nd Street Theme | Jazz |
| 6 Bagatelles, Op. 126 | Classical |
| 99 Ways to Die | Rock |
| Aces High | Rock |
| Aenema | Rock |
| A Foggy Day | Jazz |
| Aida – Gran finale II: "Gloria all'Egitto, ad Iside" | Classical |
| Aida – Romanza : "Se quel guerrier io fossi!"..."Celeste Aida" | Classical |
| Ain't No Tellin' | Blues |
| Ain't Nut'in Personal | Rap |
| Airegin | Jazz |
| Alive | Rock |
| Altri Canti d'Amor | Classical |
| A Midsummer Night's Dream – I Overture | Classical |
| April In Paris | Jazz |
| Asking For It | Rock |
| Au Privave | Jazz |
| Autumn Leaves | Jazz |
| Back At The Chicken Shack | Jazz |
| Back in Black | Rock |
| Bad Meets Evil | Rap |
| Ballade No. 3 in A flat major, Op. 47 | Classical |
| Becoming | Rock |
| Been Around Tha World | Rap |
| Before You Accuse Me | Blues |

250 classifications available, 250 feature values available

**Figure 6.8:** Sample instance class labels displayed in the *Instances Pane* of the ACE 2.0 GUI. Users are able to load, view, edit and save feature class labels and other information using this pane.



| Show Sections | Identifier | Start | Stop | Classes | Spectral Centroid | Spectral Variability... |
|---|---|---|---|---|---|---|
| ✓ | C:\Record... | NA | NA | Music | 420.0 | 250.0 |
| | Section 1 | 0.0 | 99.0 | Silence(0-89), Music(90-99) | 0.0 | 362.5 |
| | Section 2 | 100.0 | 199.0 | Music(100-194), Applause(195-199) | 440.0 | 362.5 |
| | Section 3 | 200.0 | 283.0 | Applause(200-283) | 526.0 | 362.5 |
| ✓ | C:\Record... | NA | NA | Applause, Music | 213.0 | 372.5 |
| | Section 1 | 0.0 | 99.0 | Applause(0-88), Speech(89-99) | 0.0 | 362.5 |
| | Section 2 | 100.0 | 199.0 | Speech(100-110), Silence(111-157),... | 220.0 | 362.5 |
| | Section 3 | 200.0 | 299.0 | Music(200-280), Applause(281-299) | 115.0 | 362.5 |
| | Section 4 | 300.0 | 342.0 | Applause(300-322) | 115.0 | 362.5 |
| ✓ | C:\Record... | NA | NA | Music | 201.0 | 362.5 |
| | Section 1 | 0.0 | 110.0 | Silence(0-99), Music(100-110) | 0.0 | 362.5 |
| | Section 2 | 100.0 | 210.0 | Music(100-125), Applause(126-210) | 380.0 | 362.5 |
| | Section 3 | 200.0 | 299.0 | Applause(200-299) | 115.0 | 362.5 |
| | Section 4 | 300.0 | 400.0 | Applause(300-302) | 255.0 | 362.5 |

3 classifications available, 3 feature values available

**Figure 6.9:** Sample feature values as well as class labels for an audio segmentation task displayed in the *Instances Pane* of the ACE 2.0 GUI. This figure demonstrates how the *Instances Pane* can be used to display feature values and class labels both for overall instances and for subsections of instances.

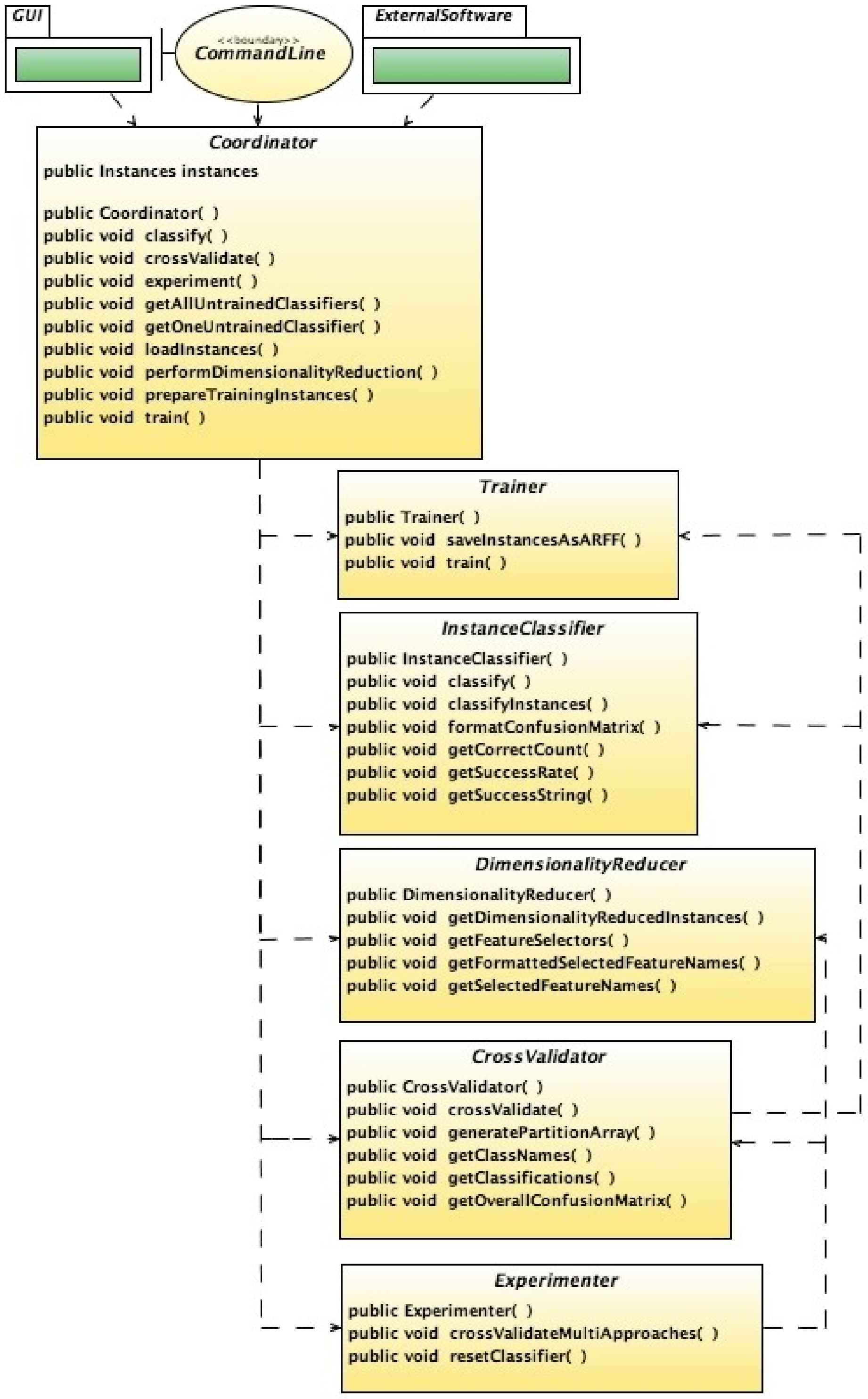

GUI
<<boundary>>
CommandLine
ExternalSoftware
Coordinator
public Instances instances
public Coordinator( )
public void classify( )
public void crossValidate( )
public void experiment( )
public void getAllUntrainedClassifiers( )
public void getOneUntrainedClassifier( )
public void loadInstances( )
public void performDimensionalityReduction( )
public void prepareTrainingInstances( )
public void train( )
Trainer
public Trainer( )
public void saveInstancesAsARFF( )
public void train( )
InstanceClassifier
public InstanceClassifier( )
public void classify( )
public void classifyInstances( )
public void formatConfusionMatrix( )
public void getCorrectCount( )
public void getSuccessRate( )
public void getSuccessString( )
DimensionalityReducer
public DimensionalityReducer( )
public void getDimensionalityReducedInstances( )
public void getFeatureSelectors( )
public void getFormattedSelectedFeatureNames( )
public void getSelectedFeatureNames( )
CrossValidator
public CrossValidator( )
public void crossValidate( )
public void generatePartitionArray( )
public void getClassNames( )
public void getClassifications( )
public void getOverallConfusionMatrix( )
Experimenter
public Experimenter( )
public void crossValidateMultiApproaches( )
public void resetClassifier( )

**Figure 6.10:** Highlights of ACE 2.0's class structure. A number of methods and fields are omitted for the sake of brevity, as are all method parameters. All of ACE's functionality can be accessed via the *Coordinator* class. Arrows indicate interactions between classes.

## 6.6 Improvements in ACE 2.0

It is useful to briefly summarize the differences between ACE 1.0 and ACE 2.0, since they are both publicly available, and Sections 6.4 and 6.5 include information relating to both versions of ACE. ACE 1.0 is the current stable release and ACE 2.0 is the current developer release. ACE 2.0 includes all of the functionality of ACE 1.0, but is not as fully tested, and still has certain features under development. ACE 1.0 was designed and implemented by Cory McKay, and ACE 2.0 was designed by Cory McKay and Jessica Thompson and implemented by Jessica Thompson (Thompson et al. 2009). The essential differences between the two versions are:

- **Improved cross-validation:** ACE 1.0 relied entirely on Weka's own classes to perform cross-validation. Unfortunately, the portion of the Weka API dealing with cross-validation is not designed with the needs of experimental meta learning in mind, and hides certain important information, such as the variance of the classification success rate across folds[163], confusion matrices for individual folds, details on which instances are assigned training and testing roles in each fold and class predictions for individual cross-validation test instances. ACE 2.0 therefore has its own cross-validation implementation, with the result that expanded statistics are available, including all of the information noted above. ACE 2.0 also automatically reserves a publication set for final meta learning validation and makes it possible to ensure that multiple cross-validation experiments with different classification algorithms can all use the same training and testing data partitioning, something that greatly facilitates statistical hypothesis testing.
- **More detailed reports:** In addition to the greatly expanded information reported in cross-validation reports, as described above, ACE 2.0 also provides expanded information in the meta learning reports as well, particularly when the *verbose* option is selected by the user.

[163] Although the Weka Experimenter does in fact calculate the variance, this functionality is not accessible from the Weka API. Third-party software using the Weka code base, such as ACE, must rely on the Weka *Evaluation* class, which does not make the variance across folds accessible.

- **ACE ZIP files:** ACE 2.0 includes functionality for creating, parsing, saving and processing ACE XML ZIP files (see Section 7.11.1), as well as uncompressed ACE XML files. ACE 1.0 is not ACE XML ZIP compatible.
- **Redesigned API:** The class structure and API have been completely redesigned in order to facilitate the use of ACE in third party software and to improve ease of extensibility.
- **Improved command-line interface:** The command-line interface has been entirely redesigned. It allows users to specify parameters in greater derail, and is also clearer and more intuitive.
- **Graphical user interface:** ACE 2.0 includes a GUI that can be used to create, parse, edit and save ACE XML files and the related data. Ongoing efforts are also being made to provide access to ACE's meta learning functionality as well, which is currently only available via the command-line and the API.
- **Manual:** A detailed HTML-based manual is available for ACE 2.0 that details the command-line interface, the GUI and the code structure.

## 6.7 Summary of original contributions

ACE is a powerful dedicated pattern recognition application and a code library that can be embedded in other applications. Although there are a variety of other excellent machine learning frameworks, as discussed in Section 6.3, ACE is unique in that it is the only dedicated meta learning framework designed to meet the special needs of music research, as outlined in Section 6.4.5.

The meta learning functionality offered by ACE can be particularly useful in evaluating the effectiveness of new features or classification algorithms. This is especially helpful in the field of music information retrieval, where new techniques are constantly being developed, and need do evaluated relative to one another.

One of the most important characteristics of ACE is that it is designed to help overcome usability barriers for users with valuable musical knowledge but little or no machine learning background. Such barriers have traditionally largely prevented experts such as music theorists, musicologists and music librarians from using machine learning in their research, for example. ACE’s meta learning makes powerful and effective machine learning accessible even to researchers with no training in machine learning.

Section 6.2 also includes a brief but still relatively detailed introduction to machine learning that is intended specifically for researchers who may have no machine learning background but who have an interest in learning more about the nuts and bolts of machine learning from an applied and not overly technical perspective. Of course, the information presented in this section is in no way necessary for using ACE.

## *6.8 Future research*

### 6.8.1 ACE’s interface

Although the ACE 2.0 GUI is currently functional with respect to parsing, viewing, editing and saving jMIR projects and ACE XML files, functionality for performing meta learning experiments and other machine learning tasks is currently only accessible via ACE’s command-line interface or Java API. One of the highest priority goals for future research is to make all of ACE’s functionality accessible via the GUI, including functionality for viewing customized result summaries and for using the GUI to specify detailed preference settings with respect to both the interface itself and the processing back end.

### 6.8.2 Additional machine learning and pre-processing algorithms

Another priority is the addition of further machine learning algorithm implementations to ACE. These will include the full range of supervised learning algorithms currently available in Weka, as well as Weka implementations of other algorithms that are not yet available in Weka.

Sequential classification algorithms, such as recurrent neural networks and hidden Markov models, are of particular interest. Melody, harmony and form, for example, are essentially sequential concepts, so the addition of sequential algorithms will open up many possibilities for ACE.

There are also plans to add unsupervised algorithms to ACE, something that will be facilitated by the fact that several such algorithms are already implemented in Weka. Although unsupervised learning is less ideally suited to tasks such as genre or mood classification than supervised learning, unsupervised learning can be very useful for exploratory clustering research or for similarity-based applications like playlist generation or music recommendation. Unsupervised learning techniques can also be used to automatically generate class ontologies.

Experimentation with more classifier ensemble variants, beyond the bagging and boosting approaches already implemented in ACE, could help improve classification performance. Ensembles oriented towards structured learning, such as hierarchical learning, could be particularly useful. Tools for constructing blackboard systems could also be built into ACE, in particular systems that can integrate expert knowledge about problem domains with machine learning.

Another important improvement would be the addition of more pre-processing and post-processing functionality, such as various data thinning algorithms. This could improve results and speed up training, as well as to make the nuances of meta learning results more apparent to users.

### 6.8.3 Distributed processing and long-term projects

One of the central difficulties that must be overcome in automatic music classification is that there can be huge quantities of data to process, especially when many features are extracted or when very large data sets are available. The result is that learning models and classifying instances can potentially take inconveniently long amounts of time, even on the fastest computers. This is particularly problematically when experimental meta learning is performed, as it can require many computationally intensive training and testing experiments.

There are therefore plans to utilize distributed computing to spread out the computational burden. Meta learning experiments are relatively easy to separate into independent tasks that may be performed in parallel, something that makes ACE particularly suitable for distributed computing.

A number of existing distributed computing technologies may be taken advantage of in order to achieve this. Grid Weka (Khoussainov et al. 2004) is particularly attractive because it is based on the Weka framework used by ACE. The most likely solution, however, will be to use ACE with SEASR (Llorà 2008), which includes a Java-based infrastructure for distributed computing. SEASR is a central part of the NEMA[164] project, of which jMIR is a part. Both Grid Weka and SEASR allow computation to be distributed amongst either multi-purpose workstations or dedicated machines, and both are compatible with a wide range of hardware and operating system configurations, thus effectively making distributed computing available to anyone with access to a typical computing lab. There are many computers in academic institutions, such as libraries, that are often idle. ACE could therefore take advantage of these computers when they are not otherwise in use to dramatically and cheaply speed up the processing of large projects.

Related functionality will also be built into ACE allowing users to specify maxima on the total time that ACE has to perform meta learning experiments on a given data set. These limits on how long the system has to arrive at a solution will result in ACE initially pursuing the most promising approaches, based on past experiments with similar data. ACE will then output the best-performing algorithms that it is able to find in the time available, and then later revisit the problem passively when no other active projects are being processed.

The first step in implementing distributed computing will simply be to modify ACE so that it can assign individual meta learning component experiments to different computers. The next step will be to set up an ACE server to keep a record of the progress and performance of all ACE experiments run on any given user's cluster. This information could then be used to calculate performance estimates on similar ACE experiments run in the future when there is insufficient time tor run all of the experiments that might ideally be desired. When no other ACE projects are active, ACE could also use this server to reactivate old inactive projects in order to further improve results, as noted above.

[164] nema.lis.uiuc.edu

# 7. ACE XML: File formats for expressing MIR data

## *7.1 Overview of MIR data mining file formats and ACE XML*

Much of the cross-institutional efficiency of MIR research hinges on the ability of researchers to share data effectively with one another. Information such as ground-truth annotations, for example, can be very expensive to produce, and a great deal of repeated effort is avoided if researchers are able to share such information efficiently. Similarly, training and testing datasets themselves can be expensive to acquire, and since they cannot typically be distributed directly because of legal limitations, the ability to share representative feature values efficiently can be very valuable. The communication of more abstract information, such as class-label ontologies or characteristics of features themselves, can also be very useful.

Well-constructed, flexible and expressive standardized file formats are essential for distributing all of these types of information efficiently and fully. Furthermore, in order to encourage adoption as a standard, such file formats must be simple for both humans and machines to understand, parse and write. The absence of such standardized formats can pose an obstacle to the sharing of research information, with the result that each lab has a greater tendency to generate its own in house data, which results in both wasteful repeated effort and, in general, lower quality data.

Although there are a variety of widely used general-purpose data mining and classification file formats in existence, none of them meet the very specific needs of MIR research. The Weka ARFF format (Witten and Frank 2005), for example, is currently the de facto standard in MIR research, but as is shown in Sections 7.3.2 and 7.4, it has severe restrictions with respect to the requirements of realistic MIR research. Formats such as ARFF impose serious limitations on the types of information that can be represented and, accordingly, on the quality of research that can be performed based on this information. To give just one example, most such formats, including ARFF, only allow instances to be labelled with just one class label at a time, something that fundamentally limits the sophistication and realism of research in areas such as genre classification.

Sections 7.2 and 7.3 respectively review existing file format technologies and the particular formats that are currently the most prominent in MIR research. Section 7.4

presents a critical analysis of the limitations of existing formats, and introduces a corresponding list of design priorities that can be used to guide the development of new file formats designed specifically for use in music data mining and classification research.

The primary focus of this chapter is on the original ACE XML file formats. These formats were designed based on the design priorities emphasized in Section 7.4, and are intended for use in music data mining and classification research of any kind. Possible applications of these formats include genre classification, artist classification, track segmentation, pitch tracking, instrument identification and so on. The ACE XML formats may be used equivalently well with respect to audio, symbolic and cultural data.

ACE XML will play a role in the NEMA (Networked Environment for Music Analysis) project,[165] a large-scale multinational and multidisciplinary effort to create a general music information processing infrastructure. NEMA is funded by the Scholarly Communications program of the Andrew W. Mellon Foundation, and involves research groups from McGill University, University of Illinois at Urbana-Champaign, University of Southampton, University of Waikato and Goldsmiths and Queen Mary at University of London.

ACE XML is the native format used by all jMIR components to communicate with one other.[166] The ACE project also includes a general API for parsing, writing and processing ACE XML files so that ACE XML functionality can be easily incorporated into other software as well. The ACE GUI prototype also includes functionality for manually generating, displaying, editing and saving ACE XML files.

There are four primary types of ACE XML files, which may be used individually or integrated together:

- **Feature Value:** These files express feature values extracted from specific instances.
- **Feature Description:** These files express abstract information about features. To give a few examples, this format can be used to express specific details about the processes used to extract feature values that are expressed in Feature Value files,

[165] nema.lis.uiuc.edu
[166] jMIR components can also read and write ARFF for compatibility reasons, although it is recommended that ACE XML be used instead.

to publish information about the features that can be extracted by a new feature extraction application, to express updates to existing feature extraction algorithms, and so on.

- **Instance Label:** These files express labelled annotations of specific instances. This format can be used to notate ground-truth labels or the predicted labels output by a classification system, for example.
- **Class Ontology:** These files express relationships between different classes. This can be used to simply specify candidate class labels, or for more sophisticated purposes such as expressing hierarchical taxonomical relationships between classes that can be taken advantage of by specialized machine learning techniques.

There are two versions of ACE XML, namely ACE XML 1.1 and ACE XML 2.0. Version 1.1 (McKay et al. 2005) is the stable version that is currently implemented in all of the jMIR components, including the ACE API. For the purposes of this publication, 1.1 is the official version of ACE XML. Section 7.5 describes the four ACE XML 1.1 formats in general, and Sections 7.6 to 7.9 focus on each of them individually. The ACE API is discussed in Section 7.10.

The newer ACE XML 2.0 is ultimately intended for candidacy as a standardized format for MIR research in general, without any inherent links to jMIR. It builds upon the ACE XML 1.1 formats to add even more expressivity and functionality. Prototypes for the updated versions of each of the four main ACE XML formats are introduced in Section 7.11. As discussed in Section 7.11.1, version 2.0 also includes the new ACE XML Project file and ACE XML ZIP file types, which can be used to more conveniently associate and potentially package different ACE XML files together. Additional potential future improvements are also discussed in Section 7.13.

The ACE XML 2.0 (McKay et al. 2009) formats are presented here for review and improvement by the MIR community at large. The final implementation of ACE XML 2.0 awaits amendment based on community feedback, so the ACE XML 1.1 formats, which are presently fully finalized and implemented, continue to serve as the standard formats used by the jMIR components at the time of this publication.

The on-line sample file appendix[167] also provides artificially generated samples of each of the ACE XML 1.1 and 2.0 formats in order to more clearly illustrate how the file formats can be used.

## *7.2 Background information*

This section provides background information on the principal fundamental concepts and technologies that are relevant to the representation of information related to automatic music classification research. The contents of this section are presented in order to ensure that the reader is familiar with the concepts that are necessary to fully appreciate and compare the ACE XML file formats outlined later in this chapter with the existing alternative formats.

### 7.2.1 ASCII and Unicode text files

The term *text file* refers to a simple kind of computer file that holds basic textual data. Text files conform to simple standards for representing text, and are thus highly portable. Most text files that are referred to as such are *plain text* formats, which is to say that they do not include any provisions for formatting codes other than very simple markers such as *end of line, end of file* and tab markers. The lack of formatting in plain text files is both an advantage and a disadvantage, in that this generally results in smaller files, but also results in less expressivity.

The primary advantage of text files is that they follow a simple standard that can be parsed on essentially any computer running essentially any operating system. Applications called *text editors* are typically used to read, write and edit text files, although word processors and many other types of applications are compatible with text files as well.

*ASCII (American Standard Code for Information Interchange)* is perhaps the most common text file format. It is widely used enough to be considered platform independent, and ASCII files are often given a *.txt* extension, although this extension is also sometimes used for other types of text files as well. ASCII files use one byte to encode each character, with one bit reserved as a parity bit to aid in the detection of data corruption. This means that 128 characters may be represented in ASCII, 33 of which are mostly

[167] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

obsolete control characters. The remaining 95 characters include the lower-case and upper-case letters, numbers and punctuation marks associated mainly with English and related European languages, as well as a few mathematical characters, accents and other miscellaneous characters.

This ASCII limit of 95 printable characters is much too small to deal with all of the characters that even an English speaker might wish to use, much less someone writing in a language that uses a different alphabet. As a result, there are many other standards, usually chosen based on the default locale setting on a user's computer. An example is the technically obsolete but still often used *ISO 8859-1* encoding used for many European languages.

Limiting the space needed to store a single character to a single byte was reasonable in the past when data storage and transmission was expensive. Such rationing is no longer as much of a priority, however, and as a result the much more expressive *Unicode* standard is replacing ASCII as the preferred standard.

The multilingual full Unicode standard includes more than 100,000 characters, and has proven to be very valuable in the internationalization of computer software. It has been adopted by the XML standard, by Java and by the Microsoft .NET framework, among many others.

There are a variety of Unicode encodings in existence. One of the most common of these is the variable-length *UTF-8* encoding, which uses one byte for all ASCII characters and up to three additional bytes for all other characters. UTF-8 has the significant advantage of being backwards-compatible with ASCII.

*UTF-16* is the most common alternative Unicode encoding. UTF-16 is also a variable-length encoding, and it uses up to four bytes. UTF-16 can sometimes be more space-efficient than UTF-8, but the reverse can also be true, depending on the particular characters that need to be encoded. Both UTF-8 and UTF-16 may be used to encode any Unicode character.

### 7.2.2 Escape characters and delimiter-separated values

Some applications make use of reserved combinations of characters to effectively add additional characters or formatting instructions to text files that are not directly permitted by their encodings. This is often done via the use of *escape characters,* such as a backslash, that indicate that the following character is to be interpreted in a special way. Although files that use this approach are of course still text files, they nonetheless lose some portability and human-readability, as the software or individual parsing the files must be aware of the rules governing these special codes in order to properly interpret them.

It is often desirable to store data in text files in some structured way, such as in the case of tables of values. One simple way of doing this is to use *delimiters,* which is to say special characters that are reserved to separate values, such as a list of names. Commas, tabs and end-of-line characters are three of the most commonly used delimiters. This approach is useful as a quick and easy solution, but once again involves a loss of portability, since it requires that parsing software be aware of the particular delimiters that are being used.

There are a variety of standardized delimited text file formats. The comma-delimited CSV standard is perhaps the best-known example.

### 7.2.3 XML

It is often desirable to be able to express textual data in structured ways that are more sophisticated and general than is possible with simple delimited text files. *XML (eXtensible Markup Language)* files, which are encoded in UTF-8 by default, provide one particularly flexible and convenient way of doing this.

XML is an example of a *markup language,* which is to say that it is an artificial language that uses annotations to impose structure and formatting on text. HTML is perhaps one of the most famous markup languages, due its role as the traditional format in constructing web pages. An essential difference between XML and HTML is that XML allows users to specify how data is to be structured, whereas HTML requires that data be formatted in rigidly pre-defined ways.

XML attempts to strike a balance between permitting data to be represented in such a way that it is conveniently structured for machine processing, and requiring that it be stored in simple text files in ways that are relatively easily human readable. The ability for humans to read XML files directly is augmented by functionality in many web browsing or text editing applications to display data stored in an XML files in an easily human-parsible way that is consistent with the structuring specified in the XML file, while at the same time hiding the infrastructure that specifies this structuring. Specialized software such as Altova XML Spy[168] provides even greater functionality in this respect.

XML data essentially consists of two parts: one that specifies the structuring and formatting that stored data must conform to, and the other storing this data using the specified infrastructure. The XML specification permits a variety of ways of performing this first task, the oldest, least powerful and simplest of which is known as a *Document Type Definition (DTD).* A DTD typically consists of a separate file or header preceding the actual data that is stored in an XML file. Alternative XML *schemas* allow added expressivity over DTDs, but at the cost of increased complexity.

The formatting of the data stored in XML files must conform to the rules laid out in the DTD or schema. There are many software packages and web services (e.g., validator.w3.org) that can be used to verify that this is the case for any given document. This is referred to as checking whether an XML document is *well-formed.*

The majority of XML documents consist of clauses of information denoted using *elements.* Each element has a *start tag* and an *end tag* as well as, often, some *content* between the tags. The tags indicate the kind of information that the content expresses, typically in the form of a field label, and the content indicates the value for the field.

For example, the element *<author_name>Alexander Solzhenitsyn</author_name>* includes start and end name tags, indicating that the content of the tags is an author's name, and the content itself specifies the name of the specific author, *Alexander Solzhenitsyn* in this case. Start and end tags of elements are always each contained within < and > signs, and end tags always have the same name as their matching start tag, but have a / sign added at their beginning.

Elements can be organized hierarchically, which is to say that the content of an element can itself contain one or more other elements. The elements that may appear and the rules governing their structuring are declared in the DTD or schema of each XML file. Figure 7.1 provides an example of how elements can contain other elements.

[168] www.altova.com

```
<my_books>
   <book>
      <title>Gulag Archipelago</title>
      <author_name>
            <first_name>Alexander</first_name>
            <last_name>Solzhenitsyn</last_name>
      </author_name>
   </book>
</my_books>
```

**Figure 7.1:** Sample hierarchically organized XML elements. This could be used, for example, to store a database of the books that one owns. The *my_books* element could be used to hold multiple *book* elements, each of which refer to one separate book, although only one is listed here. The *book* element contains a *title* element and an *author_name* element. The *author_name* element itself contains two further elements. One would expect there to be one *book* element for every book that is owned. The names of these particular kinds of elements and the relationships between them must be specified in a DTD header or other XML schema.

```
<my_books>
   <book staus="owned">
      <title>Gulag Archipelago</title>
      <author_name alive="no">
            <first_name>Alexander</first_name>
            <last_name>Solzhenitsyn</last_name>
      </author_name>
   </book>
</my_books>
```

**Figure 7.2:** A modified version of the *book* element from Figure 7.1. Two attributes are added, namely a *status* attribute for the *book* tag and an *alive* tag for the *author_name* tag. The *status* attribute could be used to indicate whether the book is *owned*, *read but not owned* or *not read*, for example, and the *alive* attribute would be used to indicate if the author is currently alive.

Elements may also contain *attributes* that provide further information. The rules governing the nature of the attributes and which elements may contain particular attributes are also specified in the DTD or other XML schema. Attributes are noted in the start tag of each element by following the name of the tag with a space, the name of the attribute, an equal sign and, finally, the value of the attribute enclosed within double quotes. Figure 7.2 shows how attributes might be used in an expanded version of the data shown in Figure 7.1.

In most cases, information expressed in attributes could alternatively be expressed using subordinate elements. For example, the *alive* attribute in Figure 7.2 could just have easily been a child element of *author_name,* just as *last_name* is. Although which approach one uses is mostly a matter of style, a general rule of thumb is to use an attribute if only restricted values are possible (e.g., yes or no) and to use an element if arbitrary values are possible. To return to the example if Figure 7.2, an alternative architecture might be to make *status* a child element, but leave *alive* as an attribute, as shown in Figure 7.3.

```
<my_books>
   <book>
      <title>Gulag Archipelago</title>
      <author_name alive="no">
             <first_name>Alexander</first_name>
             <last_name>Solzhenitsyn</last_name>
      </author_name>
      <status>owned</status>
   </book>
</my_books>
```

**Figure 7.3:** Alternative architecture to that of Figure 7.2, where the *status* field is now a field instead of an attribute.

As noted above, which elements are permitted and how they may be structured is defined in a DTD header or other schema. A DTD is essentially a statement that consists of *<!DOCTYPE fileype [...]>*, where *filetype* is a code that is chosen to identify the type of XML file that the DTD is defining (e.g., *my_books* might be used to identify a file format being used to store one's book collection). The *[...]* holds a separate declaration for each element and attribute type that is permitted. Each element declaration is a statement consisting of *<!ELEMENT elementname (elementdeclarationtype)>*. The *elementname* variable specifies the text of the start tag of the element and the *elementdeclarationtype* variable specifies which child elements are permitted, if any, which attributes are permitted, if any, and what kind of content data (e.g., character data) is permitted for the element. The order that the element declarations appear in the DTD overall and in each individual element declaration type constrain the order that the tags may be used in the document if it is to be well-formed. Figure 7.4 provides an example of a sample DTD.

```
<!DOCTYPE my_books [
   <!ELEMENT my_books (book+)>
   <!ELEMENT book (title, author_name, status?)>
   <!ELEMENT title (#PCDATA)>
   <!ELEMENT author_name (first_name, last_name)>
   <!ATTLIST author_name alive CDATA "yes">
   <!ELEMENT status (#PCDATA)>
   <!ELEMENT first_name (#PCDATA)>
   <!ELEMENT last_name (#PCDATA)>
]>
```

**Figure 7.4:** A possible DTD for the data formulation used in Figure 7.3. This could be stored in an external file or could be included as a header in the same file that stores the data about the book(s) themselves. Note that in this particular DTD there must be one or more books (because of the + sign), the status element is optional for each author (because of the *?* sign and the default value for the *alive* attribute is specified as *yes*.

The DTD also specifies how attributes are used, as can also be seen in Figure 7.4. The *<!ATTLIST elementname attributename1 CDATA "default1" attributename2 CDATA default2 ...>* declaration provides a list of all possible attributes for the *elementname* element, each of which is given the name *attributename#* and the default value of *default#,* which will be used if a particular entry does not specify a value for the attribute. The *CDATA* code simply means that the value for the attribute will be in the form of normal text.

Note that the *<!ATTLIST author_name alive CDATA "yes">* declaration in Figure 7.4 does not constrain the possible choices that may be used in a well-formed file, so an instance of the *alive* tag might be given reasonable values such as *yes, no* or *unknown* as well as unreasonable values such as *dafdasfd*. This flexibility may be desirable in some cases, but not in others. It is possible to constrain the valid values for an attribute, such as in the following example: *<!ATTLIST author_name alive (yes | no | maybe) "yes">,* which requires that the attribute be assigned values of either *yes, no,* or *maybe,* with a default of maybe if the attribute value is omitted for a particular element.

As an alternative to specifying default values for attributes, it is also possible to use the *#IMPLIED* keyword in the attribute declaration to make the attribute optional for the element, or the *#REQUIRED* keyword to require that a value for the attribute be specified whenever the associated element appears. Alternatively, the *#FIXED* keyword followed by a value in quotation marks may be used to specify that the attribute will always have the given value. For example, an *<!ATTLIST author_name alive CDATA #IMPLIED>* statement in a DTD would make the *alive* attribute optional, and an *<!ATTLIST author_name alive CDATA #REQUIRED>* statement would require that it be specified whenever the *author_name* element is used.

As is apparent from the discussion above and from Figure 7.4, there are also a number of codes that may be used in DTD element declarations. These are explained in Table 7.1.

| Element Declaration Code | Description |
|---|---|
| (#PCDATA) | Character data |
| (#PCDATA)* | Zero or more characters |
| (anelementname) | One instance of an element |
| (anelementname?) | Zero or one instances of an element |
| (anelementname*) | Zero or more instances of an element |
| (anelementname+) | One or more instances of an element |
| (anelementname1, anelementname2) | One instance of one element and one instance of another element |
| (anelementname1 \| anelementname2) | One instance of one element or one instance of another element |

**Table 7.1:** Some of the most common codes used in the element declaration codes of DTDs. This data would go in the *elementdeclarationtype* section of an *<!ELEMENT elementname (elementdeclarationtype)>* declaration. These codes can be combined so that, for example, one might have an element declaration code of *(#PCDATA, anelement1, anelement2?, anelement3+).*

There are many other options offered by XML DTDs, and still more that are made available by alternative schemas. This sub-section only describes those parts of XML that are specifically used in ACE XML, however. Whitehead, Freidman-Hill and Vander Veer

(2002) provide a much more detailed description of XML, with a particular emphasis on using Java code to manipulate XML documents.

It is clear that XML allows simple, human-readable and extremely flexible document formats to be constructed. Further adding to its appeal, there is a great deal of free, open-source and well-documented code that is available for facilitating the structured parsing of XML files. Apache Xerces[169] is an example of such a parsing library that is available in a variety of programming languages. Such XML parsers typically provide two types of APIs for accessing XML data in convenient ways: *SAX* APIs allow the data to be accessed in a sequentially structured way and *DOM* APIs allow the data to be accessed in a hierarchically structured way.

When writing XML files, or manually parsing them, it is important to remember that single-byte Unicode is the default encoding, and that corresponding codes must be used, an issue which is especially important for special characters that do not have ASCII equivalents. Fortunately, many programming languages include libraries for automatically translating plain text data appropriately. Java, for example, includes the *java.net.URLEncoder* and *java.net.URLDecoder* core classes, which can be used to ensure that all text, including special characters, is appropriately encoded and decoded.

XML also includes its own special provisions for certain special characters. XML parsers all automatically understand basic character substitutions for reserve characters that have special meanings in XML, as specified in Table 7.2.

| XML Code | Corresponding Character |
|---|---|
| &lt; | < |
| &gt; | > |
| &amp; | & |
| &quot; | “ |
| &apos; | ‘ |

**Table 7.2:** Default XML character substitutions for characters that have special meanings in the XML specification.

Special characters can also be manually referred to using their decimal or x-prefixed hexadecimal Unicode code point preceded by the *&#* characters and followed by a

[169] xerces.apache.org

semicolon. So, for example, the Euro symbol (€) can be referred to in an XML character field as *&#8364;* or *&#x20ac;*. It is typically preferable to simply pre-process all strings read from or written to an XML file by classes such as *java.net.URLEncoder* and *java.net.URLDecoder* rather than performing such manual encodings, however.

### 7.2.4 RDF

*RDF,* or *Resource Description Framework,* refers to a family of W3C[170] syntax specifications for representing relationships between different entities, or *resources*. In essence, RDF provides an abstract model for describing how anything can be related to any other thing. The result is effectively a labelled directed multi-graph.

RDF uses subject-predicate-object *triples* to make statements about the relationships between resources. The *subject* denotes a resource and the *predicate* denotes characteristics of the resource and its relationship with another resource, the *object*. For example, the statement *So What is a song that belongs to the genre of Modal Jazz* is a triple specifying that the subject *So What* is related to the object *Modal Jazz* by the predicate *is a song that belongs to the genre.*

Resources are often referred to using *Uniform Resource Identifiers,* or *URIs,* to make them accessible via the Internet. Further information on individual resources can thus be acquired by accessing the data stored at their respective URIs, a process called *dereferencing* in RDF terminology. However, linking to resources using URIs is not an obligatory requirement of RDF. Resources can in fact potentially be abstract entities that do not actually exist anywhere on the Internet or elsewhere. In order for RDF producers and consumers to be in agreement on the semantics of resource identifiers, it is often useful to externally define certain controlled vocabularies, such as the Dublin Core[171] metadata set, which is partially mapped to a URI space for use in RDF.

A process called *reification* can be used to achieve further expressiveness using triples, or to deduce measures of confidence about triples. This involves assigning a URI to each triple so that it can itself be treated as a resource about which other triples can be formulated.

[170] W3C, or the *World Wide Web Consortium,* is the primary international standards organization overseeing the World Wide Web.
[171] dublincore.org

There are a variety of specific formats, called *serialization formats,* that each provide different syntaxes for representing RDF data models. XML is often used as one way to provide a structured representation of RDF models, but there are also a variety of alternative serialization formats as well, such as Notation 3.[172] There are also several query languages designed for use with RDF graphs, the most common of which is an SQL-like language called SPARQL.[173]

A mechanism for describing relationships between resources, such as that offered by RDF, is an important component of the Semantic Web. This is because such a mechanism allows software to automatically store, exchange and otherwise use machine-readable information distributed on the Internet. This direct machine usability of RDF graphs is considered to be one of the main goals and advantages of RDF.

The generality, simplicity and abstract nature of RDF are both its keys strengths and its key weaknesses. Although these characteristics allow it to be used to describe a wide range of information in flexible ways, it can also introduce computational disadvantages and ambiguities. Similarly, the broad structural framework of RDF offers greater flexibility and extensibility than a particular given XML schema might, for example, but also does not incorporate the ability to quickly and easily define strict and sophisticated structures when in might be useful to do so, as one can in unrestricted XML.

### 7.2.5 OWL

*OWL,*[174] or the *Web Ontology Language,* is a family of languages for representing *ontologies,* which is to say formal representations of sets of concepts within some defined domain and the relationships between the concepts. Ontologies can be useful in the Semantic Web, as well as in other domains like machine learning.[175] *OWL Full,* one of the variants of OWL, provides partial compatibility with RDF.

The data contained in an OWL ontology is represented as a set of *individuals* that are related to one another via *property assertions.* OWL individuals can be collected into sets, called *classes,* whose properties are constrained by sets of *axioms.* Semantics can be inferred from explicitly defined axioms when appropriate.

[172] www.w3.org/DesignIssues/Notation3
[173] www.w3.org/TR/rdf-sparql-query/
[174] www.w3.org/TR/owl-features/
[175] Sections 9.1 and 9.2 discuss ontologies in the context of machine learning.

### 7.2.6 Binary files and serialized objects

A *binary file* is a computer file consisting of 1's and 0's that may be used to encode any type of data. Although computer text files are therefore technically binary files, the term *binary file* is typically used to refer specifically to computer files that are not encoded in plain text formats.

Storing data such as extracted features in binary form can have a number of advantages over storing it in text files. For example, storing numbers even in a relatively character-sparse format such as ASCII requires at least a full byte per digit, a much greater amount of space than if the data were stored in binary. Parsing and processing data stored as text can also carry greater processing overhead.

Storing data in text files can also have important advantages over binary storage, however. One of the greatest advantages is that text files are human-readable, something that can be very convenient during debugging or other situations where direct human inspection of data is appropriate or convenient. Text files can also be parsed in an application and platform-independent way, while binary files are generally application-specific, and therefore much less portable. Furthermore, standard text compression techniques can be used to reduce text file sizes significantly.

Advantages such as these, combined with consistently cheaper storage, data transmission and processing power, as well as the popularity of flexible text-based data formatting protocols like XML have led to an increasing use of text as the preferred choice for data storage. For example, binary file formats in Microsoft Office have recently been replaced with compressed XML-based formats.

Many programming languages, including Java, include functionality for saving any objects in memory to files as binary *serialized objects.* This can be highly convenient for programmers, as there is no need to implement any specialized parsing or saving functionality. As might be expected, the main disadvantage of serialized objects, aside from the lack of human readability, is the common lack of portability once these objects are written to disk. Even serialized objects from relatively portable languages such as Java can sometimes fail to be properly read when accessed from newer releases of the language than they were saved under. For example, serialized Java Swing objects are often not portable across different versions of the Java Virtual Machine.

## *7.3 File formats used by existing MIR systems*

Although most MIR research has traditionally used specialized in-house file formats for storing information such as feature values and instance labels, most of which have been either binary dumps or simple delimited text files, there has been an increasing push in recent years to use more standardized file formats so that data can be shared between different research groups. This section describes some of the best-known and most general file formats in use by the MIR community. Both this section and Section 7.4 stress some of the strengths and weaknesses of these file formats.

### 7.3.1 Matlab binaries and Java serialized objects

MathWorks Matlab[176] is a numerical computing environment and programming language that is particularly popular among some MIR researchers because of the variety of its associated stable and effective toolboxes implementing digital signal processing and machine learning functionality. Matlab can easily save information such as extracted features and learned models to binary .mat files, with the result that researchers that use Matlab tend to favour the .mat file format.

Unfortunately, .mat binaries suffer from the same limitations as all binaries, as discussed in Section 7.2.6. Although there are a variety of scripts written in different languages for parsing Matlab binaries, the issue remains that Matlab is commercial software that uses a proprietary file format that is subject to the design decisions of MathWorks, which may not coincide with the needs of the MIR community. Furthermore, .mat files are not directly human-readable, nor do they allow data to be as easily structured in useful ways as some of the alternative file formats.

Any serialized Java object can be saved directly to disk and parsed by the Java Virtual Machine. The ease with which this can be done has made the use of Java serialized objects attractive to some researchers, just as has been the case with Matlab .mat files, but once again, concerns about the portability, stability, structural expressivity and lack of human-readability are serious concerns.

[176] www.mathworks.com

Nonetheless, one is sometimes forced to use serialized objects when dealing with third-party software. An example of this is the saving of trained Weka models.[177]

### 7.3.2 Weka ARFF

Weka (Witten and Frank 2005) is a well-known set of open-source machine learning libraries implemented in Java. Weka includes several front ends, including graphical user interfaces, as well as the core libraries and their associated APIs. The breadth and ease of use of Weka have caused it to be adopted by many MIR research labs and, as a result, its ARFF file format is likely the most commonly used format in the MIR community for storing extracted feature values and providing them to machine learning algorithms. As the closest thing to a standardized file format that there is in MIR, the ARFF format will be given special attention here. More information on ARFF is available in Witten and Frank's book as well as on the Weka ARFF Sourceforge page.[178]

Throughout the description of the ARFF format that follows, several limitations of the format may become evident to the reader in relation to the specific domain of MIR research, something that is to be expected of any format as generally applicable as ARFF. The focus of this particular section is on a purely objective description of the ARFF specification, so these issues will not be discussed here explicitly. However, the weaknesses of the ARFF format with respect to MIR are discussed in some detail in Section 7.4.

ARFF files are basic text files that specify feature values and class labels associated with individual instances. All ARFF files consist of a *Header* section that outlines the available features and class names, followed by a *Data* section holding the feature values and, potentially, class names for each instance. Comments may also be included on lines starting with a percentage sign.

The first non-comment line of an ARFF file must begin with a single *@relation* declaration of the form:

```
@RELATION <relation-name>
```

The *<relation-name>* is a string providing a name for the basic relationship or type of information that the ARFF file represents. For example, in the case of an artist

[177] Although information on instances themselves can be saved using the text-based Weka ARFF format, as discussed in **Section 7.3.2**.
[178] weka.wiki.sourceforge.net/ARFF

identification task, it might be *artist_classification* or *artist_identification*. As with other types of ARFF data, <*relation-name*> strings with spaces or percent signs in them must be enclosed in quotation marks. Also, as with other ARFF keywords, *@relation* statements are case insensitive.

The rest of the header consists of *@attribute* declarations of the form:

```
@ATTRIBUTE <attribute-name> <datatype>
```

<*attribute-name*> and <*datatype*> are strings specifying, respectively, the name of a feature,[179] in the form of a string, and the data type of this feature. If a feature value is present in one or more instances then it must be declared in an *@ATTRIBUTE* statement in the header. The following data types may be specified for a feature in *@ATTRIBUTE* statements:

- *numeric:* Numeric data that may be an integer or a real number.
- *integer:* Integer-only numeric data.
- *real:* Numeric data that is in the form of real numbers.
- *string:* A string of text data. Strings containing spaces or percent signs must be wrapped in quotation marks.
- *<nominal-specification>:* A string of text data that must, for the feature value of any given instance, correspond to one of a set of eligible strings specified for the feature in the *@ATTRIBUTE* declaration. Such declarations are in the form: *{<nominal_name_1>, <nominal_name_2>, <nominal_name_3>, ...}*.
- *date:* A date, which may be formatted in a variety of ways. The default is yyyy-MM-dd'T'HH:mm:ss.
- *relational:* A format implemented only in the most recent developer versions of Weka that allows a simple hierarchical relationship to be specified for features so that multi-instance classifiers can be used.

[179] Weka refers to *features* as *attributes.*

The final *@ATTRIBUTE* declaration in the header usually specifies the possible class names that an instance may have. The <*attribute-name*> is specified *class,* and the <*datatype*> must be a list of strings in the <*nominal-specification*> form.

The Data section of an ARFF file is specified once all features (and class names) have been declared in the Header section. The Data section is begun by placing a single-line *@DATA* statement after the last *@ATTRIBUTE* statement.

The feature values for each instance are then specified on a single line for each instance. The feature values must each be separated by a comma, and the feature values must be listed in the same order that the features were themselves declared in the Header with *@ATTRIBUTE* statements. Unknown feature values may be denoted by using a single question mark as a place holder. String and nominal feature values are case sensitive, even though Weka keywords are not case sensitive. Figure 7.5 provides a complete example of an example Weka file.

Weka also allows a slightly modified alternative to standard ARFF files called *sparse ARFF files.* These are essentially the same as standard ARFF files, except that they do not explicitly specify zero-value feature values,[180] and feature values are linked with index values associating them with particular features.

As of Weka 3.5.8 (a developer version), weights can be associated with instances. This is done by enclosing them in curly braces and appending the weight to the end of the line, as in the following example corresponding to the first instance from Figure 7.5:

```
0.0,250.0,silence, {4}
```

A weight of 4 is assigned here to this particular instance.

As a final note on Weka data formats, it should be noted that ARFF files cannot be used to represent trained models. Weka instead stores these as Java serialized objects.

[180] Values are assumed to be zero if they are not specified.

```
% TITLE: Music, applause, speech, silence discriminator.
%
% This artificial data, intended for demonstration purposes,
% specifies possible feature values extracted from windows of
% audio that could be used for segmenting the audio into
% sections of music, applause, speech and silence.
%
% Since this is a simple demonstration, only the basic
% features of spectral centurion and the duration of the
% recording from which the window was extracted are specified.

@RELATION music_applause_speech_silence_discriminator

@ATTRIBUTE spectral_centroid      NUMERIC
@ATTRIBUTE duration               NUMERIC
@ATTRIBUTE class                  {music, applause, speech, silence}

@DATA
0.0,250.0,silence
440.0,250.0,music
526.0,250.0,applause
0.0,372.5,applause
220.0,372.5,music
115.0,372.5,music
115.0,372.5,applause
854.6,960.3,?
```

**Figure 7.5:** A complete sample Weka file. The first lines all begin with percentage signs, which indicates that they are simply comments. The *@RELATION* statement indicates the beginning of the feature declarations and specifies a name for the relationship represented by the Weka file. Three features are specified using *@ATTRIBUTE* statements, namely *spectral centroid, duration* and *class.* In practice, *class* is not really a feature, but rather a *nominal-specification* of the candidate classes that instances can have. This is how classes are always declared in Weka. The actual features are listed for eight instances after the *@DATA* declaration, with the class name specified as the last attribute for each instance. Note that the class name is not specified for the last instance, since it is replaced by a question mark. This convention can be used for feature values as well if they are unknown.

### 7.3.3 SDIF

SDIF (Wright et al. 1999; Schwartz and Wright 2000), or the *Sound Description Interchange Format,* is a standardized file format for describing sound that was jointly developed by IRCAM and CNMAT. It is popular in the signal processing community. SDIF files consist of a basic fixed framework, and also include an extensible set of description types, including time-domain descriptions, frequency-domain models and sinusoidal models. SDIF files make use of XML.

The primary purpose of the SDIF format is the storage of audio for use in signal analysis and synthesis. The storage of feature values and other MIR-oriented data mining information is not the primary emphasis of the SDIF format, although features can certainly be represented as well. For example, one of the most fundamental structures of the SDIF format is a series of *frames,* each consisting of a four-byte *Frame Type ID,* a four-byte integer *Frame Size* and the frame data itself. Each frame must be a multiple of 64 bits. It is clear that this sort of structuring is not ideally suited for most MIR-oriented tasks, where greater flexibility is preferred.

Although SDIF can be extended to store features for the purposes of MIR applications (Burred et al. 2008), the emphasis is still on efficiently representing audio data, not on representing features in ways that are as flexible and clear as would be desirable in an ideal MIR-oriented format, nor on dealing with non-audio data such as symbolic or cultural features. The SDIF format is also not as convenient as one would ideally like for representing other MIR-relevant information such as sophisticated ontological class structures.

So, while SDIF is one of the best formats available for use in the audio signal processing research community, it is not as ideally suited for MIR. Although it is true that SDIF can be adapted for MIR research, using SDIF to meet the ideal needs of an MIR standardized file format, as described in Section 7.4, can be awkward.

### 7.3.4 Music Ontology

Music Ontology (Raimond et al. 2007; Raimond and Sandler 2008; Raimond 2009; www.musicontology.com) is a framework designed for dealing with music-related data on the Semantic Web, with the particular needs of MIR applications in mind. It is designed to be very flexible, and takes advantage of RDF to facilitate connections between resources and to make it possible to access further information about resources by dereferencing them. Music Ontology is divided into three main areas that respectively deal with *editorial information* (e.g., track names, musicians, record labels, etc.), *production workflow information* (e.g., arrangements, compositions, etc.) and *event decompositions*, (e.g., specifying that a particular musician played in a particular key at a particular time).

Music Ontology makes use of several existing ontologies, including the *Timeline, Event* and *Functional Requirements for Bibliographic Records* ontologies. The Timeline ontology, which is based on OWL, is used for representing a variety of types of temporal information, and can represent instances in time, intervals in time and references to defined timelines.

The Event ontology is used to represent particular musical or other events that can be localized in both time and space. Music Ontology Events can also have *factors* (e.g., a musical instrument), *agents* (e.g., a performer playing the instrument) and *products* (e.g., the physical sound produced by the musician playing the instrument). Complex Events can also be related to subordinate *Sub-Events,* such as a large Event consisting of an ensemble of musicians playing music made up of Sub-Events each consisting of a particular musician playing particular notes.

The Functional Requirements for Bibliographic Records ontology includes *Works* (abstract artistic creations such as musical compositions), *Manifestations* (physical embodiments of Works, such as CDs in general) and *Items* (an instance of a Manifestation, such as a particular CD). Music Ontology also makes use of other existing ontologies, such as the social networking-oriented *FOAF* (Friend Of A Friend) ontology and its concepts of *Persons* and *Groups*.

Music Ontology is designed using an object-oriented approach that emphasizes inheritance. For example, the *Key* ontology, *Instrument* taxonomy and *Genre* taxonomy are all sub-classes of Events.

Of particular interest with relation to machine learning, Music Ontology includes an *Audio Features* ontology[181] intended for the expression of features extracted from audio signals. Events are interpreted in this context to be regions of time corresponding to

[181] motools.sourceforge.net/doc/audio_features.html

particular features called *FeatureEvents,* and each FeatureEvent may have a number of *Feature* factors that each represents a feature such as a musical key or a set of MFCCs.

A number of data resources have already been converted to Music Ontology, including Musicbrainz musical metadata[182] and data from the DBTune music repositories.[183] Music Ontology is also being used as the main representation format in the large OMRAS2 project, and is used by high-quality MIR-oriented software such as Sonic Visualiser (Cannam et al. 2006) and some of its associated Vamp plug-ins.[184]

A number of advantages and disadvantages of Music Ontology relative to ACE XML will become apparent to readers as the details of ACE XML are presented later in this chapter. Overall, Music Ontology has both the relative strength and weakness that it is very general, and is intended for MIR-oriented applications that are much wider in scope than the music classification focus of ACE XML. This enables Music Ontology to represent a much greater range of information more conveniently than can be done in ACE XML, but does not have all of the structural advantages of the ACE XML formats that make them particularly useful for music classification. XML in general tends to be much better suited to representing well-structured data than RDF. Although the Audio Features ontology subclass of Music Ontology does bring a greater focus on the music classification domain, this ontology as it is described in motools.sourceforge.net/doc/audio_features.html is not as flexible and convenient as ACE XML with respect to feature values, instance labels, feature descriptions and class interrelationships. Also, the Audio Features ontology is designed particularly with respect to audio features, whereas ACE XML treats audio, symbolic and cultural features equally and equivalently. Finally, the Audio Features ontology is still in relatively preliminary design stages as of the time of this writing.

Music Ontology has the advantage over ACE XML that it is explicitly designed to facilitate the referencing of external resources. Although it is certainly possible to specify URIs in ACE XML identifier fields, ACE XML does not have an explicit RDF framework that makes it particularly convenient to do so. Furthermore, the RDF

[182] zitgist.com
[183] purl.org/dbtune/
[184] www.vamp-plugins.org

framework makes it possible to use existing tools such as SPARQL to query Music Ontology data.

The choice to use basic XML rather than RDF in ACE XML does carry a number of advantages, however. For example, ACE XML files are typically much more human readable than Music Ontology's RDF files, which are more oriented towards machine readability. ACE XML files are also simpler and more obviously structured, with fewer varieties. This makes them very easy for users to learn and write code for. The majority of researchers in the MIR community and many of its associated disciplines tend to, in general, be much more familiar with XML than they are with RDF, with the consequence that they will likely be more willing to adopt an XML-based standard.

Of course, file parsing and writing code libraries have already been implemented for both ACE XML and Music Ontology, but the simple and clear structuring of ACE XML makes it easier for MIR researchers to quickly inspect the simple and self-contained ACE XML DTDs, learn them and write code for them, something that is essential for file formats intended for adoption as standards. This can be particularly true for many key areas of MIR research that are more oriented to the humanities than to technical applications. A related advantage is that ACE XML only relies on simple XML parsing, and does not require the installation of packages for parsing additional standards such as OWL, for example.

Also, although the implicit ease of linking disparate resources offered by RDF can certainly be a strong advantage in many contexts, it can also be a disadvantage when different linked documents are inconsistent or missing, which can often be an issue in MIR, at least in its current state. RDF-based approaches by their nature tend to rely on the accessibility of potentially widely distributed network resources. This can be a significant disadvantage if one does not have network access at a particular moment, or if a remote resource is removed, renamed or moved. If even one resource is eliminated it is possible that one will not only lose access to it, but potentially to all of the resources that it refers to as well. Self-contained XML files, on the other hand, do not carry this risk.

Furthermore, the ability to easily store information such as feature values and instance labels locally if desired can be an important advantage when dealing with many gigabytes of feature values. Limitations such as slow network connections and monthly bandwidth caps can pose serious obstacles when dealing with widely distributed network ontologies, but are less of a problem with file types that store the most essential information in a self-contained way and that can be downloaded or uploaded when convenient, such as ACE XML. Although RDF-based ontologies do certainly have additional important advantages of their own, they are perhaps better suited to relatively small amounts of textual data than to the very large feature associated with many typical MIR use cases and the even larger datasets that are likely to arise in the future as MIR research scales up to include larger quantities of music. ACE XML 2.0 files can also, of course, be made accessible on networks and linked to external ontologies if desired, but the ability to easily use them purely locally can be an important advantage.

ACE XML seeks to reach a compromise between the strong encouragement of distributed linked files promoted by RDF and the more conventional approach of having all data, including feature values and class labels, contained in a single file. ACE XML does this by using four different file formats to express different types of information, and makes it relatively easy to merge files of both the same and different types. The ability to merge separate files when convenient into single self-contained units when needed for the purposes of convenience and robustness is emphasized in the ACE 2.0 ZIP format described in Section 7.11.1.

ACE XML provides a good compromise between the simplicity of ARFF and the generality of Music Ontology and RDF as a whole. ACE XML is strongly and consistently structured, with a special eye to flexibility, so that features and labels of any kind can be specified in well-understood but extensible ways. Much more useful information can be represented with ACE XML than is possible with ARFF, but a much stronger structure is imposed on the data than in RDF, with the result that all of the data that is typically used by music classification researchers or is likely to be used by them in the foreseeable future can be represented in ways that can be clearly and consistently parsed using simple and standardized software. Furthermore, this diverse information can all be expressed in a fully self-contained way, without dependencies on distributed resources that are potentially fragile in terms of both their accessibility and longevity.

With respect to ACE XML 2.0, users have the option of using only the basics of the ACE XML specification if they wish. This means that the files are very simple and easy to learn for new users and for those implementing new software that uses ACE XML. ACE XML 2.0 also includes the ability to represent more sophisticated information if needed, and special attention has been paid to providing handles that can be used to link ACE XML 2.0 files to external resources in a variety of ways if this is a particular need, including RDF resources, thereby providing accessibility to the benefits of the RDF world.

In general, it can be said that Music Ontology is likely a preferable format for general representation and for the linking of musical data on-line, and that ACE XML is likely a preferable format in the specific domain of music classification, particularly with respect to feature extraction and instance labeling. Music Ontology can certainly be used for such purposes as well, however. In the long-term, it is certainly possible that RDF-based approaches and the semantic web in general will be able to truly demonstrate and take advantage of the power of their generality. This potential has yet to be reached, however, despite efforts dating back well over a decade. In the meantime, more strongly structured approaches such as ACE XML have significant practical advantages for use cases associated with most MIR classification research, both academic and commercial.

### 7.3.5 RapidMiner

RapidMiner (Mierswa et al. 2006), which was formerly known as YALE *(Yet Another Machine Learning Environment),* is an environment for performing machine learning and data mining experiments. Such experiments can be defined using nestable operators, which can be described in XML files.

Although the RapidMiner XML files can be very useful in embedding RapidMiner functionality in other applications, or in communicating experimental setups to other research groups, the particular emphasis is on communicating experimental configurations rather than on communicating data itself. This is limiting for MIR researchers who might wish to use their own algorithms developed under frameworks that are not related to RapidMiner at all. Furthermore, RapidMiner is a general machine learning environment that is not intended specifically for music, and as such has many of the same weaknesses as Weka ARFF files when applied specifically to MIR-oriented data, as described in Section 7.4.

### 7.3.6 M2K and MIREX

M2K (Downie et al. 2005), or *Music-to-Knowledge,* is a graphical feature extraction and classification framework based on the D2K parallel data mining and machine learning system. M2K has most famously been used as the primary framework for carrying out the various MIREX[185] comparisons of different algorithms. Unfortunately, at least from the perspective of developing a standardized MIR file format, the individual committees of participants organizing each of the MIREX tasks have generally tended to choose a range of differing file formats for each task, rather than coordinating to all use the same file formats for communicating information such as feature values, instance labels and class ontologies. Most of the file formats that have been used have been either simple Java serialized objects or delimited text files, and the more sophisticated data structuring made possible by frameworks such as XML or RDF has not been taken advantage of.

### 7.3.7 Marsyas

Marsyas (Tzanetakis and Cook 2000) is a set of software tools for analyzing and processing audio. It was one of the first major open-source MIR systems, and has been widely used for feature extraction, among other tasks. Although Marsyas does have a few basic text file formats, such as the .mtl *Marsyas Timeline* files, the main emphasis in Marsyas is on interoperability (Tzanetakis et al. 2008), with the result that Marsyas promotes the use of existing formats like Weka ARFF files and Matlab files.

### 7.3.8 CLAM

CLAM (Amatrain, Arumi and Ramirez 2002) is another prominent set of signal processing software tools which, among other things, can be used to extract audio features. Although oriented more towards signal modification than specifically MIR, features can be saved to either basic XML files or SDIF files. Although certainly useful for the purposes for which CLAM is intended, these files are not sufficiently sophisticated, expressive or flexible for the ideal needs of MIR, as specified below.

[185] www.music-ir.org/mirex/2009/

### 7.3.9 CrestMuseXML

CrestMuseXML (Kitahara 2008; www.crestmuse.jp/index-e.html) is an extensible framework for describing music. CrestMuse XML is part of the CrestMuse project, which emphasizes the integration of different XML formats. Although CrestMuse XML remains to be widely experimented with by the international MIR community, it does hold significant potential for integrating the benefits of different formats.

## *7.4 Limitations of existing formats and resultant design priorities*

There are a number of important shortcomings that are each found in all or most of the existing file formats that are currently used in MIR classification research. This section discusses some of the most significant of these limitations. Taking these problems into account, a number of design priorities are then proposed for consideration in the implementation of any future file formats intended for MIR-oriented data mining applications.

Since there is insufficient space here to provide a detailed analysis of all file formats that have been used in MIR, a special focus will be placed on describing the most important shortcomings of the Weka ARFF format (see Section 7.3.2) in particular, as it appears to be the most commonly used format in MIR and also illustrates many of the common shortcomings of other formats. Note that this is not in any way meant to denigrate ARFF files, which are in fact so popular precisely because they are one of the best general formats available. ARFF files are intended for general data mining research, and as such certainly cannot be expected to meet the special application-specific needs of MIR.

One serious limitation of ARFF files is that there is no convenient way to assign more than one class to a given instance.[186] This is not a serious shortcoming for most pattern recognition tasks, which typically require classification into one and only one class. However, there are many MIR research domains where the limitation of one class label

[186] There are, however, two possible workarounds. The first is to break one multi-class problem into many binary classification problems, so that there is a separate ARFF file for every class, with all instances classified as either belonging or not belonging to each class. Alternatively, one could create a separate class for every possible combination of classes, with a resulting exponential increase in the total numbers of classes. Unfortunately, both of these workarounds are inconvenient, and implicitly require classifier configurations that are much less than ideal.

per instance is a very problematic limitation. For example, any genre classifier dealing with even a moderate genre ontology would be unrealistic if it could not assign multiple labels to individual pieces of music. Similarly, a performer classification system should be able to assign multiple labels to pieces, particularly in the case of pieces with prominent soloists or with musical groups whose members have also had prominent solo careers (e.g., Cream songs should likely also be labeled with Eric Clapton, many of the pieces on the *Kind of Blue* albums should be labeled with both Miles Davis and John Coltrane, at the very least, etc.). Many MIR areas involve certain inherent ambiguities relating to class labels, and the imposition of only one class membership at the fundamental file format level is an unacceptable limitation for any realistic MIR classification system.

A second problem with ARFF files is that they do not permit any logical grouping of features. ARFF files treat each feature as an independent entity with no relation to any other feature. In contrast to this, one often encounters multi-dimensional features in music, and it can be useful to maintain logical relationships between the components of such features. Power spectra, MFCCs, bins of a beat histogram and a binary list of instruments present are just a few examples of music related multi-dimensional features. Maintaining a logical relationship between the values of multi-dimensional features allows one to perform classifications in particularly fruitful ways that take advantage of their interrelatedness, particularly with respect to classifier ensembles. Training one neural net on MFCCs, for example, and using another classifier for one-dimensional features such as RMS or spectral centroid can prove much more fruitful than mixing the MFCCs in with the other features. To give another example, it can also be useful for other reasons to group features that are derived from one another, such as in the case of the average value, average derivative and standard deviation of a particular feature.

A third problem is that ARFF files do not allow any labeling or structuring of instances. Each instance is stored only as a collection of feature values and a class label, with no identifying metadata. In music, it is often appropriate to extract features over a potentially overlapping time series of windows for each musical piece, something that results in sets of related ordered sub-sections of individual pieces. This is absolutely essential in applications such as automatic recording segmentation or structural analysis, and is convenient in a wide variety of MIR-oriented tasks. Furthermore, some features may be extracted for each window, some only for some windows and some only for each recording as a whole. ARFF files provide no way of associating features extracted from a window with the recording that the window comes from, nor do they provide any means of identifying recordings or of storing time stamps associated with each window. This means that this information must be stored, organized and processed by some external software using some additional unspecified and non-standardized file format.

A fourth problem is that there is no way to provide any metadata about features in ARFF files, other than unstructured information in comments. This can be a problem if data is to be shared amongst different groups, who may wish to use it to train their own systems, for example. In cases such as this, it is necessary to know details about the features and the particular parameters with which they were extracted (e.g., the roll-off point for the Spectral Roll-Off feature) if features are to be extracted from new instances to be classified based on the features provided in the original dataset.

A fifth problem is that there is no way of imposing a structure on class labels in ARFF files. One often encounters hierarchical or other ontological structures in music, such as in the cases of genre categories or structural analyses. Weka treats each class as distinct and independent. This means that there is no native way to use classification techniques that make use of structured ontologies, such as hierarchical tree classifiers, for example. This also means that there is no way to use weighted misclassification training strategies that penalize misclassifications into dissimilar classes more severely than misclassifications into similar classes.

The following list, based on the above analysis of ARFF files and on additional general observations, outlines a set of (sometimes by necessity contradictory) requirements that are proposed for consideration in the design of any new standardized file formats intended for general MIR classification research:

- File formats should be as simple and easy to understand as possible. This makes it easier for users to learn the formats and adopt them. It also decreases the probability of unforeseen conflicts and inconsistencies.

- File formats should be as flexible and expressive as possible, within the constraint of avoiding excessive complexity and redundancy.
- The data stored in the files should be easily human readable. This is important for purposes of debugging and general utility. It is can also be useful for the purposes of allowing humans to write files manually when the design of annotation software would be inappropriate or unnecessarily time consuming.
- The data stored in the files should be easily machine readable. If a file format is difficult to write parsers with or is parsed into inconvenient data structures then it will be difficult to convince users to adopt it as a standard.
- The data should be stored as efficiently as possible, in order to avoid excessively large files, within the constraint of maintaining human readability.
- Some widely accepted and well-known existing standard technology, such as XML, should be used. This increases the likelihood that new file formats will be themselves adopted as standards because they will be based on a proven technology, because users are likely to be already be at least somewhat familiar with the technology and because parsing libraries will already be available.
- File formats should rely on as few external technologies as possible. Each external technology that is present increases the probability that a given programming language used to develop a particular application may not include parsing libraries for that technology, that a parsing library does not function under a given operating system or that a component of the system will become obsolete in the future.
- The fundamental types of information that need to be represented are: feature values extracted from instances, class label annotations of instances, abstract descriptions of features and their parameters, and ontological structuring of candidate class labels.
- It should be possible to express features extracted from audio, symbolic and cultural sources of information, and treat these features equivalently so that they can easily be combined.

- It should be easy to reuse files, such as in the case of the same set of audio feature values being used for both genre classification and artist identification. Similarly, it could be convenient to reuse the same model classifications with different sets of features. For example, one could classify a given corpus of audio recordings and then later perform the same task on symbolic versions of the same corpus using the same model classifications.

- It is useful to emphasize a clear separation between the feature extraction and classification tasks. This is in contrast to formats such as Weka ARFF, which combine feature values with class labels. A separation between these two types of data is important because individual researchers may have reasons for using particular feature extractors or particular classification systems. The file format should therefore make it possible to use any feature extractor to communicate features of any type to any classification system. This portability makes it possible to process features generated by different feature extractors with the same classification system, or to use a given set of extracted features with multiple classification systems.

- It should be a simple matter to combine files of the same type, such as in the case of features extracted during different feature extraction sessions.

- It should be a simple matter to package files expressing related types of information (e.g., feature values extracted from particular instances and abstract information about the features themselves) together when appropriate, but also to separate them out when convenient. This helps to ensure data availability, integrity and accessibility, as well as flexibility.

- It should be possible to use files to reference external sources of information, but in such a way that doing so does not introduce dependencies on external information that may no longer be available in the future or that changes unexpectedly.

- It should be possible to assign an arbitrary number of class labels to each instance. It should also be possible to express relative weightings for each of these labels.

- It should be possible to group the dimensions of multi-dimensional features and to logically associate related features and their values in general with one another.
- It should be possible to assign identifying metadata to instances (e.g., recording titles or time stamps) that will associate meaning and context for the instances so that they can be identified, both internally and externally. In general, users should be free to specify whatever metadata fields they wish, and the file format should not limit them to specific fields. However, it may be useful to supply sample templates of particular schemas.
- It should be possible to specify relationships between specific instances, in both ordered and hierarchical ways. In the case of the former, this could be a time series of analysis windows, for example. In the case of the latter, it could be a hierarchical ranking of, from bottom to top, features extracted from individual analysis windows of a recording, compared to features extracted for recordings as a whole, compared to features extracted for a performer as a whole, etc. In any case, it should be possible to express both feature values and class labels for both overall instances and ordered sub-sections of them.
- For time-series data, it should be possible for analysis windows to overlap with one another and for section labels to overlap with one another.
- For time-series data, it should be possible for analysis windows to have variable sizes, rather than requiring them all to be the same size.
- It should be possible for feature values to be present for some instances and/or sub-instances, but not others. For example, some features may be extracted for all analysis windows, some only for some windows and some only for each recording as a whole.
- Related to this, it should be possible to abstractly specify the appropriateness of different features for different contexts. For example, some features might only be appropriate to extract for a whole recording, not its analysis windows, or some features might only be possible to extract once one or more windows have already been calculated (e.g., spectral flux).

- It should be possible to specify general metadata about features, including identifying feature names, descriptions and extraction parameters.
- It should be possible to specify relationships between different class labels, both hierarchical and otherwise. This is important for the specification of class ontologies that can be taken advantage of by machine learning strategies such as hierarchical learning algorithms or weighted misclassification penalization during training. It should also be possible to relatively weight the associations between class labels.

## 7.5 Overview of the ACE XML file formats

This section provides an overview of the ACE XML 1.1 file formats and provides motivations for some of the general design decisions behind them. In all cases, the guidelines described in Section 7.4 guided the design of these file formats. The proposed ACE XML 2.0 formats, as described in Section 7.11, go further still in meeting the Section 7.4 guidelines.

Sections 7.6 to 7.9 provide more detailed individual descriptions of each of the four ACE XML 1.1 formats, including their DTDs. There is also a sample full ACE XML 1.1 file for each of the four ACE XML file types provided in the on-line sample file appendix.[187] For the sake of comparison, these sample files are constructed such that they express the same base data as that described by the sample Weka ARFF file in Figure 7.5, but take advantage of the increased expressivity offered by the ACE XML file formats.

ACE XML 1.1 is the current stable version of ACE XML, and is supported by all of the jMIR software components. It is the first published version of ACE XML (McKay et al. 2005), and is a minor update over the never published ACE 1.0 test prototype. Section 7.11 describes ACE 2.0, which is a proposed update of ACE that is not yet finalized or implemented in software.

As implied by their name, ACE XML files are all XML-based. XML was chosen because it is not only a standardized format for which parsers are widely available, but is also an extremely flexible format. It is a verbose format, with the consequence that it is

[187] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

less space efficient than formats such as ARFF, but this verbosity has the corresponding advantage that it allows humans to read and write the files relatively easily.

It was decided to use XML DTDs rather than another XML schema to specify the structures used by ACE XML files. Although other schemas can in general be more expressive than DTDs, DTDs are nonetheless sufficiently expressive for the purposes of ACE XML. They also have the significant advantages of being simpler and easier to understand, thereby making the ACE XML formats more attractive to new users and making the work of those implementing custom ACE XML parsers and writers easier. This is a particular issue given the wide variety of alternative schemas that are available. The average member of the MIR community is much less likely to be familiar with any particular one of these schema languages, particularly in the cases of those specialized schemas that provide enough increased expressivity to arguably have advantages over the simple DTD approach. Furthermore, DTDs tend to be much simpler and straight-forward, and are therefore much easier to learn for those users who might not know any XML at all. The ACE XML DTDs are specified in each ACE XML file along with the file's data, which means that each ACE XML file is packaged with an explanation of its formatting.

As briefly discussed in Section 7.1, There are four different types of ACE XML files: *Feature Value* files that express feature values extracted from instances, *Feature Description* files that describe features abstractly, *Instance Label* files that allow labels to be associated with instances and allow the specification of metadata about instances, and *Class Ontology* files that specify relationships between different candidate class labels.

Each of these four XML file types may be used independently, or they may be associated with one another and logically merged in software using unique identifying keys, such as matching *data_set_id* fields found in both Feature Value and Instance Label files, for example. Multiple files of the same type can also be merged using the ACE software. To give just a few examples: two Feature Value files containing the same features for different instances could be merged into one file, two different Feature Value files containing different features for the same instances could be merged, an Instance Label file containing only sub-section labels and an Instance Label file containing only overall instance labels could be merged, etc.

It is not in any way necessary to provide all four ACE XML file types for any application if this is not appropriate or needed. For example, if a classifier is already trained and is to be used to classify unknown patterns, then there is certainly no need for an input Instance Label file, although one may be output during processing to express the predicted classes. The ACE software and its code libraries will automatically construct implied data for missing file types in a way that is effortless and transparent to the user. For example, if only a Feature Value file and an Instance Label file are specified, the software will automatically construct a flat class ontology based on the labels present in the Instance Label file and will also automatically generate feature descriptions based on the characteristics of the features present in the Feature Value file (e.g., the dimensionality of each of the features).

The decision to use four different file types rather than the more typical single file type is unorthodox, and therefore requires some justification. As discussed in Section 7.4, it is useful to incorporate a separation between feature values and instance labels. This is important for data reusability, such as in a case where one might extract features once from a large number of recordings, and then reuse this single resulting Feature Value file for multiple purposes, such as classification of artist, composer, genre and geographical point of origin. If there were only one ACE XML file type, then features would have to be repeated for each of these applications, but with the multiple file type approach the Feature Value file can remain unchanged, and be reused with a different Instance Label file for each classification task. Similarly, one can imagine a case where the same model classifications contained in one Instance Label file are used for separate sets of features extracted from symbolic, cultural and audio data respectively contained in three different Feature Value files.

Feature descriptions and class ontologies are each distributed in separate files as well in order to emphasize their independence from particular instances. For example, a Feature Description file could be published on its own to demonstrate general features that can be extracted by a particular feature extraction application in general, or a Feature Description file could be published on its own that contains specific extraction parameters that were found to be effective for a particular research domain, or a Feature Description file could be packaged with a Feature Value file containing feature values extracted for particular instances, as appropriate. Similarly, class ontologies can be published in a way that is independent of particular instances and features, or even of a particular data set. Such file type separations emphasize the abstract nature of many types of data that are useful in music classification, and allow them to be distributed and used either independently or together, as appropriate, rather than artificially forcing links where they may not always be appropriate.

The use of separate file formats also has advantages with respect to data longevity and convenience when updating the data. For example, if new features become available after a Feature Value file has been generated, it would only be necessary to update the Feature Value and Feature Description files since the data stored in the other two file types could be reused unmodified. Similarly, if a genre ontology changed over time, it would not be necessary to update already existing Feature Value or Feature Description files.

Overall and most importantly, the separation of different types of data into four different file types makes it possible to distribute and use one type of file for arbitrary purposes without needing to impose one's own choices with respect to the types of data described by the other three file types. Also, the separation into multiple files types makes it easier to conceptualize and represent sophisticated arrangements of information with a divide and conquer approach.

jMIR includes several software tools for facilitating the use of the ACE XML file formats in general, including use outside of the scope of the principal jMIR software components themselves. Although ACE XML files may certainly be manually read, created and edited with text editors or XML editors such as XML Spy, the best way to perform such manual operations is to use the prototype ACE GUI, which displays data from single or multiple ACE XML files in particularly convenient ways.

A separate Java application called jMIRUtilities is also included as part of jMIR for, among other things, performing a number of functions that facilitate the general use of ACE XML files. Aspects of jMIRUtilities include a simple GUI for batch annotating files into an Instance Label file, functionality for generating Instance Label files based on simple tab delimited text files, functionality for accessing data from iTunes XML files so that it can then be imported into ACE XML files, and functionality for merging features contained in separate ACE XML Feature Value files.

## 7.6 The ACE XML 1.1 Feature Value file format

ACE XML Feature Value files are used to express feature values that have been extracted from instances and sub-sections of instances. Figure 7.6 specifies the DTD for Feature Value files. A sample complete Feature Value file is shown in Code Sample 7.1 of the on-line sample file appendix.[188]

```
<!ELEMENT feature_vector_file (comments,
                               data_set+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT data_set (data_set_id,
                    section*,
                    feature*)>
<!ELEMENT data_set_id (#PCDATA)>
<!ELEMENT section (feature+)>
<!ATTLIST section start CDATA ""
                  stop CDATA "">
<!ELEMENT feature (name, v+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT v (#PCDATA)>
```

**Figure 7.6:** The XML DTD for ACE XML 1.1 Feature Value files. This DTD precisely defines the information that may appear in Feature Value files and how it must be formatted. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.1 of the on-line sample file appendix.[189]

It can be seen from the Feature Value DTD that feature values can be expressed for overall instances, called *data_sets,* that are each named using the *data_set_id* element. Each *data_set_id* refers to a unique identifier, such as a file path or a URI. Each *data_set* may or may not have sub-sections, which are each specified using the *section* element. For example, a *data_set* might correspond to an audio recording and its sub-sections might correspond to analysis windows of the recording, although there is nothing about the Feature Value specification that requires this particular arrangement.

Each *data_set* sub-section must have *start* and *stop* stamps in order to indicate what portion of the *data_set* that it corresponds to. These stamps may or may not overlap and they may or may not be of equal sizes. This makes it possible to have, for example, overlapping analysis windows of arbitrary and potentially varying sizes. There is nothing about the *start* and *stop* attributes that requires them to denote time, however, and they

[188] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf
[189] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

could just as easily be used to denote a range of pixels in an image of album art, for example.

Features may be expressed for either a *data_set* as a whole or for individual sub-sections. In either case, the *feature* element is used to denote a new feature value or feature vector, and the particular feature is named using a *name* element in *each* feature clause.

Each *data_set* or *section* may have an arbitrary and potentially differing number of features in an arbitrary and potentially differing order. This makes it possible to omit features from some data sets or sub-sections if appropriate or if they are unavailable. Each feature may also have an arbitrary and potentially varying number of values, each denoted with a *v* element, in order to allow multi-dimensional features that may vary in dimensionality based on context.[190]

More information on instances described in a Feature Value files, such as class labels or identifying metadata, may be accessed by linking instances in the Feature Value file to instances in an ACE XML Instance Label file (see Section 7.8) by using *data_set_id* values that match across the two files.

Similarly, more information on the features themselves (as opposed to their values) can be specified by linking a Feature Value file to an ACE XML Feature Description file (see Section 7.7) by using *name* values in *feature* clauses in the Feature Value file that correspond to *name* values in *feature* clauses in the Feature Description file. However, it is not necessary to include Feature Description files with Feature Value files, since software such as ACE can automatically implicitly deduce information such as the dimensionality of features or whether particular features are to be extracted for overall instances or their sub-sections.

As a final note, it should be mentioned that a *feature_vector_file* tag is used in the DTD specification rather than a *feature_value_file* tag. This is for the purpose of legacy compatibility, since Feature Value files were called Feature Vector files in the deprecated ACE XL 1.0 specification.

[190] The dimensionality of a given feature type may alternatively be fixed in a Feature Description file, if desired.

## 7.7 The ACE XML 1.1 Feature Description file format

ACE XML Feature Description files are used to express abstract information about features themselves. Feature Description files do *not* specify feature values or other information related to specific instances, as this information is instead specified in Feature Value files (see Section 7.6). Figure 7.7 specifies the DTD for Feature Description files. A sample complete Feature Description file is shown in Code Sample 7.2 of the on-line sample file appendix.[191]

Feature Description files make it possible to specify information about features in a general sense in a way that is independent from particular feature extractions. This makes it possible to publish a self-contained Feature Description file describing the features that a particular feature extraction application can extract, for example, or to publish a list of features and associated parameters that were found to be useful for different research application, such as instrument classification and pitch classification.

```
<!ELEMENT feature_key_file (comments,
                            feature+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT feature (name,
                  description?,
                  is_sequential,
                  parallel_dimensions)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT is_sequential (#PCDATA)>
<!ELEMENT parallel_dimensions (#PCDATA)>
```

**Figure 7.7:** The XML DTD for ACE XML 1.1 Feature Description files. This DTD precisely defines the information that may appear in Feature Description files and how it must be formatted. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.2 of the on-line sample file appendix.[192]

It can be seen from the Feature Description DTD that information on each feature is expressed in a separate *feature* clause. Each such clause includes a *name* element uniquely identifying the feature and an optional *description* element that can be used to include textual metadata about the feature.

[191] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf
[192] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

The *is_sequential* element for each feature specifies whether or not the feature can be extracted from sub-sections of an instance. A value of *true* means that it can, and a value of *false* means that the feature can only be extracted from instances as a whole, not from their sub-sections. So, for example, a feature such as Most Common Pitch might have an *is_sequential* value of true for a MIDI file that is broken into analysis windows and the most common pitch is calculated for each individual window, but a feature such as Overall Key might have an *is_sequential* value of false because it would only be extracted for the MIDI file as a whole.

The *parallel_dimensions* element specifies the vector size of extracted features. This value will be 1 unless the feature is a multi-dimensional feature. So, for example, a multi-dimensional Pitch Histogram feature with 128 pitch bins would have a *parallel_dimensions* value of 128, but the one-dimensional Most Common Pitch feature would only have a *parallel_dimensions* value of 1.

Feature Description files can be linked with Feature Value files if it is desirable to describe the features used in a particular feature extraction run on a particular dataset. It can often be helpful to do this, as there are often many variable implementation details about features that are not apparent from examinations of actual feature values, so the distribution of feature details with extracted feature values makes it much easier to extract new feature values from new instances in ways that are compatible with earlier feature extractions.

The linking of a Feature Description file and a Feature Value file can be achieved by using *name* elements in *feature* clauses in the Feature Description file that correspond to matching *name* elements in *feature* clauses in the Feature Value file.

As a final note, it should be mentioned that a *feature_key_file* tag is used in the DTD specification rather than a *feature_description_file* tag. This is for the purpose of legacy compatibility, since Feature Description files were called Feature Key files in the deprecated ACE XML 1.0 specification.

## 7.8 The ACE XML 1.1 Instance Label file format

ACE XML Instance Label files are used to specify class labels and miscellaneous metadata about instances and sub-sections of instances. A typical use of this file type would be to express ground-truth model classifications or predicted classifications, but there are certainly other possible uses as well. Figure 7.8 specifies the DTD for Instance Label files. A sample complete Instance Label file is shown in Code Sample 7.3 of the on-line sample file appendix.[193]

```
<!ELEMENT classifications_file (comments,
                                data_set+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT data_set (data_set_id,
                    misc_info*,
                    role?,
                    classification)>
<!ELEMENT data_set_id (#PCDATA)>
<!ELEMENT misc_info (#PCDATA)>
<!ATTLIST misc_info info_type CDATA "">
<!ELEMENT role (#PCDATA)>
<!ELEMENT classification (section*,
                          class*)>
<!ELEMENT section (start,
                   stop,
                   class+)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT stop (#PCDATA)>
```

**Figure 7.8:** The XML DTD for ACE XML 1.1 Instance Label files. This DTD precisely defines the information that may appear in Instance Label files and how it must be formatted. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.3 of the on-line sample file appendix.[194]

It can be seen from the Instance Label DTD that class labels can be expressed for overall instances, called *data_sets,* that are each named using the *data_set_id* element. Each data_set_id should refer to a unique identifier, such as a file path or a URI. Each *data_set* may or may not have sub-sections, which are each specified using the *section* element. For example, a *data_set* might correspond to an audio recording and its sub-sections might correspond to analysis windows of this recording, although there is nothing about the Instance Label specification that requires this particular arrangement.

Each *data_set* instance may have pieces of metadata associated with it via the *misc_info* element. Each *misc_info* clause may be associated with an *info_type* attribute

[193] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf
[194] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

that specifies the type of metadata (e.g., title, album, composer, etc. field names) that the *misc_info* clause specifies.

The optional *role* element can be used to specify the purpose for which an instance is to be used. Values such as *training, testing* and *predicted* are typically used for this field to specify the role of the instance with respect to machine learning, but any string may be provided here if desired. This can be useful for purposes such as specifying pre-determined cross-validation folds, for example, and can be particularly useful for bookkeeping when multiple Instance Label files are merged.

Class labels are assigned to overall instances and/or their sub-sections using the *class* element. Multiple labels may be assigned to each instance or sub-section, or the label may be left unspecified if a particular label is unknown.

Each sub-section of an instance must have *start* and *stop* stamps in order to indicate the portion of the instance that it corresponds to. These stamps might often be used to specify time intervals, although there is nothing requiring that they be related specifically to time. The resultant sub-section intervals may or may not overlap and they may or may not be of equal sizes.

This arrangement permits two partially overlapping regions, where each region is labelled with a different class name, and the overlapping portion is associated with both labels. Such an occasion might occur, for example, in the ground-truth for a music/applause discriminator where the applause in a live performance begins before the music ends. Such a situation could be expressed as either two sections with one label each overlapping in time or as three non-overlapping consecutive sections where the outer sections have one label each and the central section has two labels, whichever is more convenient.

Information on specific feature values extracted from instances referred to in an Instance Label file may be accessed by linking the Instance Label file to an ACE XML Feature Value file (see Section 7.6) by using *data_set_id* values that match across the two files. Similarly, more information on the class labels used to label instances in an Instance Label file can be accessed by linking the Instance Label file to a Class Ontology file (see Section 7.9) by using *class* values that match across the two files.

As a final note, it should be mentioned that a *classifications_file* tag is used in the DTD specification rather than an *instance_label_file* tag. This is for the purpose of legacy compatibility, since Instance Label files were called Classification files in the deprecated ACE XL 1.0 specification.

## 7.9 The ACE XML 1.1 Class Ontology file format

ACE XML Class Ontology files are used to list candidate class labels for a particular classification domain and to specify hierarchical structures that connect different classes. As discussed in Section 7.11, the ACE XML 2.0 version of the Class Ontology format extends the purview of Class Ontology files to general weighted ontologies, but the ACE XML 1.1 version only permits extended tree-based taxonomical class structuring,[195] which at least is significantly more than the simple flat class structures used in the majority of current MIR research.

Class Ontology files do *not* specify the class labels of any specific instances, as this information is instead annotated in Instance Label files (see Section 7.8). Figure 7.9 specifies the DTD for Class Ontology files. A sample complete Class Ontology file is shown in Code Sample 7.4 of the on-line sample file appendix.[196]

The ability to specify hierarchical class structuring has several important benefits. From a musicological perspective, it provides a simple machine readable way of specifying meaningful structuring of classes. From a machine learning perspective, it has the dual advantages of enabling the use of potentially very powerful hierarchical classification methodologies that take advantage of this structuring (e.g., McKay 2004) as well as the use of learning schemes utilizing weighted penalization, such that misclassifications during training into related classes are penalized less severely than misclassifications into unrelated classes.

It can be seen from the Class Ontology DTD that flat class structures can be specified simply by listing a set of *parent_class* clauses, each with a *class_name* element used to specify the name of a class. This simple approach can be useful in communicating a list

[195] The tree-based structuring permitted by Class Ontology files is referred to as "extended" because the Class Ontology format and its associated ACE data structures allow the possibility of any given class being descended from multiple parent classes, something that is not permitted in standard tree structures. This can be implemented by specifying the same class name in multiple *parent_class* or *sub_class* clauses.
[196] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

of candidate class labels to an annotator, for example, or for combining a Class Ontology file with a set of labelled instances contained in an Instance Label file in order to ensure that it does not use any unexpected class labels.

```
<!ELEMENT taxonomy_file (comments,
                         parent_class+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT parent_class (class_name,
                        sub_class*)>
<!ELEMENT class_name (#PCDATA)>
<!ELEMENT sub_class (class_name,
                     sub_class*)>
```

**Figure 7.9:** The XML DTD for ACE XML 1.1 Class Ontology files. This DTD precisely defines the information that may appear in Class Ontology files and how it must be formatted. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.4 of the on-line sample file appendix.[197]

The structural aspect of Class Ontology files become apparent when the *sub_class* element is used to specify hierarchical structures of class labels under *parent_class* level classes. Each *parent_class* clause may contain an arbitrary number of *sub_classes,* and each *sub_class* may itself also contain an arbitrary number of *sub_classes,* with the result that a hierarchical class tree of arbitrary depth can be built under each *parent_class*. Each class with no descendants, be it in a *parent_class* or *sub_class* clause, can be referred to as a *leaf class,* and can be used to label instances in Instance Label files.

A Class Ontology file can be linked to an Instance Label file for use during training, or for other reasons, by using *class_name* values in the Class Ontology file that correspond to the *class* values in the Instance Label file.

As a final note, it should be mentioned that a *taxonomy_file* tag is used in the DTD specification rather than a *class_ontology_file* tag. This is for the purpose of legacy compatibility, since Class Ontology files were called Taxonomy files in the deprecated ACE XL 1.0 specification.

[197] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

## *7.10 Using ACE XML with new and existing non-jMIR software*

A key factor in the effectiveness of any effort to encourage researchers to adopt new standardised file formats is the ease with which they can parse and write to them in their own existing and new software. The simplicity of Weka ARFF files and the ample data structures and processing functionality offered by the Weka code base have certainly contributed to its broad adoption, for example. Although all jMIR software components are of course able to read and write all relevant ACE XML formats, this in itself is not sufficient to encourage the use of ACE XML in other software platforms.

jMIR therefore includes open-source Java libraries in the ACE code package that implement parsing and writing functionality for each of the ACE XML file types, as well as convenient data structures and general processing methods for dealing with the data that is parsed from files. This data can be used and manipulated directly within these libraries, or it can be exported to individual developers' own data structures. ACE's parsing and data structure libraries may be used entirely independently of the ACE meta learning software itself if desired.

Functionality has also been implemented to automatically convert data in ACE XML data structures into Weka data structures, and vice versa, in order to take advantage of the convenient and well-established functionality built into Weka. This also makes it possible to use Weka data structures as intermediaries for conversion to yet other formats. jMIR also includes utilities for directly translating back and forth between Weka ARFF and ACE XML files, although data that fundamentally cannot be represented in ARFF files is lost when doing so.

As discussed in Section 7.13, there are plans to implement ACE XML parsing and processing functionality in other programming languages and to build custom modules for other well-established MIR systems. For the moment, however, the Java implementation of the ACE XML processing functionality makes these libraries as accessible as libraries implemented in any single language can be. Java is platform independent, and the only third-party software used by the ACE XML processing libraries is the open-source Apache Xerces[198] XML-parsing library and the Weka

[198] xerces.apache.org/xerces-j/

library[199] (and this only if Weka functionality is used), both of which are also implemented in Java. This means that the ACE XML libraries can be easily accessed under any common operating system, with the further advantage that many systems such as Matlab include functionality for accessing Java externals. Furthermore, even if one is for some reason unable to access the ACE XML code libraries, general XML parsers are available in virtually all modern programming languages.

For the purpose of clarity, the overall structure of the Java classes used to parse and process ACE XML files is briefly outlined here. This is only a basic overview, however, and those wishing more details are referred to the ACE API and the well-documented code itself, available at jmir.sourceforge.net.

An important point to note before proceeding to the architectural details of the ACE XML Java classes is that the jMIR components themselves make use of the ACE classes described below when dealing with ACE XML files. This means that any bug fixes or updates to the ACE XML standard only need to be implemented once in these classes to be automatically updated in all of the jMIR components as well.

The parsing code for all ACE XML file formats is contained in the *ace.xmlparsers* Java package. Although changes to the ACE XML standard must be implemented here, users in general should never need to reference these classes directly when incorporating ACE XML functionality into their own code. This is because all of the parsing functionality can be accessed more conveniently from the higher-level classes in the *ace.datatypes* Java package. Having noted this, those users who do wish to have low-level access to the parsed data may wish to examine the *ParseClassificationsFileHandler, ParseDataSetHandler, ParseFeatureDefinitionsHandler* and *ParseTaxonomyFileHandler* classes in the ace.xmlparsers package for parsing Instance Label, Feature Value, Feature Description and Class Ontology ACE XML files respectively.

As noted above, the significant majority of users will prefer to use the ace.datatypes classes, however, which represent the data parsed from ACE XML files at a higher level and include access to file reading, automatic error and consistency checking, file writing, file merging, data translation and data processing functionality, as well as access to the basic data structures used to hold data once it is parsed from ACE XML files. A number

[199] www.cs.waikato.ac.nz/ml/weka/

of methods are designed specifically with the intention of making relevant data available in forms convenient to external software, and this data can often be simply imported over to users' own software in the form of simple and well-established data structures.

The overarching Java class in the ace.datatypes package is the *DataBoard*, which provides access to information relating to any of the four ACE XML file types, interpreted either independently or in conjunction with one another. The DataBoard also provides direct access to the *SegmentedClassification, DataSet, FeatureDefinition* and *Taxonomy* Java classes, each of which relates specifically to information stored in Instance Label, Feature Value, Feature Description and Class Ontology ACE XML files, respectively. The ace.datatypes package also contains other Java classes, but these relate more to either ACE XML 2.0 functionality, such as ACE ZIP files (see Section 7.11.1), or to ACE machine learning functionality that is not directly relevant to accessing data stored in ACE XML files.

## 7.11 Current developments: Proposed update to ACE XML 2.0

The ACE XML 1.0 file formats were developed at the beginning of the jMIR project, before any of the jMIR software components had themselves been completed. Minor changes were introduced in version 1.1, the stable version described in the sections above, but the file formats were frozen at this version after the publication of the first jMIR component (McKay et al. 2005). This was necessary because ACE XML is intended for use as a standard, which precludes the incorporation of changes that would render existing software that uses the previous format obsolete.

Of course, certain areas of potential improvement became apparent as more jMIR components were completed. The need for an update to the ACE XML specification also became increasingly apparent as the use of ACE XML as a standardised format for use in the inter-university NEMA[200] project became probable, something that would necessitate a number of changes for the sake of compatibility with other NEMA systems.

As a result, it was decided to design an overhauled version of ACE XML called ACE XML 2.0, which is described in the following sub-sections. It should be stressed that the specified ACE XML 2.0 formats are only proposals, and that the finalization and

[200] nema.lis.uiuc.edu

implementation of these formats is beyond the scope of this document. Given that ACE XML 2.0 is proposed as a standardized set of file formats intended for general use, it is appropriate to first publish proposed changes to the MIR research community for potential modification before finalizing and implementing the changes. ACE XML 1.1 is already implemented and established in all jMIR components and is still the “official” jMIR format at the time of publication of this document.

Some of the changes proposed for ACE XML 2.0 are based on the changing needs of the MIR community since ACE XML 1.1 was established. Others are simply for the sake of improved clarity, simplicity and improved consistency across ACE XML formats. The most fundamental changes, however, are based on the needs imposed by the NEMA project. The NEMA researchers at Queen Mary, University of London are invested in the Music Ontology format (see Section 7.3.4), and the NEMA researchers at the University of Illinois at Urbana-Champaign are building the NEMA infrastructure using Meandre,[201] both of which require specific modifications to ACE XML for the sake of full compatibility. For example, the addition of optional URI and other fields to the ACE XML files makes it possible to add RDF handles to ACE XML files if it is necessary to integrate them with file formats such as Music Ontology, while at the same time maintaining the advantages of essentially self-contained structured XML files.

As a consequence of these diverse needs, many of the characteristics of ACE XML 2.0 are the result of compromises between the often competing priorities of different researchers and research groups. The changes implemented in ACE XML 2.0 are intended to meet both these specific needs as well as the original design philosophy of ACE XML as much as possible. The main priority, however, has remained the implementation of formats that are as flexible and general as possible within the specific sphere of MIR classification research.

Many of the changes to ACE XML proposed in the sections below result from very helpful conversations, criticisms and suggestions from other researchers. These include, at McGill University, John Ashley Burgoyne, Rebecca Fiebrink, Ichiro Fujinaga, Daniel McEnnis and Jessica Thompson, among others. Jessica Thompson in particular deserves special credit for her central role with respect to the ACE ZIP format, as well as for ideas

[201] seasr.org/meandre/

relating to ACE XML in general. John Ashley Burgoyne is also playing an essential role in the ongoing implementation of the new software supporting ACE XML 2.0. And, of course, Ichiro Fujinaga's input was essential throughout the process, from beginning to end. Many other researchers outside McGill University have also contributed very helpful insights, including, Mert Bay, Douglas Eck, Andreas F. Ehmann, Ben Fields, Ian Knopke, Amit Kumar, Paul Lamere, Cyril Laurier, Kevin Page, Yves Raimond, Allen Renear, Mark Sandler, David Tcheng, Karen Wickett and, especially, J. Stephen Downie and Kris West.

It is important to stress that as much effort as possible needs to be made to keep ACE XML 2.0 backwards compatible with ACE XML 1.1, especially in terms of the data structures that the file formats support.

### 7.11.1 ACE XML 2.0 ZIP files and ACE XML 1.1 and 2.0 Project files

One of the first problems that became apparent with ACE XML was that large groups of XML files associated with a particular research project could be unwieldy to deal with collectively, and could potentially be confusing to new users. Since it is undesirable to combine the four formats into a single format, for reasons discussed in Sections 7.4 and 7.5, it was necessary to find a solution that would allow the continued use of multiple discrete files while at the same time simplifying the logistics related to using groups of them together.

The first solution was to devise an ACE XML Project file format that could be used to associate related files together for a given project. This format allows users of an application such as ACE, for example, to simply specify a single Project file, and then rely on the application to itself automatically open all of the files referred to by this Project file, thus increasing user convenience significantly.

A prototype Project file format was developed under the ACE XML 1.1 framework and implemented in the ACE Java package, but never finalized as a standard or implemented in the other jMIR components. The DTD of this prototype ACE XML 1.1 Project file is shown in Figure 7.10, and a revised ACE XML 2.0 version is shown in Figure 7.11. A sample ACE XML 1.1 Project file is shown in Code Sample 7.5 of the on-line sample file appendix, and a sample ACE XML 2.0 Project file is shown in Code Sample 7.6.[202]

```
<!ELEMENT ace_project_file (comments,
                            taxonomy_path,
                            feature_definitions_path,
                            feature_vectors_path,
                            model_classifications_path,
                            gui_preferences_path,
                            classifier_settings_path,
                            trained_classifiers_path,
                            weka_arff_path)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT taxonomy_path (#PCDATA)>
<!ELEMENT feature_definitions_path (path*)>
<!ELEMENT feature_vectors_path (path*)>
<!ELEMENT model_classifications_path (path*)>
<!ELEMENT gui_preferences_path (#PCDATA)>
<!ELEMENT classifier_settings_path (#PCDATA)>
<!ELEMENT trained_classifiers_path (#PCDATA)>
<!ELEMENT weka_arff_path (#PCDATA)>
<!ELEMENT path (#PCDATA)>
```

**Figure 7.10:** The XML DTD for the prototype ACE XML 1.1 Project file format. This DTD precisely defines the information that may appear in Project files and how it must be formatted. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.5 of the on-line sample file appendix.[203]

The *feature_vectors_path, feature_definitions_path, model_classifications_path* and *taxonomy_path* elements allow references to be made to external ACE XML Feature Value, Feature Description, Instance Label or Class Ontology files, respectively. Zero to many files of each type may be referred to, except in the case of Class Ontology files, for which only zero or one files may be referenced. Weka ARFF files can also be referred to using the *weka_arff_path* element if ACE XML files are unavailable for a certain dataset.

Preference files (in as of yet unspecified formats) for the ACE meta learning application can also be specified using the *gui_preferences_path* and *classifier_settings_path* elements. Trained classification models can be referenced via the *trained_classifiers_path* element.

[202] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf
[203] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

```
<!ELEMENT ace_xml_project_file_2_0 (comments?,
                                   feature_value_id,
                                   instance_label_id,
                                   class_ontology_id,
                                   feature_description_id,
                                   weka_arff_id?,
                                   trained_model_id?,
                                   uri?)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT feature_value_id (path*)>
<!ELEMENT instance_label_id (path*)>
<!ELEMENT class_ontology_id (#PCDATA)>
<!ELEMENT feature_description_id (path*)>
<!ELEMENT weka_arff_id (#PCDATA)>
<!ELEMENT trained_model_id (#PCDATA)>
<!ELEMENT uri (path*)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT path (#PCDATA)>
```

**Figure 7.11:** The proposed ACE XML 2.0 update to the DTD of the Project file format. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.6 of the on-line sample file appendix.[204]

As can be seen by a comparison of Figures 7.10 and 7.11, most of the changes are in the specific terminology used in the element tags and in the order in which they appear. These changes are proposed for purposes of clarity and consistency with other ACE XML file formats. One of the few fundamental changes is the removal of the *ace_preferences_id* and *classifier_settings_id* elements. This was done because it is in general desirable to separate the ACE XML file formats from the ACE meta learning application or any other particular jMIR components. A new *uri* element is also added so that references can be made to external resources of any type. This *uri* element includes an optional *predicate* attribute in order to make it possible to specify RDF-like triples, such that the contents of the *uri* clause indicate the object, the clause containing the *uri* sub-clause is the subject, and the *predicate* attribute specifies the relationship between the two.

Another change that has been considered but rejected, at least for the moment, is the ability to list multiple Class Ontology files. The was not done since the merging of different ontologies could lead to significant inconsistencies if not supervised carefully, and it would probably be safer to require that such rare operations be performed manually.

[204] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

Although the Project file does make the use of multiple ACE XML files together significantly more convenient, it is an imperfect solution. Users must still maintain the individual files, and must be careful not to delete, rename or move them without making appropriate changes in the Project file.

In order to fully address this problem, it was decided to devise an ACE XML ZIP file format. This format is inspired by the Microsoft Office 2007 Open XML File Format,[205] which stores each Microsoft Office "document" as sets of XML files packaged into a single ZIP file. Similarly, ACE XML ZIP files consist of sets of ACE XML (or other) files that are packaged together into a single ZIP file.

This approach retains the advantages of maintaining separate files, as discussed in Sections 7.4 and 7.5, since each ACE XML file stored in an ACE XML ZIP file remains self-contained and can be easily extracted from the ZIP file and used on its own or with other projects. At the same time, this approach enables multiple related ACE XML files to be packaged into a single ZIP file, so no housekeeping of external files is required. ZIP files in particular are an especially appropriate format because there are many free applications and code libraries that can be used to access or store data in them.

Another significant advantage of using ZIP files is that they are compressed formats, which means that they can dramatically reduce space and bandwidth requirements. This is significant, as ACE XML files can be quite large, particularly in cases when many windowed features are extracted from large collections of data. Compression rates as high as 83% have been observed when compressing Feature Value files, although the amount of compression depends on the particular data that is being compressed.

Open-source code for using ACE ZIP files has already been implemented by Jessica Thompson. This code is integrated into the ACE code package, and can be accessed via the ACE command-line interface and general API. Work on incorporating this functionality into the ACE GUI as well is also currently underway.

Each ACE XML ZIP file is associated with a single Project file, which is either specified by the user when the ACE ZIP file is created, or auto-generated by the ACE ZIP processing code when ACE XML files are added to an existing ACE ZIP file. So, although an ACE ZIP file can hold many files of any type, each ACE ZIP file always has

[205] msdn.microsoft.com/en-us/library/aa338205.aspx

exactly one default ACE XML Project file contained within it and only one or zero Class Ontology files specified by this Project file. In order for this to work properly, each ACE ZIP file has a simple hidden file called *project.sp* automatically generated and added to it that specifies the name of the default ACE XML Project file for the ZIP file. Users do not ever interact directly with this file, however, or need to be aware of it.

An ACE XML ZIP file may be automatically generated from an ACE XML Project file using the ACE ZIP processing software. This software automatically packages all files referred to in the Project file, along with the Project file itself, into the ZIP file. Then, when the ZIP file is later opened, the Project file is automatically decompressed and parsed so that the files contained in the ZIP file can themselves be automatically accessed, decompressed and properly interpreted by the ACE code. When the project and other files contained in the ZIP file are decompressed into a new directory, the ACE zip-processing software automatically updates the Project file to reflect their new file paths.

Alternatively, users may specify a list of files or simply a directory containing ACE XML files. The ACE software will then automatically package the appropriate files into a new ACE ZIP file, along with an ACE XML Project file that is auto-generated based on the specified ACE XML files. If desired, the ACE API and command-line can also be used to add or extract individual or all files from ACE ZIP files.

ACE XML ZIP and Project file functionality will be incorporated into the jMIR components other than ACE once the ACE XML 2.0 specification is finalized via consultation with the NEMA researchers and the MIR community.

### 7.11.2 Proposed ACE XML 2.0 updates to Feature Value files

The DTD of the current ACE XML 1.1 Feature Value file format is shown in Figure 7.6, and the proposed updated DTD for the ACE XML 2.0 Feature Value format is shown in Figure 7.12. A sample ACE XML 2.0 Feature Value file is shown in Code Sample 7.7 of the on-line sample file appendix.[206] The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE

[206] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

XML 2.0 Feature Value format relative to the current ACE XML 1.1 version are as follows:[207]

- The *feature_vector_file* element is renamed to *ace_xml_feature_value_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *data_set* element is renamed to *instance* for the purposes of clarity and generality.
- The *data_set_id* element is renamed to *instance_id* for the purposes of consistency, clarity and generality.
- The *feature* element is renamed to *f* in order to reduce file size.
- The *name* element is renamed to *id* in order to reduce file size.
- The *section* element is renamed to *s* in order to reduce file size.
- The *start* and *stop* attributes are renamed *b* and *e* (abbreviations for *beginning* and *end*) in order to reduce file size, and are now obligatory in *s* elements in order to enforce proper file construction.
- Features that correspond to a precise time (or other) coordinate value in an instance rather than intervals of time (or other) coordinates or instances as a whole can be specified with the new *precise_coord* element and its associated *coord* attribute.
- The new optional *coord_units* element can be used in each instance to specify the units used for the coordinate indicators for both sub-sections of and precise coordinates in instances.
- The new optional *extractor* element can be used to specify the name of the feature extraction software used to extract each feature for an instance. A separate *extractor* clause is used for each feature. The contents of an *extractor* clause

[207] Several of the tags in the ACE XML 2.0 Feature Value file have been abbreviated. Feature Vector files in particular have a tendency to grow very long, especially when there are instances with many analysis windows and when many features are extracted. So, although in general abbreviations have been avoided in ACE XML for the sake of clarity, some abbreviations have been used in Feature Value files for those tags that are likely to be repeated often in order to avoid exorbitant file sizes.

indicate the name of the feature extractor, and the obligatory *fname* attribute indicates the name of the feature that is to be associated with this extractor. This arrangement allows scenarios where different feature extractors are used to extract the same feature for different instances, as well as scenarios where different features are extracted by different feature extractors for the same instance.

- Feature values consisting of arrays with an arbitrary number of dimensions may now be expressed, as compared to the ACE XML 1.1 limitation to only one-dimensional vectors of feature vales. Sparse arrays, or arrays missing some entries, are now supported as well, something that can be important for space efficient and flexible data representation. ACE XML 2.0 currently supports four alternative approaches to representing feature values and arrays, each with its own relative strengths and weaknesses with respect to the number of dimensions that can be represented, file size, ability to efficiently represent sparse data and human readability:
    - In the case of feature values consisting of only one value or feature vectors consisting of only one dimension, a methodology similar to the one that was used in ACE XML 1.1 (i.e. *f* clauses containing the *v* element) may be used in ACE XML 2.0 as well.
    - Arrays with any number of dimensions may be expressed using JSON (*JavaScript Object Notation*)[208] array notation. JSON is a well-established and relatively human readable text-based data interchange format for representing simple data structures. JSON arrays are expressed using simple square bracket notation, enclosed in *vj* clauses in ACE XML 2.0. So, for example, a feature vector of size three consisting of the numbers one, two and three would be represented as *<vj>[1,2,3]</vj>*. JSON arrays can be nested in order to represent arrays of arbitrary dimensionality. So, for example, a table with two identical rows each containing the values one, two and three would be represented as *<vj>[[1,2,3],[1,2,3]]<vj>*. A similar approach could have been achieved

[208] json.org

by using nested XML elements, but the JSON representation is more compact and more human readable for large arrays. There are also JSON parsing libraries available in many languages that can parse such arrays quickly, which offloads some of the work from that would otherwise need to be performed by an ACE XML parser. A disadvantage of the JSON approach, however, is that JSON is not ideally suited to efficiently representing sparse arrays. Also, JSON is less human readable than some of the alternative approaches. Ultimately, however, it is a compromise that enables potentially very large arrays to be represented relatively compactly while still being relatively readable, at least compared to binary data.

- Explicitly indexed arrays may be used as an alternative representation in the case of feature values consisting of only one number, feature vectors consisting of one dimension or feature arrays consisting of two to ten dimensions. This approach involves specifying coordinates using the *d0* to d*9* attributes in *vd* clauses, as an alternative to *v* clauses. If there is only one coordinate (i.e., a feature vector), then only the *d0* attribute would be used, if there is a three-dimensional array then the d0, d1 and d2 attributes would be used, and so on. This approach is moderately space efficient, can represent sparse arrays and is easily human readable. There is a limitation to only ten dimensions, but each of these may consist of vectors of any size, and very few features used in MIR need arrays with more than ten dimensions. Although it would be ideal to have such an approach for N dimensions, it is not possible to specify an arbitrary number of dimension attributes in an XML DTD schema. To give an example, the *JSON* feature vector of *[1,2,3]* would be represented as *<vd d0="0">1</vd><vd d0="1">2</vd><vd d0="2">3</vd>,* and the JSON array of *[[1,2,3],[1,2,3]]* would be represented as *<vd d0="0" d1="0">1</vd><vd d0="0" d1="1">2</vd><vd d0="0" d1="2">3</vd><vd d0="1" d1="0">1</vd><vd d0="1" d1="1">2</vd><vd d0="1" d1="2">3</vd>.*

    - The final option permitted by ACE XML 2.0 is to represent arrays with any number of dimensions using *vs* clauses. Each *vs* clause contains one *d* element for each dimension, and this *d* element is used to specify the coordinate value its corresponding dimension. Each *vs* clause also contains a single *v* clause to specify the feature value for the array at the corresponding coordinate. To give an example, the element of the *JSON* feature array *[[1,2,3],[4,5,6]]* with a value of 6 would be represented as *<vs><d>1</d><d>2</d><v>6</v></vs>*. This approach has the advantage that it can be used to express arrays with any number of dimensions and, unlike the JSON approach, can also efficiently represent sparse arrays as well as represent data in a more human readable way. It is significantly less compact than the JSON approach for complete arrays, however, and the ACE XML encoder must ensure that the correct number of *d* elements are present for each value and that they consistently appear in the correct order.

- Data types may now optionally be explicitly specified for each feature via the optional *type* attribute of the *f* (formerly *feature*) element. Types of *int, double, float, complex* and *string* are permitted. Although this typing is not necessary for the jMIR components, it is sometimes necessary for other applications, so it is useful to incorporate it into the ACE XML formats so that it can be used when needed. The type will typically be assumed to be *double* if it is not specified in any given *f* clause, but this not an intrinsic assumption of the Feature Value format. It is important to note that feature types may also be specified in an associated Feature Description file, in which case the feature types in the Feature Value file should either correspond to the types specified in the Feature Description file or, for the sake of brevity, be omitted in the Feature Value file.

- The *comments* element is now optional.

- It is now possible to specify other files that are related to the Feature Value file using the *related_resources* element. These can either be explicitly referenced Feature Value, Feature Description, Instance Label, Class Ontology or Project files, respectively referenced via *feature_value_file, feature_description_file, instance_label_file, class_ontology_file* or *project_file* elements, or they can be resources of arbitrary types referenced with *uri* elements. Note that references referred to via a *related_resources* element are for informal informational purposes only from the perspective of the jMIR components, and are *not* substitutes for references in ACE XML Project (see Section 7.11.1).

- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any instance (*instance*), section (*s*), precise coordinate (*precise_coord*) or feature (*f*) clauses. These are not intended for use by the jMIR components, but may be used by other software to access external resources whenever appropriate.

### 7.11.3 Proposed ACE XML 2.0 updates to Feature Description files

The DTD of the current ACE XML 1.1 Feature Description file format is shown in Figure 7.7, and the proposed updated DTD for the ACE XML 2.0 Feature Description format is shown in Figure 7.13. A sample ACE XML 2.0 Feature Description file is also shown in Code Sample 7.8 of the on-line sample file appendix.[209] The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Feature Description format relative to the current ACE XML 1.1 version are as follows:

- The *feature_key_file* element is renamed to *ace_xml_feature_description_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.

- The *name* element is renamed to *fid* to make it consistent with the Feature Value format.

- The *is_sequential* element is replaced by the *scope* element. The old *is_sequential* element could only be used to specify whether a feature could be extracted only over a whole instance or only over sub-sections of an instance. The new *scope* element makes it possible to specify that a feature may be extracted for an instance as a whole, for sub-sections of an instance, from only a precise point in an instance (e.g., a moment in time) or from any combination of these. This

[209] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

```
<!ELEMENT ace_xml_feature_value_file_2_0 (comments?, related_resources?,
                                          instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    uri*,
                    extractor*,
                    coord_units?,
                    s*,
                    precise_coord*,
                    f*)>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT extractor (#PCDATA)>
<!ATTLIST extractor fname CDATA #REQUIRED>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT s (uri*,
             f+)>
<!ATTLIST s b CDATA #REQUIRED
           e CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         f+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT f (fid,
             uri*,
             (v+ | vd+ | vs+ | vj))>
<!ATTLIST f type (int | double | float | complex | string) #IMPLIED>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT v (#PCDATA)>
<!ELEMENT vd (#PCDATA)>
<!ATTLIST vd d0 CDATA #REQUIRED d1 CDATA #IMPLIED d2 CDATA #IMPLIED
             d3 CDATA #IMPLIED d4 CDATA #IMPLIED d5 CDATA #IMPLIED
             d6 CDATA #IMPLIED d7 CDATA #IMPLIED d8 CDATA #IMPLIED
             d9 CDATA #IMPLIED>
<!ELEMENT vs (d+,
              v)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT vj (#PCDATA)>
```

**Figure 7.12:** The proposed ACE XML 2.0 update to the DTD of the Feature Value file format. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.7 of the on-line sample file appendix.[210]

[210] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

information is expressed via the *overall, sub_section* and *precise_coord* attributes respectively, which may each have values of either *true* or *false*. This information must all be specified for all features. It is possible to enter comment data in the *scope* clause, but this data will have no technical meaning.

- The *parallel_dimensions* element is replaced by the *dimensionality* element in order to accommodate the new ability to represent feature value arrays of arbitrary dimensionality in ACE XML 2.0 Feature Value files, as opposed to the ACE XML 1.0 limitation to one-dimensional feature vectors. The old *parallel_dimensions* element was only used to specify the size of feature vectors, but the *dimensionality* element is used to specify the number of different dimensions of the coordinate system for the feature (e.g., one for a feature vector, two for a table structure, etc.) as well as the size of each of the dimensions. The *orthogonal_dimensions* attribute indicates the former, and *size* clauses within the *dimensionality* clause are used to indicate the size of each of these (e.g., one size clause each for the number of rows and the number of columns in a table structure). The *dimensionality* element may also be omitted if a particular feature can have variable number of coordinate dimensions, and *size* clauses may be omitted as well if they also vary.

- The optional *data_type* element and its *type* attribute are added in order to allow the specification of the particular data type for a given feature. Specifically, the permitted types are *int, double, float, complex* and *string,* and one of these must be specified in the *type* tag. Although this typing is not necessary for the jMIR components, it is sometimes necessary for other applications, so it is useful to incorporate the option of using it into the ACE XML formats so that it can be used when needed. The type will typically be assumed to be *double* if it is not specified in any given *feature* clause, but this not an intrinsic assumption of the Feature Description format. Although data types may be specified in Feature Value files, it is preferable to do so in Feature Description files, which take priority. Note that comments may be entered in the *data_type* clause itself, but they do not have any technical meaning.

- It is now possible to specify feature parameters using the optional *parameter* element and its associated *parameter_id, description* and *value* elements. This could be used for specifying the roll-off point for the Spectral Roll-off feature, for example. A separate *parameter* clause is used for each parameter of a feature, the *parameter_id* element is used to identify the parameter uniquely, the *description* element can be used to describe the parameter in general, and the *value* element can be used to express numerical parameter values.

- Global parameters may also be specified for all features in the Feature Description file using the *global_parameter* element. This is useful for specifying overall pre-processing of audio files before features are extracted, for example, such as down sampling or normalization. The mechanics of the global_parameter are the same as those of the *parameter* element.

- A new optional *related_feature* clause may be used to specify other features that are related to any given feature. This could be used, for example, to note that one feature is an alternative implementation of another. The *fid* element in the related feature clause should be used to specify the name of a feature specified in the *fid* clause of another feature. The *relation_id* element can be used to specify a specific externally defined type of relationship, and the *explanation* element can be used to provide a qualitative description of the relationship.

- The *comments* element is now optional.

- It is now possible to specify other files that are related to the Feature Description file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in Section 7.11.2.

- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any *feature, dimensionality, parameter* or *related_feature* clause. These are not used by the jMIR components, but may be used by other software to access external resources if appropriate.

```
<!ELEMENT ace_xml_feature_description_file_2_0 (comments?,
                                                 related_resources?,
                                                 global_parameter*,
                                                 feature+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT feature (fid,
                   description?,
                   related_feature*,
                   uri*,
                   scope,
                   dimensionality?,
                   data_type?,
                   parameter*)>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT related_feature (fid,
                           relation_id?,
                           uri*,
                           explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT scope (#PCDATA)>
<!ATTLIST scope overall (true|false) #REQUIRED
                sub_section (true|false) #REQUIRED
                precise_coord (true|false) #REQUIRED>
<!ELEMENT dimensionality (uri*,
                          size*)>
<!ATTLIST dimensionality orthogonal_dimensions CDATA #REQUIRED>
<!ELEMENT size (#PCDATA)>
<!ELEMENT data_type (#PCDATA)>
<!ATTLIST data_type type (int | double | float | complex | string) #REQUIRED>
<!ELEMENT global_parameter (parameter_id,
                            uri*,
                            description?,
                            value?)>
<!ELEMENT parameter (parameter_id,
                     uri*,
                     description?,
                     value?)>
<!ELEMENT parameter_id (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

**Figure 7.13:** The proposed ACE XML 2.0 update to the DTD of the Feature Description file format. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.8 of the on-line sample file appendix.

### 7.11.4 Proposed ACE XML 2.0 updates to Instance Label files

The DTD of the current ACE XML 1.1 Instance Label file format is shown in Figure 7.8, and the proposed updated DTD for the ACE XML 2.0 Instance Label format is shown in Figure 7.14. A sample ACE XML 2.0 Instance Label file is also shown in Code Sample 7.9 of the on-line sample file appendix.[211] The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Instance Label format relative to the current ACE XML 1.1 version are as follows:

- The *classifications_file* element is renamed to *ace_xml_instance_label_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *data_set* element is renamed to *instance* for the purposes of clarity and generality.
- The *data_set_id* element is renamed to *instance_id* for the purposes of consistency, clarity and generality.
- The *info_type* attribute is renamed to *info_id* for the purpose of consistency with other element and attribute identifiers. Also, *misc_info* clauses now contain *info_id* and *info* elements, and there are no longer any attributes for the *misc_info* element. This is to enable the addition an arbitrary number of *uri* annotations to *misc_info* clauses, as noted below.
- For similar reasons, class labels are now specified within a *class_id* element contained in a *class* clause.
- The *role* element is now an optional attribute of the *instance* element instead of an element itself. This makes it possible to explicitly constrain its possible values to *training, testing* or *predicted.*
- A new optional *related_instance* clause may be used to specify other instances that are related to any given instance. This could be used, for example, to note that one recording is a cover of another. The *instance_id* element in the related

[211] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

instance clause should be used to specify the name of an instance specified in the *instance_id* clause of another feature. The *relation_id* element can be used to specify a specific externally defined type of relationship, and the *explanation* element can be used to provide a qualitative description of the relationship.

- The *start* and *stop* elements for denoting coordinate ranges within an instance are replaced with the *begin* and *end* attributes of the *section* element. This change is in order to maintain consistency with the *b* and *e* attributes of ACE XML 2.0 Feature Value files.
- Class labels that correspond to a precise time (or other) coordinate rather than intervals of time (or other) coordinates or instances as a whole can be specified with the new *precise_coord* element and its associated *coord* attribute.
- The new optional *coord_units* element can be used in each instance to specify the units used for the coordinate indicators for both sub-sections of and precise coordinates in instances.
- The *classification* element is removed for the sake of improving file simplicity. Section labelling and overall instance labelling now simply occur directly within an *instance* clause rather than within a *classification* clause within an *instance* clause.
- The *class* element has a new *weight* attribute that can be use to specify proportional support for a class relative to other classes when more than one class apply. So, for example, a given musical recording might be labelled with the *Blues* genre with a specified weight of *2* as well as with the *Jazz* genre with a specified weight of *1.* Depending on context, this could be intended to mean either that the recording is a member of both the *Blues* and *Jazz* genres, but the influence of the former is twice that of the latter, or it could mean that a classifier is unsure whether the piece is *Blues* or *Jazz,* but believes that the former label is twice as likely as the latter. If the *weight* attribute is not specified for a class, it is assigned a value of *1* by default. All weight values are proportional, so the absolute value of

a weight has no meaning other than its value relative to the weights of other class labels with the same scope.

- The optional *source_comment* attribute may now be used with the class element. This permits the specification of whether an instance was labeled by a machine, human expert, survey, etc.

- The *comments* element is now optional.

- It is now possible to specify other files that are related to the Instance Label file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in Section 7.11.2.

- Optional *uri* elements (and their associated *predicate* attributes) may be added to any *instance, related_instance, misc_info, section, precise_coord* or *class* clause. These are not used by the jMIR components, but may be used by other software to access external resources when appropriate.

### 7.11.5 Proposed ACE XML 2.0 updates to Class Ontology files

The DTD of the current ACE XML 1.1 Class Ontology file format is shown in Figure 7.9, and the proposed updated DTD for the ACE XML 2.0 Class Ontology format is shown in Figure 7.15. A sample ACE XML 2.0 Class Ontology file is also shown in Code Sample 7.10 of the on-line sample file appendix.[212] The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Class Ontology format relative to the current ACE XML 1.1 version are as follows:

- The *taxonomy_file* element is renamed to *ace_xml_class_ontology_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.

- The *class_name* element is renamed to *class_id* in order to make it consistent with the naming conventions used in the other ACE XML 2.0 formats.

[212] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

```
<!ELEMENT ace_xml_instance_label_file_2_0 (comments?,
                                           related_resources?,
                                           instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    misc_info*,
                    related_instance*,
                    uri*,
                    coord_units?,
                    section*,
                    precise_coord*,
                    class*)>
<!ATTLIST instance role (training | testing | predicted) #IMPLIED>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT related_instance (instance_id,
                            relation_id?,
                            uri*,
                            explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT section (uri*,
                   class+)>
<!ATTLIST section begin CDATA #REQUIRED
                  end CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         class+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT class (class_id,
                 uri*)>
<!ATTLIST class weight CDATA "1">
<!ATTLIST class source_comment CDATA #IMPLIED>
<!ELEMENT class_id (#PCDATA)>
```

**Figure 7.14:** The proposed ACE XML 2.0 update to the DTD of the Instance Label file format. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.9 of the on-line sample file appendix.

- The *parent_classs* element is removed because it made an implied hierarchical organization of classes obligatory, which is not always appropriate. Information about each class is now contained in a *class* clause, regardless of the presence or absence of a hierarchical structure.

- Since it is sometimes useful to be able to specify hierarchical class structuring, the optional *sub_class* element is repurposed so that it can be used to reference one or more other classes that are hierarchical subordinates to the class whose clause contains the *sub_class* element. Such sub-classes must now also be separately declared in their own *class* clauses. Tree structures can be built by referring to subordinate classes using *sub_class* clauses, then referring to further subordinate classes at the next depth level of the tree in the *sub_class* clauses of these classes, and so on.

- As an alternative to hierarchical class structuring, the ACE XML 2.0 Class Ontology format now allows more general ontological relationships to be specified between classes using the *related_class* element. A relationship specified from one class to another with this element is unidirectional, unless the same relationship is explicitly specified from the second class to the first class as well in the second class' declaration. The *relation_id* element may be used to specify the meaning of the relationship using some externally defined keyword if desired.

- Weights may be assigned to both *related_class* and *sub_class* elements using the *weight* attribute, which defaults to 1 unless specified. Relating to this, the global *weights_relative* attribute of the *ace_xml_class_ontology_file_2_0* element must be specified as either *true* or *false.* If it is true, then the weights for each class will be normalized upon parsing, if not they will be interpreted as is.

- The *explanation* element can be used to provide qualitative explanations of any class connections in the class ontology.

- The *misc_info* element may now be used to specify miscellaneous metadata about each class. Each *misc_info* clause contains *info_id* and *info* elements to specify some externally defined unique keyword for the metadata field and the metadata itself, respectively.

- The *comments* element is now optional.
- It is now possible to specify other files that are related to the Class Ontology file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in Section 7.11.2.
- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any *class, related_class* or *sub_class* clause. These are not used by the jMIR components, but may be used by other software to access external resources when appropriate.

### 7.11.6 Summary abstraction of ACE XML 2.0

An abstraction of the overall data model and serialisation of ACE XML 2.0 is illustrated graphically in Figure 7.16. The following conventions are used:

- Nodes:
    - Rectangular nodes represent entities.
    - Diamond nodes represent relationships.
    - Oval nodes represent attributes.
    - Exclamation marks close attributes that comprise an entity or relationship's primary key.
    - Question marks close attributes that are optional.[213]
    - Double borders indicate that an entity or relationship may be associated with one or more RDF triples.

[213] Mandatory attributes may, however, have default values in the case of null entries. This diagram omits this type of information.

```
<!ELEMENT ace_xml_class_ontology_file_2_0 (comments?,
                                           related_resources?,
                                           class+)>
<!ATTLIST ace_xml_class_ontology_file_2_0 weights_relative (true|false)
                                                            #REQUIRED>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT class (class_id,
                 misc_info*,
                 uri*,
                 related_class*,
                 sub_class*)>
<!ELEMENT class_id (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT related_class (class_id,
                         relation_id?,
                         uri*,
                         explanation?)>
<!ATTLIST related_class weight CDATA "1">
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT sub_class (class_id,
                     relation_id?,
                     uri*,
                     explanation?)>
<!ATTLIST sub_class weight CDATA "1">
```

**Figure 7.15:** The proposed ACE XML 2.0 update to the DTD of the Class Ontology file format. See Section 7.2.3 for an explanation of XML DTDs. A sample file based on this DTD is shown in Code Sample 7.10 of the on-line sample file appendix.[214]

[214] www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf

- Edges:
    - Edges define the structure.
    - Solid edges connect relationships to their component entities and are labelled with the cardinality of the relationship for each component entity.
    - Arrowed edges point to entities that are defined weakly, which is to say that their primary keys include the primary keys of the other entities in the relationship (see Pin-Shan Chen 1976).
    - Bold edges indicate relationships of an entity with itself and are labelled with both relevant cardinalities.
    - Dotted edges connect entities and relationships to their attributes.
- Colours (for colour versions of this document)
    - Colours give information related to the serialisation.
    - Dark brown designates entities and relationships that appear in Feature Description files only.
    - Medium brown designates entities and relationships that appear in Feature Description and Feature Value files.
    - Light brown designates entities and relationships that appear in Feature Value files only.
    - Light grey designates entities and relationships that appear in Feature Value and Instance Label files.
    - Light blue-green designates entities and relationships that appear in Instance Label files only.
    - Medium blue-green designates entities and relationships that appear in Instance Label and Class Ontology files.
    - Dark blue-green designates entities and relationships that appear in Class Ontology files only.

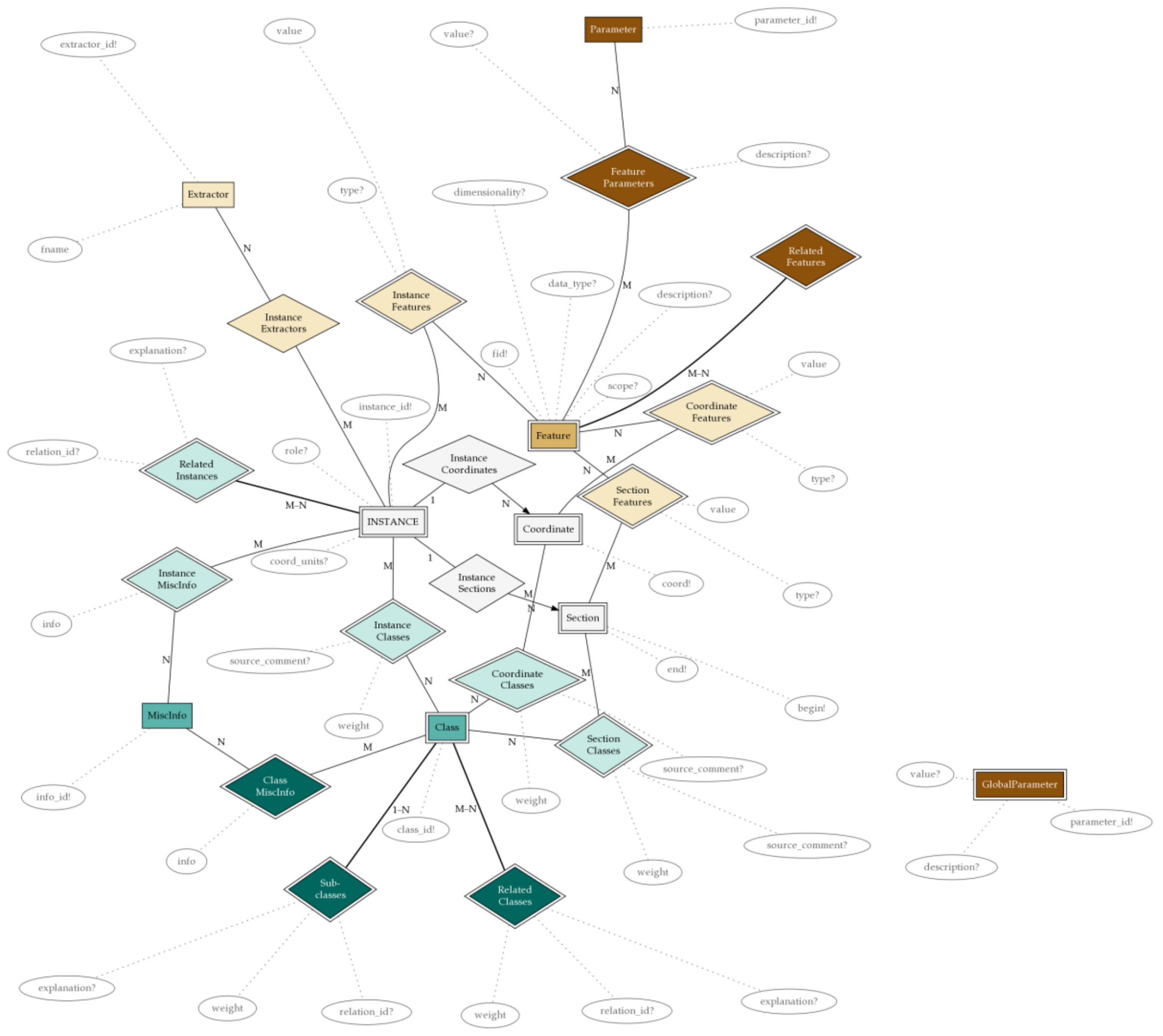


**Figure 7.16:** An overall visualisation of the proposed ACE XML 2.0 data model and serialisation. Conventions are explained in the text.

## *7.12 Summary of original contributions*

This chapter provides a critical analysis and overview of existing file formats that are used by MIR researchers for data mining and music classification. It also provides an original and previously lacking set of design priorities for consideration when devising new file formats in this domain.

The four original ACE XML 1.1 file formats are presented as an implementation of these design priorities. The jMIR components can all use these file formats to communicate with each other, and a general API is provided as part of the ACE package so that ACE XML functionality can be easily integrated into other software. General ACE XML file processing utilities are also provided in the jMIRUtilities software.

Original prototypes for the four ACE XML 2.0 file formats are also presented to the MIR research community for general discussion and comment. These formats expand upon the ACE XML 1.1 by adding further expressive power and flexibility, and it is hoped that the MIR community will collaborate to improve upon them and eventually adopt them.

## *7.13 Future research*

The clear priority for future research is to collect input from the MIR community on changes and improvements to the ACE XML 2.0 formats.

One specific issue that needs to be addressed is that Feature Value files can end up being very large, particularly when many features are extracted for small windows over many instances. Although ACE ZIP packaging does address this issue to an extent, it in turn introduces additional processing overhead when compressing and decompressing the files. One solution would be to represent feature values, including arrays, in binary rather than in text. Unfortunately, this would entirely undermine ACE XML's design philosophy of permitting human readability. Furthermore, there are many alternative ways of representing values and arrays in binary, which could cause incompatibilities if different users encode binary in different ways and then distribute the Feature Value files to others. This could also pose problems with respect to data longevity. One final related issue is that XML validation processing cannot be applied to binary representations, thereby increasing the work that must be performed directly by ACE XML parsers. All of this having been said, it is always desirable to avoid excessively large files, so this is something that needs to be considered further.

The representation of feature values consisting of N-dimensional arrays is another issue that needs to be considered further. All of the options supported by ACE XML 2.0 can require a significant amount of space to represent large N-dimensional arrays, so the addition of a binary representation option to the specification might be useful in cases where this could be a concern and where human readability is not a priority.

Another issue that would be appropriate to pursue would be the design of some external file format for specifying a vocabulary that could be defined with respect to units that are specified using the *coord_units* element. This would, for example, provide a way to automatically make a recording annotated with times stamps based on milliseconds compatible with another annotated based on samples (if the sampling rate is known, of course).

It would also be useful to take advantage of existing metadata standards in general. One sample approach might be to integrate Dublin Core RDF functionality into the ACE XML formats.[215]

Complex numbers also represent something that needs to be considered further. Under the current implementation, complex numbers can be simply represented as feature vectors of size 2. Unfortunately, this does not explicitly distinguish them from any other feature vector of size 2. Ideally, one would like to have some way of typing complex numbers in the same way that doubles or strings can be typed. Unfortunately, there is no complex primitive type in XML, nor is there a complex primitive in many of the most common programming languages, including Java. Furthermore, with respect to array representation, JSON does not allow the explicit use of complex numbers in arrays, beyond the current implicit practice of using general sub-arrays of size 2. This limitation of JSON is in itself a strong argument, in particular, against using an explicit complex number type. Furthermore, in most cases feature values are simply treated as features during machine learning, so it is often not relevant if a complex feature value is explicitly typed as such. Nevertheless, it could be important to more sophisticated learning techniques to have an explicit complex data type, so further though needs to be put into incorporating explicit complex typing into ACE XML Feature Value and Feature Description files, in both Cartesian and polar forms.

Another issue to address is multilingual support. XML is generally Unicode-based, so there is built-in support for many character sets, but testing of the ACE XML processing software to date has focused on English and French data, a scope that needs to be expanded to ensure that the file formats are ready for wide international use. Further thought is needed in this area.

[215] dublincore.org/schemas/rdfs/ and dublincore.org/documents/dc-rdf/index.shtml

It would also be helpful to define specific standards for setting unique keys for the ACE XML ID fields that are used to merge data stored in the four different file formats. The onus is currently on the data encoder to ensure uniqueness, something that could potentially result in missed correspondences and conflicts. One possible solution might be to use something such as MusicBrainz[216] IDs as primary keys, for example, but solutions like this tend to focus only on specific types of data, namely audio in this particular case. Extracting MusicBrainz IDs for something like MIDI files is not tenable. Further thought is needed in this area as well.

There would also be advantages to writing versions of each of the ACE XML DTDs using an alternative XML schema that would specify the same formats using an alternative methodology. Such schemas would co-exist with the current DTDs, due to the advantages of DTDs expressed in Section 7.5 with respect to accessibility to new ACE XML users. The main advantage offered by the use of an alternative more expressive schema format is the offloading of some of the file validation load from the ACE XML parsers to the general XML parsing libraries, since constraints on what could validly be contained in XML clauses could be more precisely defined.

Once the ACE XML 2.0 file formats are finalized, the next step will be to update the ACE parsers, processing utilities and data structures. The ACE API is ready as is for the porting of ACE XML 1.1 functionality to external software, but significant updates will be necessary to make it ACE XML 2.0 ready.

The implementation of reading, writing and processing ports for specific existing widely used MIR systems like Marsyas and CLAM and for general programming environments like Matlab and C++ will likely do much to encourage the wide adoption of the ACE XML formats by making them more easily accessible. Translation software will also be needed to translate information stored in existing formats, including ACE XML 1.1, into ACE XML 2.0.

Another area for future research is the development of a standardized way of storing lyrics for processing. Lyrics represent a potentially very rich source of information that is currently underexploited in MIR, and it might be useful to develop a new ACE XML

[216] musicbrainz.org

format specifically designed for storing lyrics and making it easy to extract various kinds of features from them.

It would also be helpful to develop a standardized file format for specifying queries that could be applied to data stored in ACE XML files. There are many general possible sources of inspiration, such as SQL, Z39.50[217] and FRBR.[218]

[217] www.loc.gov/z3950/agency/
[218] www.ifla.org/VII/s13/wgfrbr/index.htm

# 8. jMusicMetaManager, the Bodhidharma MIDI dataset, Codaich, SAC and jMIRUtilities: Building, managing and using music research datasets

## *8.1 Overview and background information*

### 8.1.1 Introduction

Labelled musical datasets play an essential role in almost all research on automatic music classification. The models that are built by supervised machine learning algorithms are all learned from such datasets, and even the best algorithm is ultimately limited by the quality of this data. Furthermore, different algorithms are evaluated and compared using test data drawn from such datasets, with the consequence that poorly labelled or non-representative datasets can result in inaccurate evaluations of trained models. Even outside the realm of research on automatic music classification, high-quality datasets can provide valuable resources for empirical musicological and music theoretical studies of many kinds.

For the purposes of this document, the term *labelled musical dataset* is used to refer specifically to any collection of musical recordings, which may be either audio or symbolic, each of which is annotated with one or more metadata labels that specify model classifications of some kind. Examples include collections of MP3 or MIDI recordings that are each labelled with labels specifying information such as artist, mood, genre, etc. The metadata labels may also be more directly content-related, such as time-stamped annotations of section divisions or note onsets in audio files.

Musical datasets were often treated almost as an afterthought in early MIR research, with the bulk of the emphasis placed on feature extraction and classification algorithms. It was not unusual for labs to construct training and testing datasets simply by combining the personal music collections of a few graduate students, with all labelling done as quickly as possible by a single individual.

Fortunately, as the MIR field matures, researchers are increasingly realising the limitations of such simplistic datasets, and are beginning to perform more expansive studies that require much larger, more varied and carefully labelled collections. Such approaches are ultimately necessary in order to develop and validate MIR tools that can be applied to the vast and varied universe of music that exists outside of the lab. Developing effective frameworks for building and maintaining large musical databases is also becoming increasingly important as libraries digitize their collections and on-line commercial databases continue to grow.

The need for high-quality datasets that can be shared between research labs has become particularly evident during the design and implementation of MIREX (Music Information Retrieval Evaluation eXchange) (Downie 2006), a yearly event where different MIR algorithms are evaluated against each other in a variety of research domains. Although the training and testing datasets that have been used in MIREX have varied across years and task categories, the general approach has been to either have researchers submit their own datasets, often with questionable legality and with varying quality, or to use in-house datasets at the University of Illinois at Urbana-Champaign, where MIREX is held, that are not distributable outside researchers.

The controversy that has occasionally broken out over the reliability of algorithm rankings based on such datasets has helped to cause the MIR community to begin to recognize that training and testing datasets deserve much more attention than they have traditionally received. Serious thought is now being given to issues such as attaining realistic collection sizes and diversity, arriving at and taking advantage of ontological class structuring, dealing with serious problems with metadata quality and consistency, developing good instance labelling and selection methodologies and overcoming limitations on sharing datasets imposed by copyright laws.

Such issues are discussed in some detail in Section 8.2, which ends with a list of recommended general guidelines for building research datasets and overcoming many of the problems that have to date hindered efforts to build, use and share high-quality research datasets. Section 8.3 reviews a number of MIR research datasets that have been published by others, and briefly discusses some of their strengths and weaknesses.

This chapter also describes two jMIR software components for managing, improving and processing datasets, and three original MIR research datasets. jMusicMetaManager (Section 8.4) is a tool for statistically profiling and inventorying music collections, as well as automatically finding metadata errors, inconsistencies and redundancies in them.

jMIRUtilities (Section 8.8) is a set of utilities for performing miscellaneous tasks such as labelling instances and merging features extracted from different datasets.

The Bodhidharma MIDI dataset (Section 8.5) is a collection of 950 MIDI files belonging to 38 different genres of music. Codaich (Section 8.6) is a collection of 26,420 MP3s belonging to 55 different genres that has had its metadata carefully reviewed both manually and using jMusicMetaManager. OMEN, a proposed framework for legally distributing features extracted from collections such as Codaich, is briefly reviewed in Section 8.6. Finally, SAC (Section 8.7) is a matched collection of 250 MIDI recordings, 250 MP3 recordings and 250 sets of metadata tags that can be used as a basis for cultural feature extraction using software such as jWebMiner (Chapter 5). SAC is intended for research that combines and compares features extracted from audio, symbolic and cultural sources.

These three datasets were originally collected for training and evaluating the jMIR components and for performing experiments such as those described in Chapter 9. They were, however, all conceptualized, designed and collected with the long-term goal of potentially being shared among different MIR research groups for use in a much wider range of research projects.

### 8.1.2 Central concepts and terminology

Before continuing, it is useful to first briefly review or introduce certain key terms and concepts.

Musical datasets typically consist of one or more types of data: *audio data, symbolic data* and *cultural data*. Audio data refers to digital sound samples stored in files (e.g., MP3s, AACs, WAV files, etc.), symbolic data consists of sets of instructions that can be used by a computer synthesizer or human to approximately reconstruct music (e.g., MIDI files, printed scores, Humdrum files, etc.) and cultural data refers to information about pieces of music other than the actual sound itself (e.g., listener playlists, album reviews, Billboard sales statistics, etc.).

*Metadata* plays an essential role in most datasets. Metadata is essentially data about data. For example, if one is building a system for classifying audio recordings, then the audio samples themselves would be the central data of interest, and all other information about them would be considered to be metadata. So, to continue this example, the composer, year of recording and mood of each recording would all be considered metadata, to give just a few examples.

Each category of metadata is called a *field*, and the particular value or values of a field for a particular recording is called an *entry*. For example, *Musical Genre* and *Recording Title* are two common fields, and the particular title of a single recording would be the entry for that field for that recording.

Metadata is not limited to text data. The album art associated with a recording or a musical score providing a transcription of a piece are just two examples of other kinds of metadata. Even features extracted from audio samples are technically metadata, although these are typically treated separately as a matter of organizational convenience.

The labelled datasets that are used in machine learning are often referred to as *ground-truth.* This means that their labels are effectively considered to be the accepted truth which is used to train and test classifiers. Each separate unit of ground-truth, such as an individual recording, is known as an *instance*. Each instance is typically labelled with a model category name, such as a particular mood or genre, that it is hoped a classifier will learn to map features to. Each such category is commonly referred to as a *class*.

There are often certain relationships that exist between classes. For instance, if one is dealing with genres, then some genres are often sub-genres of other genres (e.g., Nashville is a sub-genre of Country). Structures that indicate relationships between different classes are often referred to as *ontologies* or *taxonomies*. Although these two terms have been used with a variety of different meaning in different publications, and have sometimes been used interchangeably, it is useful to distinguish between them in a precise way for the sake of clarity. It has been decided to base the use of the terms in this document on observations made by Van Rees (2003), namely that a *taxonomy* is a strictly hierarchical structuring of classes, while an *ontology* is any general structuring of classes.[219] A taxonomy is thus a special type of ontology.

During the training and testing of a classifier the ground-truth is typically divided into a *training set,* which is used for training the algorithm, and a *test set,* which is used to test the quality of the trained model in order to ensure that overtraining has been avoided. If

[219] Van Rees also proposes a specific interpretation of the word *classification* that is not adopted here, as doing so would only create confusion in this context.

multiple classifiers are to be compared, then two test sets are needed, namely a *validation set* that is used to test and compare each individual classifier after training and a *publication set* that is used to retest the classifier that is selected as the best based on performance on the validation set. There must not be any overlap between any of these sets, as this would leave vulnerabilities to overtraining untested.

Sets of ordered text characters are referred to in computer science as *strings.* The title of a recording is thus a string, for example, as is this sentence, any word in it and the period ending this sentence. As this makes evident, strings can be broken into sets of substrings, each of which is also a string. These substrings are sometimes called *tokens.* A string that does not contain any characters is referred to as an *empty string*.

### 8.1.3 Essentials of copyright law

Copyright law plays a controlling role in defining the extent to which and the circumstances under which music can legally be copied or transferred between individuals and organizations. As such, copyright laws have a large impact on the sharing and reuse of musical research datasets. It is therefore useful to briefly review some of the central aspects of musical copyright law and associated legislated *intellectual property rights* before proceeding.

Unfortunately, the legislation governing musical intellectual property rights tends to vary widely between jurisdictions. Although many countries are signatories to a range of *World Intellectual Property Organization (WIPO)* treaties that, in theory, attempt to standardize copyright laws across international borders as much as possible, there is in practice a great deal of diversity between countries in terms of both how permissive their laws are and the extent to which they enforce those laws. So, for example, the United States of America has adopted a relatively restrictive stance on music since the *Digital Millennium Copyright Act (DMA)* was passed in 1998, and has attempted to enforce it fairly vigorously. In contrast, pirates in Vietnam are permitted to copy and sell copyrighted music with relative impunity.

Further complicating matters, the rights associated with musical compositions, individual performances and actual recordings of performances are treated separately by many countries. Not only do different administrative bodies typically govern each of these rights, even within a single county, but they often do so based on entirely different legislative documents.

Unfortunately, copyright legislation can be particularly ambiguous with respect to music. The fundamental legislative frameworks administering intellectual property rights were designed with printed materials in mind, and were conceptualized long before recorded music became commercially available, to say nothing of digitized music.

Some jurisdictions have frameworks providing academic institutions with special rights, while others do not. For example, the United States of America and Israel incorporate the notion of *fair use* into their legislation, which is to say that limited use may be made of copyrighted material without explicit permission from rights holders under certain conditions. Certain common law jurisdictions have a similar notion, known as *fair dealing.* Such exceptions can be extremely important from the perspective of MIR research, since the task of acquiring permission from individual rights holders, especially from all three types rights holders, can be overwhelming if it must be performed for every recording in a dataset. Unfortunately, interpretations of how such laws apply to digital media can vary widely, and in many cases remain untested in the courts. The consequence of this is that many universities are reluctant to risk allowing the transfer of musical research datasets, even if in some cases they may arguably be legally permitted to do so.

There are also a number of legal details that can have a significant impact. For example, it is illegal under the DMA to bypass technological barriers (known as *Digital Rights Management,* or *DRM*) put in place to prevent the copying of music, even when it would fully be within one's legal rights to copy the music in question if DRM protection were absent. Such legislation can impose serious limitations on the sources from which the contents of research datasets can be collected.

Canada, where the datasets described in this chapter were assembled, has no such limitations at the time of this writing. On-line file sharing of music was actually explicitly legal in Canada for a brief period in 2004 and 2005, following a ruling in Federal Court on the case of BMG Canada Inc. v. John Doe.[220] This decision was set aside by the Federal Court of Appeal in May 2005, however, and the file sharing of music is now

[220] This decision applied specifically to music, and did not apply to other content, such as software or video files.

neither conclusively legal nor illegal in Canada. Some legal analysts interpret Canada's Copyright Act as indicating that it is legal to download music but not to upload it, provided that the downloaded content is not used in any for-profit ventures. This is a controversial interpretation that remains unproven in court, however. In any case, Canada's federal police force has stated that it does not currently intend to attempt to prosecute those sharing music or video for personal use (Deglise 2007). Canada's governments have tried twice since 2005 to introduce legislation that would explicitly prohibit the file sharing of music, but were unsuccessful both times due to procedural issues.

There are a number of general instances where music may be used and distributed legally by anyone without specific permission from rights holders and without any financial recompense. Such music is said to be in the *public domain*. Music is placed in the public domain if a rights holder chooses to waive his or her intellectual property rights, and music also usually enters the public domain following the expiration of its copyright. In the case of Canada, this happens fifty years after the death of the last rights holder. Caution must be exercised here, however, since this applies to both the performance and the composition. So, for example, even though Bach's 1741 composition *The Goldberg Variations* is in the public domain, Glenn Gould's 1981 recording of it is not.

There are also a number of frameworks that have been set up to facilitate the process of choosing to waive certain intellectual property rights but not others. This is helpful, for example, to individuals who wish to allow anyone to download their recordings for free, but still wish to retain licensing rights controlling the use of their music in advertising. Some of these are formulated with particular types of content in mind, such as the various GNU public licenses[221] that are oriented towards software, and others are more general, such as the Creative Commons licenses.[222]

[221] www.gnu.org/licenses/
[222] creativecommons.org

## 8.2 Fundamental issues

### 8.2.1 Overview of problems to overcome

As discussed in Section 8.1.1, individual MIR research labs have traditionally assembled their own musical datasets for training and testing algorithms, an understandable approach in the past given the lack of available alternatives at the time. This approach is ultimately flawed, however, since collections assembled in this manner tend to be limited in size and variety, and are often biased in ways relating to the characteristics of the particular system being developed and the musical backgrounds of the individuals assembling the dataset.

Furthermore, such datasets typically contain copyrighted material preventing them from being legally distributed to other research centers for refinement, expansion and comparison. Although such datasets have sometimes been "unofficially" shared amongst researchers in the past, this quasi-legal distribution is not viable in the long term. The ability to legally share datasets between MIR research labs is essential, not only for making it possible to comparatively evaluate different algorithms designed to perform a given task by testing them on the same data, but also for providing sufficient motivation and resources to develop large, varied, carefully chosen and well-labelled datasets.

Some of the most problematic aspects of dataset construction relate to the assignment of ground-truth labels to individual recordings, as discussed in Section 8.2.3. There are significant problems with metadata quality not only in MIR research datasets, but with respect to the music industry as a whole (Freed 2006). The labelling of subjective fields such as genre or mood is particularly problematic, as multiple interpretations may be equally valid. Such fields are nonetheless important when constructing a dataset, as many researchers and end users are interested in them. Furthermore, one must not only consider an appropriate methodology for labelling individual recordings, but also the choice of candidate classes and the relationships between these classes, as discussed in Section 8.2.2.

Even assuming that one does have a good ontology of labels and a reliable and consistent methodology for assigning these labels to the ground-truth recordings, the problem of choosing which particular recordings to include in a dataset remains. Ideally, one would like to have extensive examples of music of all types for which the algorithms at hand might reasonably be applied. This is necessary if one wishes to produce software that is flexible enough to deal with the real-world needs of end users who could potentially wish to apply the software to any music that one might find in the full universe of extant music.

In practice, however, there are a number of significant limitations that researchers must deal with. To begin with, research labs do not typically have access to anywhere approaching the entire universe of music of interest, for both financial and infrastructural reasons, including technological limitations that become apparent when dealing with huge collections of music. Not only are there very real limitations on how many recordings can be used to train and test a system because of the computational time that it takes to process each recording and the space or bandwidth needed to store or transfer them, but even the most recent algorithms are still often incapable of dealing with large class ontologies. For example, even as few as 38 genre classes can be too many for current genre classifiers to learn to reliably classify (McKay 2004).

Even if one were to be given access to all of the music ever recorded, the selection of the particular pieces to include in a dataset is certainly non-trivial. For example, even if one could reasonably limit oneself in a particular research project to Hip-Hop music, there are many styles of Hip-Hop, and many thousands of recordings belonging to each of these styles. Given the limitations discussed above on acquiring and processing huge datasets, one must devise some methodology for selecting a particular subset of all Hip-Hop recordings to include in the dataset. One must ideally include all styles of the music under consideration, which can be no easy mater. For example, if primarily American Hip-Hop is included in a dataset, this would mean that a system trained on it might not be able to properly process British Grime music. Furthermore, while one must certainly be sure that one's system is able to deal with prototypical examples of the types of music under consideration, at the same time the system must also be able to deal with outliers. This means that sufficient numbers of both prototypical and outlying exemplars must be included for all styles of music under consideration, while still conforming to limitations on the size of the dataset and the availability of particular pieces of music.

In the past, these issues could be glossed over to a certain degree, since many MIR research was more oriented towards achieving exploratory proofs of concept than building practically useful finished products fit for general use. Now that reasonably good results have been achieved in many MIR areas in ideal lab conditions, however, the importance of being able to deal with realistically diverse and extensive ranges of music is becoming increasingly apparent. Correspondingly, the construction of datasets that take the issues discussed above into account are becoming increasingly important, since high-quality realistic datasets will not only result in systems that perform better because they have been provided with a better basis for learning effective models during training, but will also result in more realistic evaluations of competing systems.

### 8.2.2 Formulating class ontologies

In many cases, some form of structure can reasonably be said to exist that delineates various kinds of organizational relationships between class labels. For example, in the case of mood classification, the *Gloomy* class could arguably be said to be a more specific sub-class of the *Sad* class*, Sad* and *Unhappy* are perhaps equivalent classes, *Ecstatic* might be said to be a stronger version of *Joyful,* and *Happy* is related to *Joyful,* but in some way that is not as hierarchical in nature as *Gloomy* and *Sad* or *Ecstatic* and *Joyful.* These examples illustrate the variety that can exist in the types of relationships between classes, and correspondingly in their ontological structuring. This variety certainly offers intriguing possibilities for study, but it also provides in indication of the many difficulties associated with building realistic class ontologies.

There have been a few preliminary studies exploring the possibility of using machine learning algorithms to form structured clusters of music in a representational space, usually related to research on similarity and/or visualizations (e.g., Lamere and Eck 2007). Ontological class structures could potentially be automatically derived from such approaches.

Unfortunately, there have been no serious attempts to date to use such clusters to form class ontologies that could then be used as a basis for organizing and labelling MIR datasets, probably because automatically generated class clusters often have little clear correspondence with the class labels that are actually used by humans beyond the coarsest level. Similarly, researchers in automatic music classification have for the most part avoided putting more than passing efforts into manually building realistic ontologies. As a result, the great majority of published MIR research has made use of either purely flat or very simple hierarchical class structures. There are only a few cases where more sophisticated ontologies have been used, such as Bodhidharma's genre ontology (Section 8.5.2), which is still little more than an extended taxonomy. This reliance on simple and unrealistic ontologies is probably due to the fact that manually constructing ontologies that are more than artificial constructions of convenience can be very difficult, as will soon be made apparent.

Nonetheless, this general failure to utilize realistic ontological class structuring is unfortunate, as such organizations can impart important advantages. From a purely practical perspective, non-flat ontologies can be taken advantage of by certain powerful technical classification methodologies, such as the hierarchical classification algorithm that was used by McKay (2004) to significantly improve genre classification performance compared to standard flat classification. Non-flat ontologies also make it possible to use weighted training strategies that penalize misclassifications to very dissimilar classes more harshly than misclassifications to more similar classes, an approach that can also ultimately improve the classification performance of final trained models.

From a more general perspective, sophisticated ontologies reflect the structuring that humans often informally use to associate classes with one another, and can thus be useful in musicological, music theoretical or psychological research. Furthermore, the process of forming such structures can force one to reconsider and improve the quality and consistency of the basic class labels that one is using, since constructing structures relating classes can quickly make it apparent that certain classes are missing or that certain classes that are being used should be split into sub-classes, for example.

This sub-section therefore discusses some of the key issues that should be considered when building musical class ontologies. When appropriate, possible solutions are proposed or reviewed. The particular case of genre ontologies is used in this discussion as a case study, since genre can be one of the most problematic types of ontologies and thus brings to light many important issues. Genre labels are widely used and understood by music consumers, yet most classes remain poorly defined. Many of the issues raised with respect to genre are equally applicable to other types of class ontologies, such as in tasks like mood classification or even artist identification (e.g., Thom Yorke could arguable be said to be a sub-class of Radiohead).

The lack of a commonly accepted set of clearly defined genre labels makes it tempting to simply devise one's own artificial ontology consisting of genre labels chosen out of convenience based on the music that one is the most familiar with and has the easiest access to. For the sake of simplicity, one might also be drawn towards designing an ontology where the genre classes are well-defined, independent and consistent, with a simple and logical structure based on obvious class similarities. A further temptation is to omit some of the more problematic genres from the ontology, such as Worldbeat or Indie. The genre labels in common use are often inconsistent, illogical and seemingly haphazard, and someone designing an automatic classifier would certainly ideally like to have an ontology that does not suffer from these problems.

Giving into such temptations would, of course, be a mistake, as any attempt to artificially impose an unrealistic genre ontology on the public is doomed to failure. The Canadian Content radio genre categories used by the Canadian government are an example of such a failure (Frith 1996).

One must instead use class labels and relationships between them that are meaningful to real people who listen to the real universe of music if one wishes to design an ontology that is actually meaningful and useful to them. So, genre class ontologies used in MIR research must at least be consistent with those that individuals with moderate musical knowledge might use to perform their own categorizations.

An additional factor to consider is that genre labels are constantly being created, forgotten and modified by musicians, retailers, music industry public relations departments, DJs, critics and audiences as musics develop. A static, ideal system is therefore ultimately not sustainable. Genres are not defined using strictly objective and unchanging qualities, but are rather the product of dynamic cultural processes. One must therefore be careful to avoid thinking of genres in terms of immutable snapshots, as both their membership and their definitions change with time.

For example, the genre labels attached to a particular recording can change, even though the audio in recording itself, of course, remains static. The Bob Marley recordings that we recognise as *Reggae* today were once classified as *Folk Music* in most Canadian record stores, for example, and the changes in what has been considered to be *Rock* in each decade from the 1950's to today provides further illustration of this. In order for a genre ontology to have true practical utility, it must therefore be possible to bring it up to date as it changes.

Genre ontologies should ideally still include historical music, since such music is still listened to, or could be listened to in the future, and it should not be forgotten that even historical genres can change with time. Of course, some might argue that historical genres tend to be at least somewhat more static and codified than current genres, with the results that they easier to describe and their membership is fairly set. There is, for example, a large amount of literature on Baroque music theory and practice, and there is not any significant quantity of new Baroque-style pieces being composed that might cause this genre to mutate. Although there is of course some truth to this, historical genres can nonetheless still evolve to a significant degree, as is demonstrated by a comparison of the relatively Romantic recordings of Baroque music from the early 20th century with modern Baroque recordings that emphasize period instruments and stylings.

*Syncretic music*, which is to say music that combines characteristics of multiple genres, introduces a further set of issues to consider. Although syncretic music can often lead to the creation of new sub-genres, there is generally at least a transitional stage where such music does not definitively fit into any of its parent genres, but does not yet have a genre of its own.

So, how does one approach the design of an effective and realistic genre ontology? One approach would be to look at the genre labels used by the music industry, such as in music sales charts published in *Billboard* or awards shows like the Grammies. Unfortunately, there are a number of problems with this approach. Such ontologies often only reflect the current trends in music to the exclusion of older or lesser known genres, and the categories that are used tend to reflect the labelling system that the music industry would ideally like to see used for commercial reasons, rather than the type of ontologies actually used by the public. Negus (1999) offers an intriguing analysis of the effects of business interests on musical genres and their development.

Specialty shows on radio, television and the Internet offer a somewhat better source of genre labels, as they have a greater tendency to reflect realistic classes in order to attract real listeners who are interested in particular genres. They do still often suffer from the influence of commercial biases, however, and their content tends to be influenced as much by the interests of advertisers as by the musical preferences of listeners. Although university radio stations do not suffer from this problem in the same way, they are often limited in scope and by the variable expertise and knowledge of their DJs.

Music retailers, particularly Internet retailers, such as the Apple iTunes Store[223] or AmazonMP3,[224] may perhaps be some of the best sources of genre labels. They use genre classes that are likely the closest to those used by most people, as their main goal is to use an ontology that makes it easy for customers to navigate to music that they are looking for. Retailers also tend to respond relatively quickly to changes in genre usage, as adding new genres and keeping existing genres up to date allows them to draw customers into focused sales areas that contain other music that customers may wish to buy, thereby increasing sales.

Although one might argue that it would be preferable to base genre labels on the views of concert goers, clubbers, musicians, DJs, music reporters and others who are at the front lines of genre development, doing so can have the disadvantage that genres at this stage of development may be unstable. Additionally, favouring the genre labels used by specialists may result in confusion for non-specialists. Waiting for retailers to recognize a genre and thus make it “official” is perhaps a good compromise in that one can hope to keep relatively abreast of new developments while at the same time avoiding excessive ontological specialization and associated overhead in terms of data collection and training.

The problem of inconsistency remains, unfortunately, even with the ontologies used by retailers. Employees of different record companies, distributors and retailers may not only classify a given recording differently, but may also make selections from entirely different sets of candidate genre labels or emphasize different identifying features. This is, unfortunately, an unavoidable problem, as there are no widely accepted labelling standards or classification criteria, and individual consumers also use varying genre ontologies and instance labelling.

So, in the end, one has little choice but to adopt one or several of the imperfect labelling systems that are in use. This can be done either manually or by automatically

[223] www.apple.com/itunes/store/
[224] www.amazon.com

mining information from the Internet. The former approach can certainly be effective, as long as one is careful to avoid personal biases and the temptations discussed earlier, but the latter option has the significant advantage of making it possible to constantly update ontologies automatically as genres change, with relatively little human overhead.

Aucouturier and Pachet (2003) provide an excellent theoretical discussion on the relative advantages of these two approaches to constructing ontologies, and ultimately argue strongly in favour of a variant of the automatic approach that they call the *emergent* approach. Specifically, they suggest using similarity measurements based on audio content as well as on cultural information gleaned from applying data mining techniques to text documents. They also propose the use of collaborative filtering to search for similarities in the taste profiles of different individuals as well as the application of co-occurrence analysis to the play lists of radio programs and the track listings of CD compilation albums.

Unfortunately, this approach remains untested, and would be very difficult to implement. Furthermore, there is no guarantee that the recordings that get clustered together would be consistent with groupings that humans use in reality or that humans would find convenient to use. There is also no obvious provision for defining the types of ontological structuring that humans find useful when browsing through categories. Nonetheless, the emergent approach certainly does warrant further investigation.

It also possible to automatically harvest existing ontologies found on the Internet in ways that retain existing structure. There are two general philosophical approaches to doing this. The first is to selectively harvest data from particular sources where the ontologies are known to have been constructed by individuals with at least some degree of musical expertise, whether this construction was performed in a closed framework (e.g., the All Music[225] genres) or is the result of collaborative editing (e.g., Wikipedia genre classes[226]). The second approach is to harvest information from the wisdom of the crowds in general, which can be done either by mining the Internet as a whole or by extracting data from particular sources where any user may tag music (e.g., Last.fm's ontology of tags[227]).

[225] www.allmusic.com
[226] en.wikipedia.org/wiki/Category:Music_genres
[227] www.last.fm/charts/toptags

Of course, even if one chooses to base one’s ontology on information that is mined automatically from the Internet, one must still ultimately make the final decision as to whether or not an ontology is appropriate for use, and it may be necessary to perform some degree of manual tweaking or correction. In order to do this properly, one must consider the particular kind of perspective that one would like the ontology to provide on music at a fundamental level, as well as the dimensions of organization that are most appropriate for the ontology.

Pachet and Cazaly (2000) provide an interesting discussion of related issues. The authors observe that brick and mortar retailers tend to use a four-level hierarchy for organizing their music. From broadest to most specific, these are: global genres (e.g., Classical, Jazz, Rock), sub-genres (e.g., Opera, Dixieland, Heavy Metal), artist names and album titles. Although this ontology is effective when navigating a physical record store, the authors argue that it is inappropriate from the viewpoint of establishing a major musical database, since the different levels are organized using different dimensions. In other words, a genre like Classical is fundamentally different from the name of an artist.

Pachet and Cazaly continue on to note that Internet companies, such as Amazon, tend to build tree-like ontologies, with broad categories near the root level and specialized categories at the leaves. The authors suggest that, although this is not in itself necessarily a bad approach, there are some problems with it. For instance, the level that a category appears at in the hierarchy can vary from taxonomy to taxonomy. Reggae, for example, is sometimes treated as root-level genre, but at other times is considered to be a sub-genre of World Music.

A further problem is that there is a lack of consistency in the types of hierarchical relationships between a parent and its children. For example, sometimes this relationship is genealogical (e.g., Rock is a parent of Hard Rock), sometimes it is geographical (e.g., Africa is a parent of Algeria) and sometimes it is at least partially based on historical periods (e.g., Baroque is a parent of Baroque Opera). Although these inconsistencies are not necessarily significant obstacles for people manually browsing through catalogues, they could potentially be problematic for automatic classification systems or data miners.

To provide another example, many users tend to organize the Popular music in their digital collections based on performer names, with relatively little attention provided to composer or instrumentation tags. With Classical music, however, the composers' names are often judged to be more important from an organizational perspective than the performers' names. Classical music is also often organized based on instrumentation (e.g., symphony, string quartet, opera, etc.), whereas one rarely sees Popular music organized this way (e.g. power trio, ska quintet, rap crew, etc.)

Pachet and Cazaly ultimately suggest the use of a tree-based taxonomy organized based on genealogical relationships, where only leaves would contain musical examples. They further propose that each node should indicate its parent genre and the differences between its own genre and that of its parent.

A similar approach is implicitly proposed by Cumming as well, who has adapted Ludwig Wittgenstein's general ideas about the *family resemblance* between genres specifically to music (Cumming 1999). She uses this theoretical basis as justification for favouring an exclusively genealogical organization of classes, much like Pachet and Cazaly. She argues that, since lists of simple and well-defined binary features are insufficient to distinguish between sometimes amorphous genres, it would be wise to consider genres in terms of the similarities that they share with the features of genre families that they have descended from.

Although a hierarchical and exclusively genealogical approach does in some ways make the best of a bad situation, ultimately caution should be exerted before exclusively conforming to it. Doing so ultimately forces one to sacrifice the flexibility that is both the greatest strength and the most problematic aspect of ontologies, and also limits the extent to which one can model the way that people actually construct genre structures.

It seems apparent that some modifications are needed to Pachet and Cazaly's approach if a more accurate model of real ontologies is to be achieved. For example, an extended version of a hierarchal tree-based taxonomy certainly does have appeal and, if designed cleverly, could encompass more sophisticated and realistic genre structures. It is for this reason that such an extended taxonomy is used in the Bodhidharma MIDI dataset, as described in Section 8.5.2.

There are a number of additional arguments supporting the incorporation of some level of hierarchy in genre ontologies. For example, Fabbri (1982) suggests that most individuals, when faced with the prospect of describing a genre to a person who is unfamiliar with it, will do so by defining the genre as an intersection of other similar genres with labels known to both parties, by using a broader label under which the genre in question might fall, or by explaining the genre using familiar terms such as definitions and emotive meanings. The former two methodologies are certainly consistent with a hierarchal structure where a genre's parents and siblings are visible.

A related issue to consider is the variable degree to which different genres branch out into sub-genres. Considered from a tree-based perspective, this variability applies to both the depth and breadth of various branches. Some genres have many very specialized sub-genres, such as Electronica (e.g., House, Ambient, Jungle, etc.). Others, such as Pop-Rock, tend to have fewer, broader and less well-specified sub-genres. For the purposes of creating a genre hierarchy, one must be careful to accept these inconsistencies rather than imposing unrealistically broad or narrow categories in order to avoid messy dissymmetry in the genre structure.

An additional issue of some controversy is whether ontologies should be designed in such a way as to accommodate the classification of artists by genre or the classification of individual recordings by genre. Aucouturier and Pachet (2003), for example, suggest that one should in fact use ontologies oriented towards classifying artists as a whole rather than individual recordings, as the classification of individual recordings would involve too many instances to practically classify and would result in categories that are overly narrow and have contrived boundaries.

The problem with this approach, however, is that many musicians, such as Neil Young and Miles Davis, for example, write and perform music in entirely different genres throughout their careers. Such genre variations for a given artist can even occur on the same album or, in rarer cases, in different sections of the same piece. It seems clear that choosing to classify musicians rather than individual recordings is problematic in some cases.

Such problems can be partially addressed by allowing multiples classes to be assigned to the same instance (e.g., allowing Miles Davis to be assigned classes of Bebop, Cool Jazz and Jazz Fusion all at once). Practically speaking, however, this still leaves certain problems unresolved if one is attempting to train an automatic music classifier to classify artists as a whole by genre. For example, training a learning algorithm on a particular

Miles Davis piece that happens to be purely Cool Jazz will confuse the classifier if the only available ground-truth is a broad Miles Davis genre label that indicates that it is Jazz Fusion as well as Cool Jazz, two styles that are very different. Nonetheless, at least the errors that would result from such situations would not be as bad as those that would result if only one label were permitted per artist.

Ultimately, the separate classification of each recording seems to be the wiser path to take, despite the additional work required to assign model labels to all of the recordings in the ground-truth individually before training. Even if one is only interested only in broad artist classifications, these can easily be extrapolated from the aggregate of the labels assigned to the individual recordings associated with each artist

Even if one chooses to take this approach of classifying individual recordings rather than artists as a whole, one should still allow multiple class labels per recording. This is necessary in order to take into account the reality that there can be significant overlap between different genres, and only allowing one genre label per recording could essentially involve arbitrarily choosing one genre label over others. Allowing multiple labels per instance therefore allows more accurate simulations of the amorphous boundaries between genre classes and of the varying levels of similarity between them.

Many of the problems discussed throughout this sub-section lack ideal solutions, partly because of the nature of the problem: the inherently incompatible goals of, on the one hand, forming well-organized and reasonable ontologies while, on the other hand, attempting to accurately simulate the often disorganized and illogical ontologies that one encounters in the real world. Ultimately, the building of ontologies will likely remain as much of an art as it is a science, at least until improved and more easily accessible data mining techniques make emergent approaches more viable. It should certainly be noted that, at least until this can be done, those building ontologies should incorporate an understanding of the psychology of human classification and categorical organization, as this is at the root of what they are attempting to model. Lakoff's book (1987), for example, is an often cited source on this topic.

One general (albeit difficult to implement) solution to many of the issues described in this sub-section would be to develop a super-ontology such that multiple parallel and potentially independent (or dependent, as appropriate) ontological class structures could co-exist side by side. Each sub-ontology in such a framework could incorporate those potentially differing class labels and relationships between them that are meaningful to different types of listeners, potentially based on dimensions such as time, musical expertise, cultural background, etc.

### 8.2.3 Labelling individual instances

Even given an appropriate ontology from which to draw class labels, the assignment of good ground-truth labels to individual recordings can be an area that is particularly problematic in some cases. This is a significant issue, considering the essential role that these ground-truth labels play in training and testing.

The easiest way to acquire ground-truth labels is to simply make use of pre-existing label annotations. These can be extracted from the metadata packaged with recording files, such as the ID3 tags associated with MP3 recordings, or they can be automatically downloaded from existing metadata databases, such as the Gracenote CD Database[228] or fingerprinting services such as MusicBrainz.[229]

Unfortunately, the metadata that one typically acquires from such sources is often at best noisy and inconsistent, and at worst entirely incorrect. Such sources typically make use of metadata collaboratively entered by many most often unqualified individuals, with the result that at least some of the annotators are likely to have made errors, and different annotators, even when well-qualified, may use incompatible labelling methodologies.

Although manual metadata proofreading by individual MIR research labs after preliminary labelling using such sources is certainly an option, this can be extremely time consuming, and even the most conscientious proofreaders can miss errors when dealing with huge collections of music. Furthermore, even if a lab decides to do all or most of the metadata annotation themselves, a number of problems remain. Both significant amounts of time and extensive musicological knowledge are typically needed to properly label recordings, both of which can be lacking in labs that are performing technical MIR research where dataset collection is only a small part of a larger project. For example, carefully labelling the note onsets of even a relatively simple monophonic piece by hand with any reasonable degree of precision will easily take at several times the duration of

[228] www.gracenote.com
[229] musicbrainz.org

the piece. In addition, even well-trained annotators who spend appropriate amounts of time on each recording can still make somewhat different annotations in some cases. This can be very problematic if different algorithms are to be evaluated and compared based solely on how well they agree with such ground-truth annotations, which may not agree with each other, and may even contain annotations that are simply incorrect.

Various techniques for automatically mining data from the Internet can be helpful for labelling ground-truth. Such automatic labelling has the important advantages of saving the time and resources needed to manually label recordings, of taking advantage of the collective wisdom of the crowds and of using information that by definition largely consists of the types of annotations that consumers find to be useful. However, simplistically or naively mining information such as user assigned tags from the Internet can result in labels that are even noisier than those found in ID3 tags, for example.

Fortunately, more sophisticated techniques can result in much better labels. The work of Hu, Bay and Downie (2007) is a good example of some very practical efforts in this direction. They harvested mood labels from Last.fm[230] user tags for use as ground-truth in the MIREX 2007 mood classification evaluation, utilizing strategies for dealing with noisy raw tagging data such as text pre-processing, clustering and principal component analysis, with some good results. There is still, however, certainly still a need for refinement and improvements in methods for cleaning such raw data. Interestingly, Geleignse, Schedl and Knees have found experimentally that there is a correlation between the Last.fm tags annotated by users and general artist similarity (2007).

Another approach for acquiring collaboratively generated ground-truth labels is to take user surveys, as Ellis and his colleagues have done with respect to similarity judgements (Ellis et al. 2002). A particularly effective way of doing this can be to create web labelling games to motivate participants not only to think carefully about the labels that they are assigning, but also to encourage them to attempt to label in ways that they feel others will also label rather than simply labelling in ways that might only be meaningful to themselves. The *ESP Game* for labelling images (von Ahn and Dabbish 2004) was a ground-breaking move in this direction, and similar but expanded approaches

[230] www.last.fm

have been adapted to music in games such as *MajorMinor* (Mandel and Ellis 2007) and *ListenGame* (Turnbull et al. 2007).

Manually annotating recordings of course still remains as an option, assuming the annotators have sufficient time and expertise to perform the task properly. This can be done exclusively by hand or can be done after the harvesting of preliminary labels using one of the techniques described above. Ideally, manual labelling should be done by a committee rather than entirely by a single individual, and the committee members should have a wide and diverse perspective on music. Not only does this help to at least minimize individual bias, it also makes it possible for the committee to discuss particularly problematic instances when needed.

Jones, Downie and Ehmann (2007) have performed an excellent statistical study of categorical judgement data generated by human evaluators with respect to judgement stability, inter-grader reliability and patterns of disagreement. This information was acquired during data collection for MIREX using the *Evalutron 6000* web-based tool for capturing human similarity judgments. Although there was some significant disagreement between individual raters, results were still generally good, and the authors present several good recommendations for use in future data collection.

### 8.2.4 Guidelines for building successful datasets

The previous sections of this chapter have discussed in some detail many of the problematic issues associated with the design and construction of MIR research datasets, as well as emphasized the importance of high-quality datasets. Unfortunately, the construction of a realistically large and varied ideal dataset with excellent ontological organization and reliable ground-truth labelling is beyond the resources of most MIR research labs, with the possible exception of a few commercial labs that have access to the necessary manpower and musical catalogues.

There are, however, certain general guidelines that can help to improve the quality of datasets collected by even those research labs with very limited resources. The following list of proposed guidelines for building research datasets combines summaries of some the essential ideas discussed in previous sub-sections with a number of new suggestions:

- Data should be freely and legally distributable to researchers. This includes both metadata and, ideally, sufficient information about the music itself for other researchers to use it in their own work. This could be in the form of actual samples when legally permissible, or researchers could be provided with the opportunity to remotely extract custom features.
- Information on entire recordings should be accessible, not just short excerpts. Each researcher should be able to choose how much and what parts of recordings they wish to utilize.
- Given that different audio compression methods can influence extracted feature values (see Section 3.2.5), the audio format(s) most commonly used by the public should be adopted in order to reflect realistic conditions. This is important for many types of end user oriented research, although uncompressed audio is also useful for some theoretical research. A variety of encoders and bit rates should be used for similar reasons.
- The dataset should contain many different types of music. It should not only include music belonging to many different genres, but also a sufficient number of examples illustrating as full a range of styles within each genre as possible.
- Examples from as many different performers and composers as possible should be included for each style of each genre of music in the dataset, and both prototypical examples of each type of music and stylistic outliers should be included as well.
- The dataset should include a significant amount of commercial music, although independent music should certainly be included as well. The vast majority of the public is interested primarily in professionally produced music, so MIR systems must demonstrate that they are able to deal with such music.
- The dataset should include many thousands of recordings. This is important not only in order to allow sufficient variety, but also to avoid research overuse of a relatively small number of recordings, which could result in overtraining. Furthermore, even good-quality metadata annotations will inevitably contain some errors, and a large dataset helps to average out such noise in the ground-truth.

- Each recording should be annotated with as diverse a range of metadata fields as possible in order to make the dataset usable as ground-truth for as wide a range of MIR research areas as possible.
- It should be possible to assign multiple independent labels to a single field so that, for example, a recording could be classified as belonging to both the *Swing* and *Blues* genres, or a Beatles song classified as being composed by both *Paul McCartney* and *John Lennon*, rather than one non-existent songwriter named *Paul McCartney and John Lennon*.
- It should ideally be possible to label segments of recordings as well as recordings as a whole.
- Annotations of subjective fields such as genre or mood should include a wide range of candidate class labels, since only allowing an artificially small number of classes is unrealistic.
- Candidate class labels should be organized with appropriate ontological structuring when appropriate. Such structures should reflect the types of ontologies that are actually used by the public, not ontologies that are chosen based on the convenience of those constructing the dataset. Also, these structures should be easy to modify as cultural conditions change. Ontologies can be constructed using either manual or automatic techniques, and in many cases the ideal situation may be to have multiple ontologies co-existing with each other with respect to the same data.
- Whenever possible, multiple sources should be consulted for the labelling of each recording. Labelling can be performed by panels of experts, metadata repositories available on-line and surveys of the public, ideally using a game-oriented interface to improve results, as discussed in Section 8.2.3. Whichever approach is used, a standardized methodology should be adopted so that the annotations of individual recordings are as correct, complete and as consistent as possible.
- Metadata should be made available to users in formats that are easy to both manually browse and automatically parse.

- Automatic tools should be available to error-check metadata and generate profiles of datasets.
- It should be easy to add new recordings and their metadata to the dataset.

This chapter includes presentations of three sample datasets, namely the Bodhidharma MIDI dataset (Section 8.5), Codaich (Section 8.6) and SAC (Section 8.7). Each of these datasets illustrate several ways in which at least some of the guidelines outlined above can be followed in practice, and insights are provided on the solutions that were used to address specific problems that were encountered during dataset design, collection and annotation.

## 8.3 Existing music information retrieval research datasets

As discussed in the preceding sections, individual MIR research labs have traditionally assembled their own simple musical datasets for training and testing their algorithms. However, as also described in detail above, there are a number of important drawbacks to this approach, and there is a strong need for large high-quality datasets that can be used by a variety of labs to compare the effectiveness of different algorithms. This section reviews existing datasets that were designed with the specific goal of being used by MIR researchers at a variety of research institutions or that have been widely used by many researchers.

The *USPOP2002* dataset (Berenzweig et al. 2004) is a dataset that was very often used in early MIR research. It is a collection of 8764 MP3 recordings of popular music from the U.S. at a bitrate of 128 kbps. This music was extracted from 706 albums by 400 artists. The metadata includes artist and album labels as well as *style* tags harvested at the time from the All Music Guide. Although it was an extremely valuable contribution at the time, from a contemporary point of view it suffers from a number of problems, including limited diversity and, most significantly, the lack of a legal way to share it between research labs.

A more recent but comparable dataset is the *Latin Music Database* (Silla, Kaestner and Koerich 2007), which consists of 3160 MP3 recordings belonging to ten different genres of Latin dance music. Although this is a well-annotated and high-quality dataset, it is, of course, limited to specific styles of music, and the legality of its distribution is questionable.

One approach to building datasets that are legally distributed is to make use of recordings that are in the public domain, such as the recordings found on sites like GarageBand[231] and Jamendo.[232] Homburg and his colleagues (2005) have explored such an approach, and have also incorporated multiple views and high-quality annotations. Unfortunately, this approach generally restricts one to a limited number of recordings that are usually either very old, are by amateur musicians or consist of only short low-quality extracts. These limitations are demonstrated by Homburg et al.'s database, which, despite its many advantages, contains only 1886 recordings, each of which is only a 10-second extract.

Some improvement can be achieved by utilizing music protected under more limiting but still relatively lenient legal frameworks, such as those offered under the Creative Commons. Web merchants such as Magnatune[233] and Epitonic,[234] for example, allow the public to preview recordings for free, and entire music collections can sometimes be licensed for research purposes at little or no cost. Unfortunately, size can still be an issue, as such merchants tend to have very small catalogues compared to commercial distributors. Also, the music available on such sites usually excludes those artists that are contracted to major record labels and are the most visible to the public eye, which is the music that most listeners are interested in. In addition, the metadata contained in the ID3 tags of music sold by such sites is generally not entirely reliable, with problems such as multiple spellings of genre names and strange treatment of special characters, problems that one would expect to be absent given that each of these sites has complete control over the metadata encoded in their catalogues. Finally, there can still be legal restrictions on distributing music downloaded from such sites to other researchers, even if one is permitted to access and store entire recordings for free oneself.

[231] www.garageband.com
[232] www.jamendo.com
[233] magnatune.com
[234] www.epitonic.com

Of particular note, Edith Law, Olivier Gillet and John Buckman have produced the Magnatagatune dataset.[235] This dataset consists of Magnatune audio recordings combined with human annotations resulting from the TagATune game (Law et al. 2007) and audio analyses produced by Echo Nest[236] code.

Another option is to contract arrangers and musicians to produce original recordings for use in research, as was done in the construction of the RWC database (Goto et al. 2002; Goto 2006). This approach has the important advantages of overcoming copyright limitations, since the copyrights are owned by the responsible research institution, and of generally including high-quality metadata. Unfortunately, costs prevent such datasets from scaling to any reasonably large number of recordings. There can also be doubts as to how well such original music simulates what one encounters in the real world.

There are, of course, many other datasets that have been used in individual research projects or that are available for sale in contexts not specifically oriented towards MIR. Donald Byrd is maintaining a partial listing of many such collections at www.informatics.indiana.edu/donbyrd/MusicTestCollections.HTML.

## *8.4 jMusicMetaManager: A tool for profiling musical datasets and detecting metadata errors in them*

### 8.4.1 Overview of jMusicMetaManager

jMusicMetaManager (McKay, McEnnis and Fujinaga. 2006) is an open-source software package that is designed to profile and manage large collections of music. It emphasises functionality for detecting metadata errors and inconsistencies and for generating a variety of reports that analyze, statistically describe and summarize the contents of music collections.

As discussed in Section 8.2, the metadata available from sources such as the ID3 tags of MP3 recordings or the Gracenote CD Database is often unreliable. This is an important issue if one wishes to apply machine learning to collections of music that are too large to conveniently and effectively correct by hand, since the consistency and correctness of ground-truth can play an essential role in the ultimate efficacy of any learned model.

[235] tagatune.org/Magnatagatune.html
[236] echonest.com

jMusicMetaManager was therefore developed with a particular emphasis on detecting errors in the metadata that is used to train and test learning algorithms. As such, jMusicMetaManager has been used to manage the Codaich music research database (see Section 8.6). It is important to note, however, that jMusicMetaManager is in no way limited to MIR projects, and can be a useful tool for users ranging from individuals who simply want to organize their own personal music collections to employees of libraries or other large institutions wishing to profile their music databases.

Users need a convenient interface for viewing and editing the metadata associated with individual recordings in music collections. Although jMusicMetaManager does include a GUI (described in Section 8.4.5) that provides users with a large degree of control over which reports are generated and what parameters are used during error detection, it does not provide functionality for editing metadata. It was decided to instead take advantage of existing software by making jMusicMetaManager compatible with the Apple iTunes software, which is not only free, well-designed, and commonly used for editing metadata, but also includes a relatively easily parsed XML-based file format. iTunes has the important advantage of saving metadata directly to the ID3 tags of MP3s as well as to its own files, which means that the recordings can easily be disassociated from iTunes if needed. iTunes can also automatically access Gracenote's metadata.

jMusicMetaManager can therefore extract metadata from iTunes XML files as well as directly from MP3 ID3 tags.[237] Since MIR systems do not typically read these formats, jMusicMetaManager can also be used to generate ground-truth data by transferring metadata stored in these formats to ACE XML or Weka ARFF (Witten and Frank 2005) files.

Once jMusicMetaManager has extracted metadata from iTunes XML files or ID3 tags, it then checks this metadata for probable errors. The error detection algorithms focus on detecting differing metadata values for the same field that should in fact be the same (e.g., entries of *Stravinski* and *Stravinsky* in the artist field) and on detecting redundant duplicates of the same recordings. Many different approaches to finding errors are used,

[237] In order to overcome the iTunes limitation of only being able to store one genre per recording, jMusicMetaManager parses genres that are separated by " + " as belonging to multiple genres. For example, jMusicMetaManager would interpret a genre label of *Blues Rock* as the single genre of Blues Rock, and a label of *Blues + Rock* as the two different genres of Blues and Rock.

as described in Section 8.4.2, since no one approach is likely to detect all errors. Section 8.4.4 provides further details on the specific algorithms that are used.

jMusicMetaManager can generate 42 different kinds of reports in HTML format. These describe both probable metadata errors and statistical profiling information about music collections. This latter set of reports includes, among other things, multiple data summary views and breakdowns of co-occurrences between recording titles, artist names, composer names, album titles and genres. Such reports allow one to easily publish music collection profiles on-line and review the contents of collections from a variety of perspectives. Section 8.4.3 describes each of the different types of reports that jMusicMetaManager can generate

Like all jMIR components, jMusicMetaManager is written in Java in order to maximize portability. It is also designed to be easily extensible so that additional error-checking functionality, types of reports, metadata input sources and other types of functionality can be added to the software.

The jMusicMetaManager compiled Java bytecode, the associated source files, the jMusicMetaManager manual and other documentation are freely available at jmir.sourceforge.net/index_jMusicMetaManager.html. jMusicManager makes use of two freely distributable third-party Java libraries, namely the Apache Xerces[238] XML parser and the de.vdheide.mp3[239] package for parsing ID3 tags from MP3 files.

### 8.4.2 Error-checking operations

One of the primary functions of jMusicMetaManager is to detect different metadata values for the same field (e.g., artist name) that should in fact be the same. Such differences could be due either to spelling mistakes, as in the case of misspellings of *Lynyrd Skynyrd,* for example, or to multiple valid spellings, such as in the name of certain Russian composers.

Failure to detect metadata inconsistencies can be highly problematic from a machine learning perspective. For example, a system designed to classify music by composer that is trained on ground-truth where some pieces are marked as being by *Tchaikovsky* and others are marked as being by *Tchaikovski* will erroneously treat the two as entirely

[238] xerces.apache.org
[239] www.vdheide.de/java_mp3/

different classes. Aside from simple incorrectness, such an increase in the overall number of classes will likely have the consequence of increasing the difficulty of learning an effective model, and could thus lower classification performance overall. Such problems are compounded if a large ground-truth training dataset is used that contains multiple labelling inconsistencies. An additional issue is that differently labelled duplicate copies of the same recording could contaminate evaluation results by being placed in both training and testing subsets.

Even outside of the realm of machine learning, such labelling inconsistencies can be inconvenient for general usage of music collections, such as in cases of searches that only return a subset of the actual recordings in the collection that correspond to a particular metadata label.

The following four sub-sections of this section describe in some detail the four different overall types of error detection processing that are used by jMusicMetaManager. Further details on how these different types of processing are implemented and combined are provided in Section 8.4.4.

#### 8.4.2.1 Edit distance thresholding

The e*dit distance* (Levenshtein 1966) between two strings, also known as *Levenshtein distance,* can be a convenient tool for detecting the types of metadata inconsistencies described above. Edit distance is a measure of the similarity between two strings that is often used in information theory and computer science. It is defined as the minimum number of operations needed to transform one given string into another given string. Each of the following is considered one operation:

- Deletion of a single character in one string
- Insertion of a single character in one string
- Replacement of a single character in one string

Edit distance provides one of the core metrics used in jMusicMetaManager's error detection functionality (after significant pre-processing, as described in Section 8.4.2.2). In essence, the software calculates the edit distance between each pair of entries for a given metadata field. So, for example, the edit distance is calculated for all possible pairs of unique strings in the artist field. The resulting edit distances are then compared to thresholds in order to determine whether two entries are likely to in fact correspond to the same true value. This is done separately once each for the title, artist, composer, album and genre fields. The three types of edit distance thresholds that jMusicMetaManager can apply are as follows:

- **Absolute threshold:** A fixed number of edit operations that is independent of the nature of the particular pair of strings being compared. A pair of strings are flagged as being probable different spellings of the same thing if their edit distance is below or equal to this threshold.
- **Proportional threshold:** This threshold is varied dynamically based on the length of the longer of the two strings being compared. The threshold will thus be higher for longer strings than for shorter strings, because the probable number of erroneous characters increases with the length of a string.
- **Subset threshold:** This threshold is used to account for cases where one string might be a subset of another. For example, the artist names *Stevie Ray Vaughan* and *Steve Vaughan* likely refer to the same person, but there is a relatively high edit distance of 5 between them. If, instead, *Steve Vaughan* and *Stevie Vaughan* are compared, then the edit distance is reduced to one. *Steve Vaughan* is effectively a modified subset of *Stevie Ray Vaughan,* without the *Ray* and missing the *i* in *Stevie*. Subsets such as these tend to appear often in musical datasets, such as in cases where long recording titles have had their endings cut off by the limitations on field length in early ID3 tags. The availability of an edit distance threshold that takes this into account is therefore important. The subset threshold used by jMusicMetaManager subtracts the difference in length between the pair of strings from the edit distance measurement that is used in the threshold comparison. The threshold is also weighted by the length of the shorter string.

The user may choose which of these thresholds are applied, as well as each of their numerical values. These choices will, in practice, depend on the musical dataset under consideration and on the contrasting balance of priorities between false negatives and false positives. Edit distances are calculated after find/replace operations (see Section 8.4.2.2) and after reordered word subset operations (see Section 8.4.2.3). This ordering was chosen in order to reduce the edit distances of field entries that have specific types of erroneous differences that have commonly been observed in field values that are intended to refer to the same entity. The pre-processing operations applied prior to the measurement of edit distances thereby reduce certain edit distances in a targeted way that will increase the probability of the detection of errors using edit distance metrics.

For example, *REM* and *R.E.M.* have an edit distance of three, a value that is too high to fall within typical error detection thresholds, particularly considering that the length of the longer string is only six characters long. Pre-processing that removes periods would reduce the edit distance to 0, and thus result in a detected correspondence using even the most selective thresholds.

#### 8.4.2.2 Find/replace transformations

Although edit distance calculations can be very effective at detecting certain kinds of spelling inconsistencies, there are certain kinds of differences that will typically be missed if appropriate pre-processing is not applied, such as in the case of the *R.E.M.* example above. jMusicMetaManager can therefore apply a range of pre-processing transformations to all strings for which edit distances will be compared, with the intent of removing certain selectively chosen differences between strings and thereby reducing edit distances and helping to prevent appropriate inconsistent spellings from being missed. These pre-processing transformations essentially consist of searching all strings under consideration and modifying them in ways that are likely to reduce the number of false negatives while not increasing the number of false positives

For example it can be common for the titles of certain types of music to sometimes end words with *in'* instead of *ing*. This is something that can increase edit distances by a value of one. jMusicMetaManager can therefore replace all occurrences of *in'* with *ing* before edit distances are calculated. Note that these types of modifications are only performed internally by jMusicMetaManager, and the actual metadata itself is not altered.

Since the particular choice of modifications that are likely to be appropriate can depend on the particular musical datasets under consideration, the user is able to choose which to apply to the title, artist, composer, album and genre fields. The options are as follows:

- **Treat upper/lower case as identical:** Whether or not all upper case letters are converted to lower case letters (e.g., *Set the Controls for the Heart of the Sun* should match *set the controls for the heart of the sun*). Since some metadata annotators typically omit capital letters from their labels, this option reduces the edit distance between such annotations and annotations by labellers who observe the proper use of case.

- **Remove numbers and spaces at beginning of titles:** Whether or not all spaces and/or numbers before the first non-space and non-number character of a string are removed (e.g., *Sour Times* should match *02 Sour Times*). This is done because some annotators commonly include track numbers at the beginnings of track titles while others do not, and because spaces at the beginnings of titles are a common error that can be difficult to detect visually.

- **Convert *in'* to *ing*:** Whether or not all occurrences of *in'* are converted to *ing* (e.g., *Breakin' Down* should match *Breaking Down*).

- **Convert personal titles to abbreviations:** Whether or not the following replacements are made: *Mister* to *Mr., Doctor* to *Dr.* and *Professor* to *Prof.* This is done because these types of contractions are common in certain types of music (e.g., *Professor Longhair* should mach *Prof. Longhair*).

- **Remove all periods:** Whether or not all periods are removed (e.g., *REM* should match *R.E.M.*).

- **Remove all commas:** Whether or not all commas are removed.

- **Remove all hyphens:** Whether or not all hyphens are removed.

- **Remove all colons:** Whether or not all colons are removed.

- **Remove all semicolons:** Whether or not all semicolons are removed.

- **Remove all quotation marks:** Whether or not all quotation marks are removed.

- **Remove all single quotes and apostrophes:** Whether or not all apostrophes and single quotes are removed.

- **Remove all brackets:** Whether or not all parentheses, square brackets and curly braces are removed.
- **Convert *and* to *&*:** Whether or not all occurrences of *and* are converted to *&* (e.g., *Simon and Garfunkel* should match *Simon & Garfunkel*).
- **Remove all occurrences of *the*:** Whether or not all occurrences of *the* are removed (e.g., *The Police* should match *Police*).
- **Remove all spaces:** Whether or not all spaces are removed. Their occasional omission or the erroneous usage of multiple consecutive spaces instead of one is a common typo in many metadata annotations.

All of the operations in this section are performed in the order that they are listed above and are cumulative. Any fields that become newly identical after these transformations are applied are noted and merged (as described in Section 8.4.4) before reordered word subset operations are applied, as described below.

#### 8.4.2.3 Reordered word subset operations

It is often the case that some annotators label performer or composer names using differing word orderings than other annotators. One annotator might label a recording as being written by *Johnny Cash,* for example, while another might label it as being by *Cash, Johnny*. These two strings have very high edit distances, and would therefore not be detected as probable matches by any of the edit distance metrics described in Section 8.4.2.1. However, if word ordering is ignored, the edit distance is only one (the comma), which is low enough to indicate a probable match. There are many other musical annotation scenarios where similar word reorderings would thwart naïve edit distance-based error detection, such as in the case of *Django Reinhardt & Stéphane Grappelli* compared to *Stéphane Grappelli & Django Reinhardt*.

Another common type of difference than can be problematic when naïve edit distance approaches are used concerns shortened versions of strings that refer to the same entity. The difference between *Duke Ellington* and *Duke Ellington & His Orchestra* is one example of this word subset problem, since recordings of Duke Ellington with his various orchestras are often found annotated with either of these labels.

Although the more sophisticated version of the edit distance that uses the subset threshold edit distance (described in Section 8.4.2.1) can sometimes detect such simple examples, it is not effective in cases where the word reordering and subset problems are combined. An example of such a case is *Hendrix, Jimi* compared to *The Jimi Hendrix Experience.*

jMusicMetaManager can therefore perform two additional types of error-detection processing independently of edit distance in order to deal with the reordered word and word subset problems, either individually or combined. To summarize, *reordered word subset operations* are designed to deal with cases where one field value should likely be identical to another, but where the words in one field value are a subset of the words in the other and/or the words are in a different order.

Reordered word subset operations operate by first *tokenizing* each string in the title, artist, composer, album and genre fields into words by partitioning using whitespace characters. This is to say that every metadata label string is converted into a set of shorter strings, each of which is called a *token,* where each token consists of a word in the parent string. A word is defined to be any portion of the parent string that has a space to its left and/or right in the original string. Two kinds of operations can then be applied to these tokens by jMusicMetaManager:

- **Word ordering analysis:** This operation checks whether two labels are likely intended to refer to the same entity, but do not because the words are out of order. The percentage of tokens that match, regardless of order, is calculated and compared to a user defined threshold. The denominator used in the calculation of this percentage is the number of words in the field with the largest number of words. If 100% is selected, for example, any two labels will be marked as probable matches if and only if they contain exactly the same words, but in any order. More lenient percentages allow matches to still be reported even if some words do not match.

- **Word subset analysis:** This operation checks whether two labels are likely intended to refer to the same entity, but do not because the words in one are a subset of the words in the other, and are also potentially out of order. The user specifies a minimum percentage of tokens that must match between the two field values, regardless of order, in order for the labels to be considered likely to refer to the same entity. The denominator used in the calculation of this percentage, unlike the denominator used in the word ordering analysis processing described above, is the number of words in the field with the fewer number of words.

These two operations are performed after all find/replace operations described in Section 8.4.2.2 have been performed, in order to facilitate matches between appropriate tokens. A notable exception to this is the removal of spaces option, which is performed immediately after the reordered word subset operations, since spaces are needed for tokenization to be performed.

The token matching that is used requires tokens that are fully identical in order for them to be marked as matching, and would therefore, for example, mark *Charles Mingus* as corresponding to *Mingus Charles*, but not to *Mingus Charlie*, unless a very permissive threshold of 50% were used. The reordered word subset operations are performed separately from edit distance calculations, which are later applied to detect probable matches such as *Charlie* compared to *Charles* that can occur within word tokens. The separate and sequential application of processing that treats words as fixed units and processing that treats each character separately makes jMusicMetaManager more flexible from the user's perspective, and has also been found during informal experimentation to result in fewer false positives than processing that simultaneously applies token matching and edit distance matching.

#### 8.4.2.4 False-positive filters

The choice of the particular error detection parameterization to use when processing a music collection with jMusicMetaManager can sometimes be a balancing act between minimizing the probability that an error will be missed while at the same time minimizing *false positives,* which is to say field values that are marked as likely corresponding to the same entity that in fact refer to different entities (e.g., *Corelli* and *Torelli*). These two goals are often at odds because making the various detection thresholds more lenient tends to reduce the number of *false negatives,* or differing field values that in fact refer to the same entity but are not detected as such, but at the cost of sometimes increasing the number of false positives.

Recording titles are particularly vulnerable to false positives. This is partly because music collections tend to contain many more unique title labels than labels in other fields (because the same artist, composer, album and genre label are typically used in multiple recordings) and partly because it is common for legitimately different recordings to have the same or very similar titles. Two recordings might correctly have the same title but be performed entirely differently, for example, such as in the case of an original Led Zeppelin song compared with a Dread Zeppelin cover, or in the case of live and studio versions of the same song both performed at different times by Led Zeppelin. Although it is essential that redundant versions of the same recordings be detected, in order to avoid overlap between training and testing or validation sets as well as to avoid wasted space in general, it is also important not to highlight many recordings that are legitimately different.

jMusicMetaManager therefore includes filters to reduce the number of false positives in the title field. The input to these filters are the sets of recordings with the same or similar titles that are produced by edit distance and reordered word subset processing. The filters selected by the user then remove those recordings that are likely to in fact be legitimately different. The filters offered by jMusicMetaManager are as follows:

- **Filter by duration:** Recordings with similar titles are eliminated from consideration as redundant duplicates if they have durations whose difference is less than some percentage of their lengths that is specified by the user.
- **Filter by artist:** Recordings with similar titles are eliminated from consideration as redundant duplicates if they have different artist field values.
- **Filter by composer:** Recordings with similar titles are eliminated from consideration as redundant duplicates if they have different composer field values.
- **Filter by genre (at least one common):** Recordings with similar titles are eliminated from consideration as redundant duplicates if they do not have at least one genre field value in common.
- **Filter by album:** Recordings with sufficiently similar titles are eliminated from consideration as redundant duplicates if they have the same album field values.

This option is useful for dealing with situations such as extended albums containing both studio and live versions of a piece, for example.

These filters are applied after all selected find/replace, reordered word subset and edit distance operations have been performed.

Some of these filters are more appropriate for certain types of music collections than others, and should thus be used only with informed caution. While the *Filter by duration* option, for example, is typically appropriate for most music collections, the *Filter by genre* option is only suitable for collections where the recordings have been reliably and consistently been labelled by genre.

### 8.4.3 Reports generated

jMusicMetaManager can generate a variety of reports after analyzing a music collection, including reports that statistically profile and analyze the collection, reports of probable metadata errors and inconsistencies, and processing reports that can be used for debugging jMusicMetaManager or acquiring detailed technical information. The sub-sections below describe each of the reports that can be generated.

All reports generated by jMusicMetaManager can be viewed directly in jMusicMetaManager and can also be saved to disk as a set of frames-based HTML files that can be published on-line and viewed using any web browsing software. The user may specify which reports to generate, and is only permitted by the interface to select those reports that are appropriate given the user-selected processing options.

#### 8.4.3.1 Music collection profiling reports

The reports described in this sub-section provide descriptive details about music collections. This information can be useful for purposes such as statically profiling, cataloguing, planning expansions to, organizing and otherwise examining music collections and publishing information on them. The corresponding reports that jMusicMetaManager can generate are as follows:

- **All recordings:** This report consists of a table describing all available metadata for each recording sorted in alphabetical order by title.

- **Artist breakdown:** This report lists the names of all artists found in the music collection, along with the number of recordings by each artist and the percentage of the total recordings in the collection that this represents. The numbers of unique composers, albums and genres that have at least one recording associated with each artist are also reported.
- **Composer breakdown:** This report is similar to the *Artist breakdown* report, but with each unique composer listed instead of each unique artist.
- **Genre breakdown:** This report is similar to the *Artist breakdown* report, but with each unique genre listed instead of each unique artist.
- **Artists listed by genre:** This report lists all genres found in the music collection. A table is generated under each genre that includes a line for each artist that has at least one recording belonging to the corresponding genre. Each line in each table includes the artist's name, the number of recordings that the artist has in the collection that belong to the genre under consideration and the percentage of the total number of recordings in the genre that this represents.
- **Composers listed by genre:** This report is similar to the *Artists listed by genre* report, but with the tables listing composer names rather than artist names.
- **Albums listed by artist:** This report lists the names of all artists found in the music collection. A table is generated under each artist's name that consists of a line for each album that includes at least one recording by the corresponding artist. Each line includes the album's name, the number of tracks by the given artist on the album, the total number of tracks on the album in total, the percentage of the tracks in the album that are be the artist under consideration (marked in bold if not 100%) and whether the album is tagged as being a compilation album.
- **Albums listed by composer:** This report is similar to the *Albums listed by artist* report, except that the albums are broken down by composer rather than artist.
- **Incomplete albums:** This report consists of a list of all albums that are missing tracks. The percentage of tracks that are actually present in the music collection, whether the album is marked as being a compilation album and the artist name(s)

corresponding to the recordings on the album are also provided. If the music collection includes fewer than a user specified percentage of the tracks on the album then the album is marked in bold. A second list is also generated that indicates all albums that do not include metadata on the total number of tracks in the album, and for which whether or not the album is present in the collection in completion could therefore not be determined.

- **Artists with few recordings:** This report lists all artists with fewer recordings in the music collection than some threshold specified by the user. Artists with fewer than half this number of recordings are listed in bold.
- **Composers with few recordings:** This report is similar to the *Artists with few recordings* report, but it is generated based on the number of composers present in the collection rather than the number of artists.
- **Comment statistics:** This report alphabetically lists all unique comment tags found in the recordings, as well as the number of recordings marked with each comment and the percentage of all recordings that this represents. The numbers of unique artists, composers, albums and genres that have at least one recording tagged with each comment are also reported.
- **Exactly identical recording titles:** This report lists all recordings that have exactly the same title, before any processing such as find/replace operations have been applied. Both a summary list of all titles that occur more than once and sets of tables describing each individual recording making up a cluster of recordings with the same title are provided. This report can provide a useful indication of the number of versions of the same piece existing in the music collection, although it is at this point unknown whether these duplicates are duplicates of the same recording or are different versions of the same piece or different pieces with the same name. The *Probable duplicates of the same recording* report described in Section 8.4.3.2 can be useful for resolving this ambiguity.

Note that none of the reports described above indicate probable metadata errors. This is because normal practice would be to generate these reports only after metadata errors have first been detected using the reports described in Section 8.4.3.2 and then corrected.

#### 8.4.3.2 Reports of probable errors and inconsistencies

The reports referred to in this section indicate probable metadata errors that have been detected by jMusicMetaManager. These reports in general each indicate one of four main types of problems: redundant duplicate recordings, metadata fields that should likely be the same but are not, recordings missing important metadata or containing logically contradictory metadata and inconsistencies between iTunes XML files and actual MP3 files. The user can use this information to manually correct the metadata using software such as Apple iTunes.

Most of these reports report errors in the form of clusters of related errors. It is possible that the errors in each cluster may have been detected using different kinds of error detection, such as both reordered word subset processing and edit distance calculations. For example, *Jackson Mahalia*, *Mahalia Jackson* and *Mahal Jackson* would all be reported together in the same cluster.

Under its default settings, jMusicMetaManager has a tendency to err on the side of caution, and has a bias towards avoiding false negatives at the expense of some false positives. The user may adjust this balance as desired by changing jMusicMetaManager's user settings. Such adjustments can also be useful for reducing overly large clusters of errors. More details on such user configurations are available in jMusicMetaManager's manual.

The error detection reports that jMusicMetaManager can generate are as follows:

- **Summary:** This report provides key overview statistics on the music collection and probable metadata errors detected in it. The statistics include the number of MP3s parsed from a user specified directory structure, the number of recordings for which metadata was extracted from a user specified iTunes XML file, the number of unique recordings when the two sources were combined and the total number of unique title, artist, composer, genre and album labels present in the collection. The number of recordings missing metadata labels for each of these fields is also provided. This report also indicates how many clusters of probable errors were found in each of these fields, how many recordings were implicated in total in the clusters for each field and the number of probable duplicate recordings found.

- **Probable duplicates of the same recording:** This report lists recordings that are likely to be redundant duplicates of the same recording. All recordings whose title fields are judged to be sufficiently similar based on any selected find/replace, reordered word subset and/or edit distance processing are grouped together into clusters. False-positive filters are then applied to each such cluster to eliminate inappropriate candidates. Only the surviving clusters of probable duplicates are listed in this report. Metadata on each surviving recording, including artist, album, track number, duration, etc., is also reported to help the user make his or her own evaluations as to whether the recordings truly are duplicates.

- **Probable errors in title field:** This report lists probable metadata errors in the title field of the music collection under consideration. All recordings whose title fields are judged to be sufficiently similar based on any selected find/replace, reordered word subset and/or edit distance processing are grouped together into clusters and listed in this report. Each such cluster indicates a set of differing yet sufficiently similar metadata values. Additional metadata on each recording in each of these clusters is reported, including its artist, album, track number, duration, etc. Details on the specific processing operations that resulted in the detected similarities are also provided. Note that this report differs from the *Probable duplicates of the same recording* report in that no false-positive filters are applied. This is useful since, even if two recordings are in fact different versions of the same piece (e.g., live and studio recordings) and are thus filtered out from the *Probable duplicates of the same recording* report, one title may still be mistakenly spelled different from the other, and thus should be highlighted in this report for correction.

- **Probable errors in artist field:** This report lists probable errors in the artist field of the music collection under consideration. It is generated in a very similar way to the *Probable errors in title field* report, except that processing and reported errors correspond to the artist field rather than the title field.

- **Probable errors in composer field:** This report is similar to *Probable errors in title field* report, except that processing and reported errors correspond to the composer field rather than the title field.
- **Probable errors in album field:** This report is similar to *Probable errors in title field* report, except that processing and reported errors correspond to the album field rather than the title field.
- **Probable errors in genre field:** This report is similar to *Probable errors in title field* report, except that processing and reported errors correspond to the genre field rather than the title field.
- **Report on compilation albums:** jMusicMetaManager uses the notion of compilation albums that is most consistent with that used by iTunes, which is to say that an album should be marked as a compilation if and only if not all of its component recordings have the same value in the artist field. Three different sections are included in this report. The first simply lists all unique albums that contain at least one recording marked as a compilation or with an unknown compilation status. Albums where some recordings are marked as compilations and others are not are marked in bold, since all tracks in an album should have the same compilation marking. The second part lists all albums that contain recordings with different values in their artist fields but are incorrectly not marked as compilations. The third section lists albums that contain only recordings that have the same artist field value but are incorrectly marked as compilations.
- **Albums with duplicate or unknown track numbers:** This report consists of a list of all albums with the same name that contain more than one recording with the same track number or that contain one or more recordings that do not have a track number specified.

- **Recordings missing key metadata:** This report lists all recordings that have empty title, artist, composer, album and/or genre fields. A separate table is produced for each of these fields.
- **Albums with unspecified year:** This report lists all albums that have one or more recordings that do not have the recording year annotated.
- **Files in iTunes XML but not at specified path:** This report lists all recordings that are parsed from the user specified iTunes XML file for which there are no valid readable audio files at the specified paths. The user may of course choose not to generate this report if he or she wishes to process an iTunes file without access to the corresponding audio files.
- **Recordings in one source but not the other:** This report lists any recordings that were found in the user specified iTunes XML file but for which corresponding MP3 files were not found in the user specified audio directory structure, and vice versa. This differs from the *Files in iTunes XML but not at specified path* report in that this other report only checks to see if files referred to in the iTunes file actually exist, and does not also check to see if other audio files that might be in the user specified directories are also in the iTunes file.
- **MP3 files found that could not be parsed:** This report lists all files with an MP3 extension that were found but whose ID3 tags could not be parsed.
- **Non-MP3 files found:** This report lists all files in audio directories specified by the user that do not have an MP3 extension. This can be useful for detecting files that were erroneously placed in audio directories.
- **Fields that do not correspond between sources:** This report lists any differences in significant fields in the metadata for each recording parsed from the iTunes XML file and its corresponding MP3 file.

#### 8.4.3.3 Technical reports

The reports described in this section are not intended directly for use in music collection profiling or error checking. Rather, they are useful for tracking the intermediate processing stages for the purpose of debugging existing and new functionality and for comparing processing using different settings. Although some of these reports do indicate probable metadata errors, the same information is summarized in a more easily consumable form in the reports described in Section 8.4.3.2. The technical reports that can be generated are as follows:

- **Options selected:** This report indicates the types of processing selected by the user, their selected parameters and the names of the reports chosen to be generated. This can be useful for maintaining a record of the settings that resulted in a given set of processing results.
- **Processing time log:** This report indicates the duration of each type of processing performed by jMusicMetaManager.
- **All recordings parsed (before merge):** This report lists the path, title and artist of all recordings for which metadata was extracted, before any processing was performed. Up to two separate lists are produced, one for the metadata extracted from ID3 tags and one for the metadata extracted from an iTunes XML file.
- **All post-iTunes and ID3 merge metadata:** This report lists all available metadata for each recording immediately after metadata has been extracted from an iTunes file and/or the ID3 tags of MP3 files. If both sources are used, the report is generated after the metadata for each file referred to in the iTunes XML file has been merged with the metadata extracted from the tags of the MP3 file with the corresponding path, so that there is only one combined set of metadata for each unique file path.
- **Fields starting with a space:** This report separately lists all title, artist, composer, album and genre field values that start with a space. This report is useful in immediately highlighting this particularly common error that is difficult to detect visually.
- **Fields differing only in case:** Entries listed in this report indicate metadata values that are identical between any two recordings in all ways except letter case (e.g., *reggae* and *Reggae*). This is done before any other find/replace, reordered word

subset or edit distance processing has been performed. One list is produced for each of the title, artist, composer, genre and album fields.

- **Detailed replacements made:** This report provides details on all changes that were made during find/replace operations after lowercase conversion (if requested) and before any other processing.
- **Newly identical fields after find and replace:** This report lists all metadata values that were made newly identical after find/replace operations were applied
- **Fields with scrambled word orderings:** This report lists metadata values that contain identical words, but in any order, and where sometimes the words in one field are a subset of the words in the other. For example, occurrences of both *Martha Wainwright* and *Wainwright Martha* would be reported here. This is a summary of some of the processing described in Section 8.4.2.3.
- **Fields whose words are subsets of another:** This report lists metadata values that contain at least some matching words, and where the words may be in any order. For example, occurrences of both *The Royal Philharmonic Orchestra* and *The Royal Philharmonic* might be reported here. This is a summary of some of the processing described in Section 8.4.2.3. Items that might have otherwise been listed here may be included in the *Fields with scrambled word orderings* report instead if this report is also set to be generated.
- **Edit distances:** This report lists the actual edit distances between metadata values from all pairs of different recordings for each of the title, artist, composer, album and genre fields. One table is generated for each field. The tables provide up to three edit distance values, depending on the preferences selected by the user, namely the absolute, proportional and the subset edit distances, as described in Section 8.4.2.1. Any distances that fall below the thresholds set by the user are marked in bold. The reported edit distances are calculated after any selected find/replace and/or reordered word subset operations have been performed and any resulting merging of recordings has occurred.

- **Filtered candidate duplicate recordings:** This report lists all recordings that were found to have sufficiently similar titles to be marked as likely corresponding to the same piece, but were in the end rejected as being duplicate recordings because of the selected filters of the type described in Section 8.4.2.4.

### 8.4.4 Overview of the error detection algorithms

The list that follows provides a step-by-step outline of the essential processing performed by jMusicMetaManager when searching for metadata errors. This list revisits the operations described in the subsections of Section 8.4.2, but from an integrated implementation perspective. This implementation-level detail is included here because some of the processing algorithms are original. The list is still only a broad overview, however, as there are far too many details to cover here in completion. The algorithms used to generate the reports themselves are omitted entirely because of space limitations. Much more detail is available in the code documentation, particularly in the *AnalysisProcessor* class. Note that some of the operations described below may sometimes be omitted during processing, depending on the options selected by the user.

1) Parse metadata from the user specified iTunes XML file and/or ID3 tags from MP3 files in the user specified directory and its sub-directories.

2) If both of the metadata sources from Step 1 are being used, then the metadata is *merged* for corresponding iTunes references and MP3 ID3 tags, so that there is only one set of metadata per unique file path. Inconsistencies and omissions between the two sources are noted. The metadata extracted from the iTunes file is used by default in further processing when there is a disagreement between the two sources for a field. An exception to this is when a field is empty in the iTunes file but present in the ID3 tags, in which case the value from the ID3 tag is used.

3) The metadata for each recording is stored in a separate *RecordingMetaData* object that has a field for each type of metadata processed by jMusicMetaManager as well as methods for, among other things, comparing, organizing, importing and exporting the metadata.

4) The five main metadata fields (recording title, album title, artist name, composer name and genre names) are treated specially, as follows:

    - The value for each such field is stored in a separate *Entry* object for each recording. There are five Entry objects per recording, one for each of the five main fields. Each Entry holds an identifying string (e.g., a recording title or a composer name) as well as a vector of references pointing to the unique recording(s)[240] that it is associated with.

    - All Entry objects for a given field are stored in an *Entries* object. There are five Entries objects, one for each main metadata field. This arrangement makes it possible to efficiently simultaneously independently sort and otherwise organize recordings differently for different metadata fields.

5) All of the Entry objects in each of the five Entries objects (corresponding to recording title, album title, artist name, composer name and genre name) are sorted alphabetically based on their identifying string.

6) Entry objects with identical text fields in each Entries object are *merged*. This is a different kind of merge than that described in Step 2, and all merging referred to in the remaining steps of this section refer to this second type of merging.

    - In this new type of merging, all Entry objects with an identical string in a given Entries object are merged into one new Entry object, and the original source Entry objects are then deleted.

    - The new Entry object maintains references to all of the recordings that were referred to by the source Entry objects, which were merged into the new Entry object. Since only Entry objects with identical strings are merged, the new Entry object stores the same string as each of the source Entry objects did.

    - In order to do this processing, it is only necessary to compare each Entry object's string with that of the Entry object that immediately follows it. This is because all Entry objects were alphabetically sorted in Step 5.

[240] In this step, there is always only one recording referred to per Entry. However, multiple recordings can later be referred to by an Entry object via Entry merging, as described in Step 6.

- This overall approach has the advantages of conceptually simplifying later processing (similar merging occurs in many of the following steps), maintaining links between clusters of similar strings (e.g., *The Zoobombs, Zoobombs* and *Zoobomb*) and speeding up processing by reducing the number of strings to be processed.
- For example, consider a case where there are ten recordings with ten unique titles, all by one artist. In this case, let us say that each recording belongs to one or more of a set of three genres. There would originally be ten RecordingMetaData objects, ten title Entry objects, ten artist Entry objects and ten genre Entry objects. The following would result after merging:
    - There would still be ten RecordingMetaData objects, since it is only the Entry objects that are merged.
    - There would be one artist Entry objects, since all ten of the original artist Entry objects referred to the same artist, and were thus merged. This Entry object would include ten references to recordings, one for each of the ten recordings.
    - There would be three genre Entry objects, since there are three unique genres. Each of these Entry objects would refer to one to ten recordings, depending on how many are annotated as belonging to the given genre.
    - There would still be ten title Entry objects, each with a reference to one recording, because there are ten unique titles. However, even if there had been multiple recordings with the same title, they would not have been merged at this point, as recording titles are a special case that are not merged until Step 8 (unlike artist names, composer names, album titles and genres). The exception is in place in order to facilitate the generation of certain reports.
- As will soon become apparent, Entry merging is often used in jMusicMetaManager as an essential way of detecting and cataloguing errors. Vectors of *MergeReport* objects that hold details about all merges that occur

are stored in each Entries objects. These MergeReport objects can be used to generate various reports

7) Fields starting with spaces are detected, if this find/replace operation is selected by the user. This is performed separately from other find/replace operations for the purpose of generating a specialized report.

8) Title Entry objects with identical titles are merged, as in Step 6.

9) If this option is selected, Entry object strings are all converted to lowercase and then re-merged, as in Step 6. The MergeReports that result therefore indicate all fields that differ only in case (e.g., *van Morrison* vs. *Van Morrison*).

10) All selected find/replace operations are performed (see Section 8.4.2.2) and associated reports are generated in order to detect strings that differ but are likely meant to refer to the same entity in one of a number of common ways (e.g., *Dr. John* vs. *Doctor John*).

    - These operations involve searching all of the Entry object strings to check if they contain as a sub-string one of a set of specified strings. If they do, this sub-string is replaced with another specified string, or deleted, as appropriate.

    - The Entry objects in each Entries object are then resorted and remerged, as described in Step 6. MergeReport objects resulting from these merges then indicate those strings that were different before a find/replace operation was performed and then became identical after the replacement was made.

11) All selected word ordering and subset operations are performed (see Section 8.4.2.3) and associated reports are generated. This is done in order to detect strings that refer to the same entity but use different word orderings (e.g., *Billie Holiday* vs. *Holiday Billie*) or contain a string that is a subset of another string (e.g., *Stevie Ray Vaughan* and *Stevie Ray Vaughan and Double Trouble*). Processing occurs as follows:

- All words in the string stored in each Entry object are tokenized based on whitespace characters. This means that a separate token is stored for each word in each string.
- The tokens for the string in each Entry object are compared to the tokens for each other Entry object in the same Entries object. The number of identical tokens, regardless of order, is calculated for each pair of Entry objects.
- The percentages of matching tokens between Entry pairs are compared to the threshold percentage chosen by the user for the *word ordering analysis* option, as described in Section 8.4.2.3. Entry objects with percentages falling within the specified range are merged, as described in Step 6. The post-merge Entry string is assigned the value of the source Entry object that had the longer string.
- The previous step is repeated for all Entry objects, but this time using the *word subset analysis* percentage.

12) All selected edit distance calculations and comparisons are performed (see Section 8.4.2.1) and associated reports are generated in order to detect strings that are spelled differently but are likely intended to refer to the same entity (e.g., *Sibelius* vs. *Sibellius*).
    - jMusicMetaManager calculates the edit distances between all pairs of Entry object strings in each Entries object.
    - Each pair of Entry objects is merged, as described in Step 6, if the edit distance is equal to or less than one of the three user-defined thresholds.

13) Error filtering (see Section 8.4.2.4) is applied to all MergeReport objects associated with titles. This is done in order to filter out titles that have sufficiently similar titles to have been detected by earlier inconsistency detection steps, but are in fact not likely to be redundant identical copies of the same recording (i.e., one might be a live recording compared to another which is studio recording). Recordings with the same or similar titles are therefore reported as probable

duplicates unless one or more of the conditions described in Section 8.4.2.4 is true.

### 8.4.5 jMusicMetaManager's interface

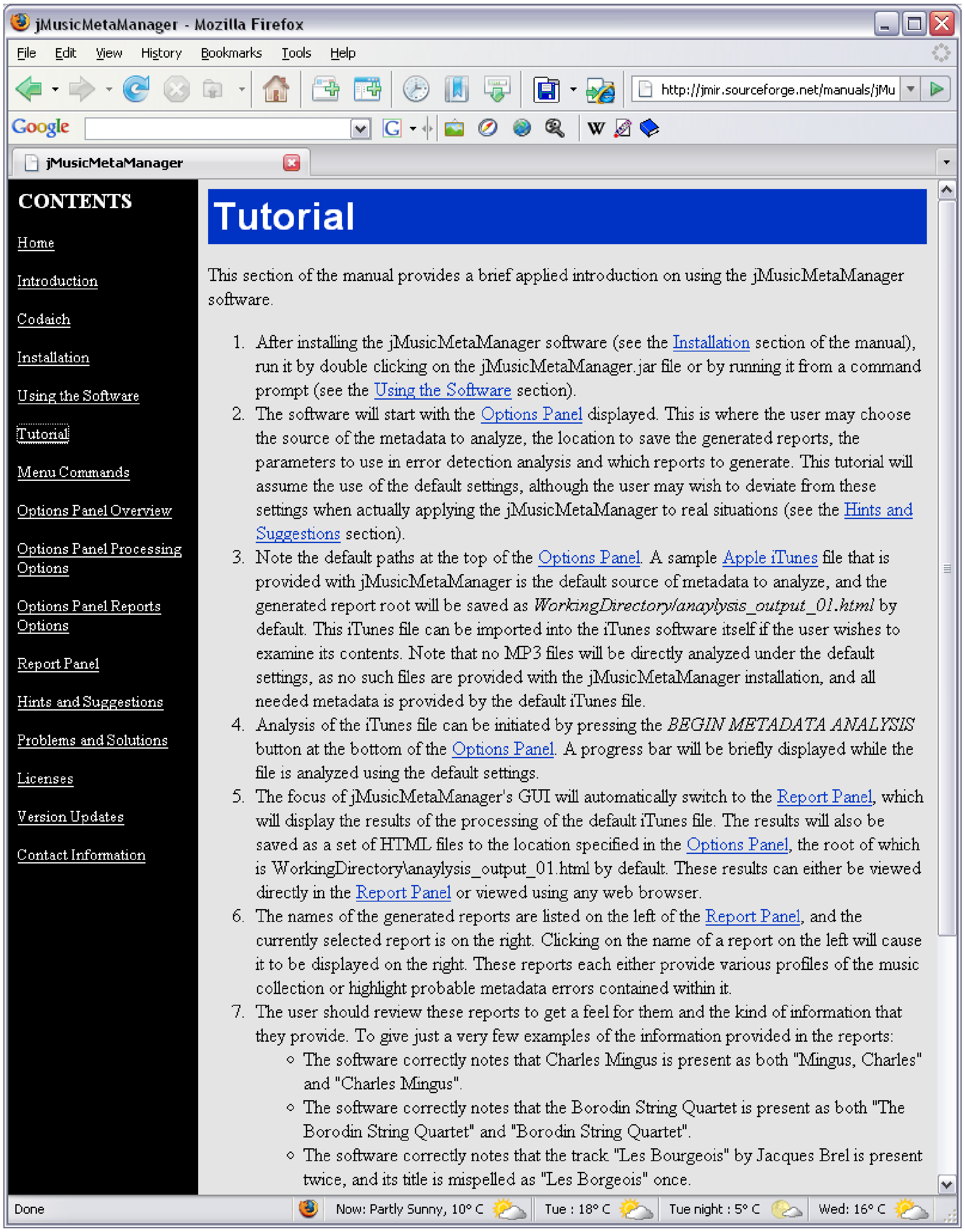
Like all of the jMIR components, jMusicMetaManager has been designed to be easy to use for researchers with a wide variety of technical backgrounds. This can range from MIR researchers to casual users who simply wish to catalogue their personal music collections to librarians dealing with major music collections to developers who wish to add functionality or modify the code to meet their own needs.

jMusicMetaManager is also designed to be applicable to many different kinds of music. The types of metadata errors that are common in popular music collections can be very different from those encountered in classical music collections, for example, and an error detection parameterization that is suitable for one type of collection can be ineffective for another.

jMusicMetaManager is therefore designed to be highly customizable, and allows the user to set 74 different parameters controlling error detection, music collection analysis and report generation. Since this number of options can be overwhelming for novice users, the default settings have been chosen such that reasonably good quality results can be achieved on reasonably large and diverse personal music collections.

For those wishing to perform customized processing, the user options are well-documented in a detailed HTML manual (Figure 8.1) that also includes general guidelines on choosing effective settings, a step-by-step tutorial and general reference information. This manual can be viewed either using a web browser or via jMusicMetaManager's built-in help functionality.

The jMusicMetaManager GUI is divided into two main panels. The first, the *Options Panel,* (Figure 8.2), allows the user to choose the source(s) of the metadata to be processed and the settings to be use when doing so. The second panel, the *Report Panel,* is used to display generated reports, which can alternatively be saved to disk and viewed using a web browser (Figure 8.3). There is also a menu bar that can be used to perform miscellaneous tasks such as restoring the default options, exporting metadata to Weka ARFF or ACE XML files, and accessing help functionality.


jMusicMetaManager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://jmir.sourceforge.net/manuals/jMu
Google
jMusicMetaManager
Done
Now: Partly Sunny, 10° C
Tue : 18° C
Tue night : 5° C
Wed: 16° C

**CONTENTS**

Home

Introduction

Codaich

Installation

Using the Software

Tutorial

Menu Commands

Options Panel Overview

Options Panel Processing Options

Options Panel Reports Options

Report Panel

Hints and Suggestions

Problems and Solutions

Licenses

Version Updates

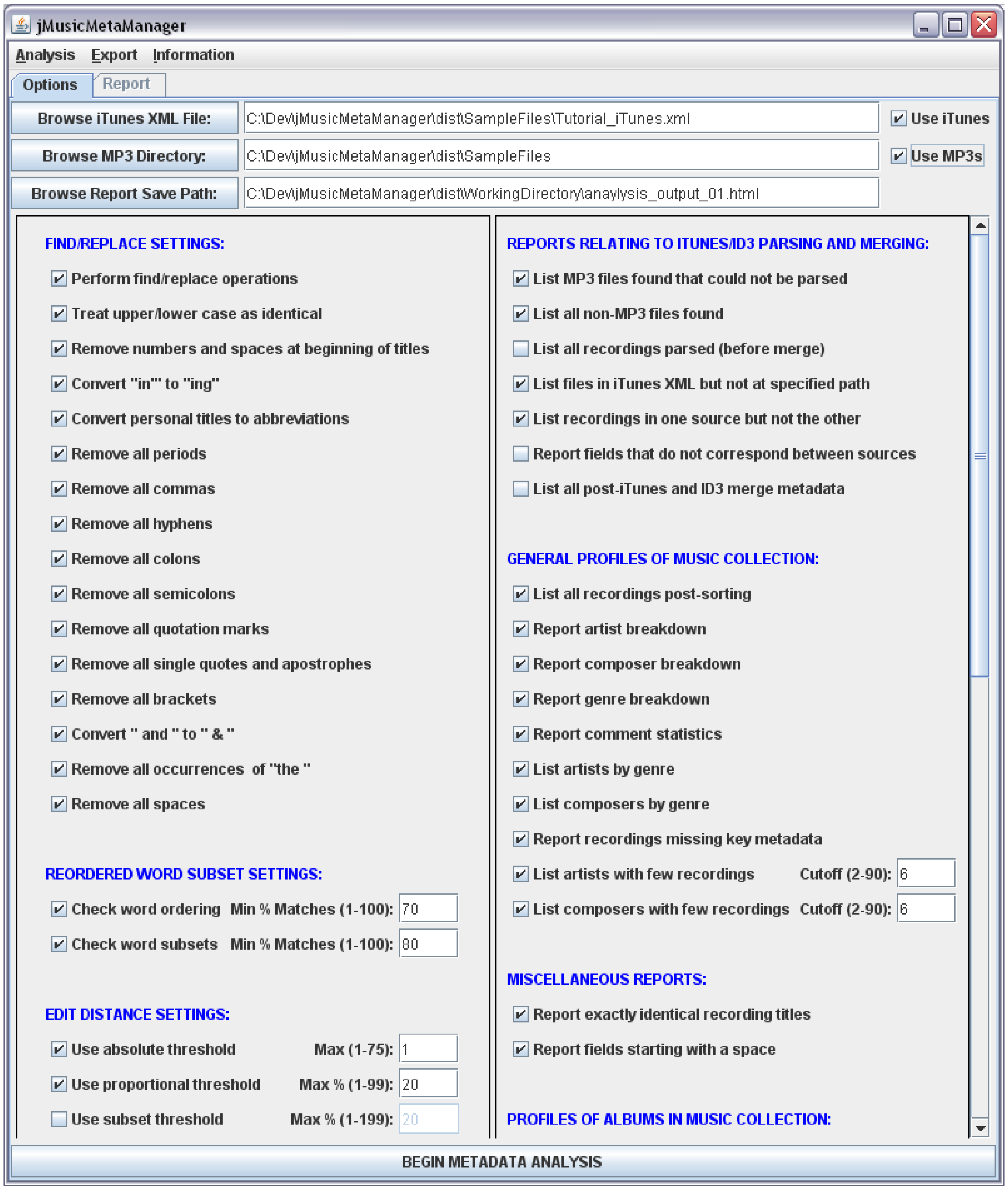
Contact Information

**Tutorial**

This section of the manual provides a brief applied introduction on using the jMusicMetaManager software.

1. After installing the jMusicMetaManager software (see the Installation section of the manual), run it by double clicking on the jMusicMetaManager.jar file or by running it from a command prompt (see the Using the Software section).
2. The software will start with the Options Panel displayed. This is where the user may choose the source of the metadata to analyze, the location to save the generated reports, the parameters to use in error detection analysis and which reports to generate. This tutorial will assume the use of the default settings, although the user may wish to deviate from these settings when actually applying the jMusicMetaManager to real situations (see the Hints and Suggestions section).
3. Note the default paths at the top of the Options Panel. A sample Apple iTunes file that is provided with jMusicMetaManager is the default source of metadata to analyze, and the generated report root will be saved as *WorkingDirectory/anaylysis_output_01.html* by default. This iTunes file can be imported into the iTunes software itself if the user wishes to examine its contents. Note that no MP3 files will be directly analyzed under the default settings, as no such files are provided with the jMusicMetaManager installation, and all needed metadata is provided by the default iTunes file.
4. Analysis of the iTunes file can be initiated by pressing the *BEGIN METADATA ANALYSIS* button at the bottom of the Options Panel. A progress bar will be briefly displayed while the file is analyzed using the default settings.
5. The focus of jMusicMetaManager's GUI will automatically switch to the Report Panel, which will display the results of the processing of the default iTunes file. The results will also be saved as a set of HTML files to the location specified in the Options Panel, the root of which is WorkingDirectory\anaylysis_output_01.html by default. These results can either be viewed directly in the Report Panel or viewed using any web browser.
6. The names of the generated reports are listed on the left of the Report Panel, and the currently selected report is on the right. Clicking on the name of a report on the left will cause it to be displayed on the right. These reports each either provide various profiles of the music collection or highlight probable metadata errors contained within it.
7. The user should review these reports to get a feel for them and the kind of information that they provide. To give just a very few examples of the information provided in the reports:
   - The software correctly notes that Charles Mingus is present as both "Mingus, Charles" and "Charles Mingus".
   - The software correctly notes that the Borodin String Quartet is present as both "The Borodin String Quartet" and "Borodin String Quartet".
   - The software correctly notes that the track "Les Bourgeois" by Jacques Brel is present twice, and its title is mispelled as "Les Borgeois" once.

**Figure 8.1:** The tutorial portion of jMusicMetaManager's user manual, which can be viewed either via the software itself or, as shown here, through a web browser.



Browse iTunes XML File: C:\Dev\jMusicMetaManager\dist\SampleFiles\Tutorial_iTunes.xml [✓] Use iTunes

Browse MP3 Directory: C:\Dev\jMusicMetaManager\dist\SampleFiles [✓] Use MP3s

Browse Report Save Path: C:\Dev\jMusicMetaManager\dist\WorkingDirectory\anaylysis_output_01.html

FIND/REPLACE SETTINGS:

- [✓] Perform find/replace operations
- [✓] Treat upper/lower case as identical
- [✓] Remove numbers and spaces at beginning of titles
- [✓] Convert "in'" to "ing"
- [✓] Convert personal titles to abbreviations
- [✓] Remove all periods
- [✓] Remove all commas
- [✓] Remove all hyphens
- [✓] Remove all colons
- [✓] Remove all semicolons
- [✓] Remove all quotation marks
- [✓] Remove all single quotes and apostrophes
- [✓] Remove all brackets
- [✓] Convert " and " to " & "
- [✓] Remove all occurrences of "the "
- [✓] Remove all spaces

REORDERED WORD SUBSET SETTINGS:

- [✓] Check word ordering Min % Matches (1-100): 70
- [✓] Check word subsets Min % Matches (1-100): 80

EDIT DISTANCE SETTINGS:

- [✓] Use absolute threshold Max (1-75): 1
- [✓] Use proportional threshold Max % (1-99): 20
- [ ] Use subset threshold Max % (1-199): 20

REPORTS RELATING TO ITUNES/ID3 PARSING AND MERGING:

- [✓] List MP3 files found that could not be parsed
- [✓] List all non-MP3 files found
- [ ] List all recordings parsed (before merge)
- [✓] List files in iTunes XML but not at specified path
- [✓] List recordings in one source but not the other
- [ ] Report fields that do not correspond between sources
- [ ] List all post-iTunes and ID3 merge metadata

GENERAL PROFILES OF MUSIC COLLECTION:

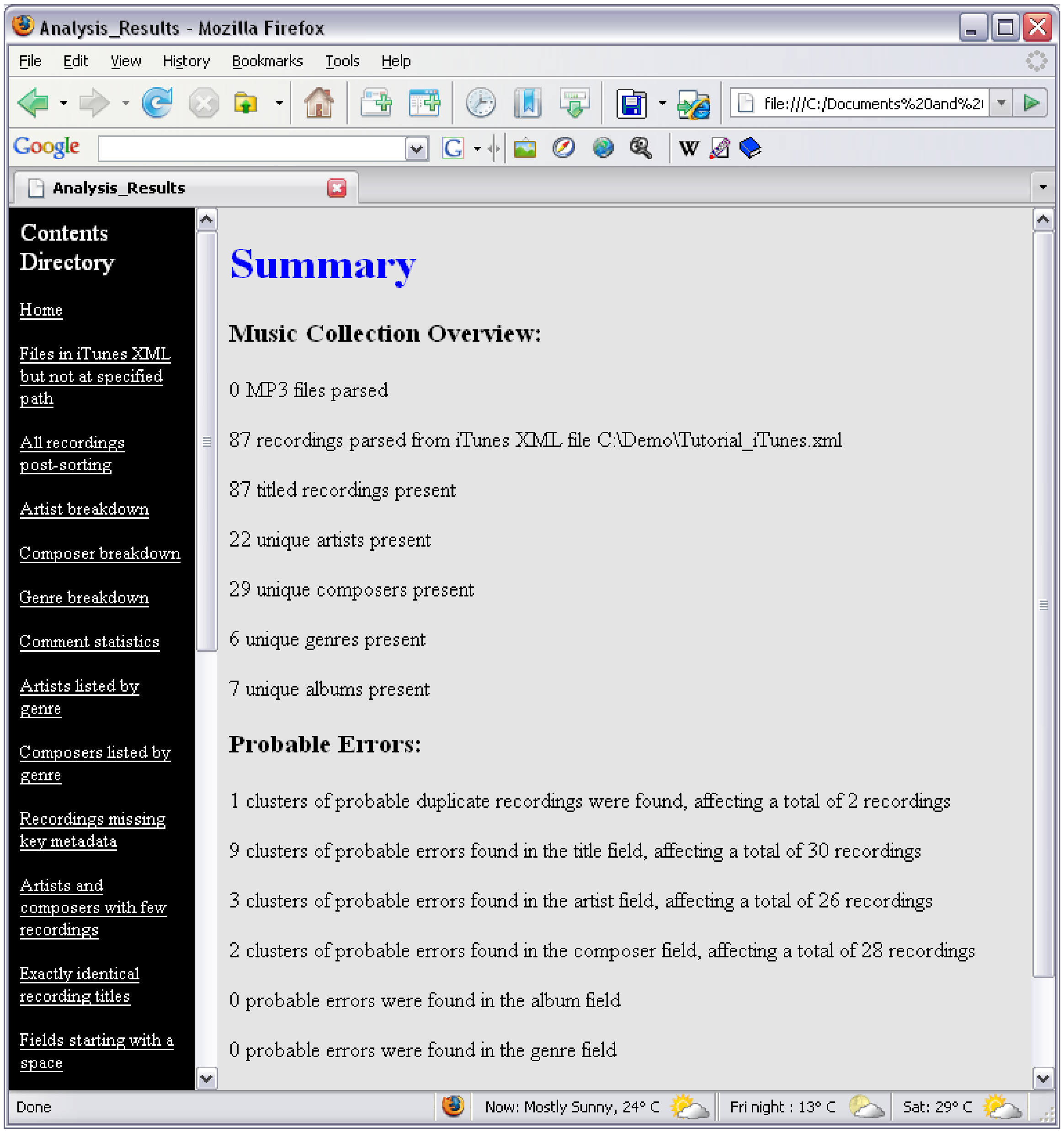
- [✓] List all recordings post-sorting
- [✓] Report artist breakdown
- [✓] Report composer breakdown
- [✓] Report genre breakdown
- [✓] Report comment statistics
- [✓] List artists by genre
- [✓] List composers by genre
- [✓] Report recordings missing key metadata
- [✓] List artists with few recordings Cutoff (2-90): 6
- [✓] List composers with few recordings Cutoff (2-90): 6

MISCELLANEOUS REPORTS:

- [✓] Report exactly identical recording titles
- [✓] Report fields starting with a space

PROFILES OF ALBUMS IN MUSIC COLLECTION:

**Figure 8.2:** jMusicMetaManager's *Options Panel,* which allows the user to choose which types of processing to apply, which reports to generate and what parameters to use during processing.


Analysis_Results - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Documents%20and%2
Google
Analysis_Results
Done
Now: Mostly Sunny, 24° C
Fri night : 13° C
Sat: 29° C

Contents Directory

Home

Files in iTunes XML but not at specified path

All recordings post-sorting

Artist breakdown

Composer breakdown

Genre breakdown

Comment statistics

Artists listed by genre

Composers listed by genre

Recordings missing key metadata

Artists and composers with few recordings

Exactly identical recording titles

Fields starting with a space

Summary

Music Collection Overview:

0 MP3 files parsed

87 recordings parsed from iTunes XML file C:\Demo\Tutorial_iTunes.xml

87 titled recordings present

22 unique artists present

29 unique composers present

6 unique genres present

7 unique albums present

Probable Errors:

1 clusters of probable duplicate recordings were found, affecting a total of 2 recordings

9 clusters of probable errors found in the title field, affecting a total of 30 recordings

3 clusters of probable errors found in the artist field, affecting a total of 26 recordings

2 clusters of probable errors found in the composer field, affecting a total of 28 recordings

0 probable errors were found in the album field

0 probable errors were found in the genre field

**Figure 8.3:** A sample report generated by jMusicMetaManager after analyzing a music collection. The *Summary Report* is shown here in particular. Additional reports can be accessed from the frame on the left.

## *8.5 The Bodhidharma MIDI dataset: A symbolic dataset*

### 8.5.1 Overview

The Bodhidharma MIDI dataset is a collection of 950 MIDI files intended for use as training and testing data in MIR research. It was originally assembled in order to evaluate the Bodhidharma MIDI genre classifier (McKay 2004), and has since also been used by others doing research in musical genre classification (e.g., DeCoro, Barutcuoglu, and Fiebrink 2007). However, the Bodhidharma MIDI dataset can also certainly be used for types of MIR research other than genre classification.

The Bodhidharma dataset includes recordings belonging to 38 different musical genres, with 25 recordings per genre. These genres are organized into a relatively sophisticated ontology, as described in Section 8.5.2. Details on the approaches used to collect and select the particular recordings that make up the Bodhidharma dataset are provided in Section 8.5.3.

Although 950 recordings is a smaller number of recordings than one would ideally prefer, the Bodhidharma dataset is to the best of the author's knowledge the largest and most diverse non-commercial MIDI collection that has been used to date in MIR research. This is likely because of the difficulty associated with acquiring reliable MIDI recordings representing a sufficiently diverse range of musical styles and artists. Although there is no shortage of MIDI recordings in many Classical styles, for example, it can be much harder to find more than a few recordings in many other genres, such as Rap. This problem is particularly emphasized if one wishes to avoid amateur compositions that may not be representative of the sorts of music that most listeners consume.

Details on the particular recordings that comprise the Bodhidharma MIDI dataset can be obtained by contacting Cory McKay at cory.mckay@mail.mcgill.ca.

### 8.5.2 Genre ontology used

This section describes the specialized ontological structuring that was used to organize the Bodhidharma dataset's genre classes, and specifies the motivations behind its development.

A good genre ontology should include coarse classes that are meaningful to the average, potentially relatively musically illiterate music consumer. At the same time, however, it is also desirable that the ontology provide the option of making classifications into finer classes that may be more useful to music specialists. A hierarchal tree-based structure fulfils these dual requirements. Broad classes, such as *Classical* or *Jazz,* are found at the root of the tree (topmost level of the tree), and classes become increasingly fine as one progresses towards the leaves (i.e. nodes in the tree without children).

As discussed in Section 8.2.2, utilizing such a hierarchical organization of classes can also have a number of machine learning benefits, including the ability to take advantage of hierarchical classification algorithms and the ability to used weighted training methodologies that penalize misclassifications between very dissimilar classes more than misclassifications between more similar classes.

A strictly hierarchical class structure can be somewhat limiting, however, which is why a modified tree structure is used in the Bodhidharma dataset. This ontological structure has two important differences from traditional tree structures:

- A given recording can be associated with more than one leaf genre.
- A sub-genre can be a direct descendant of more than one parent genre (e.g., *Blues Rock* is a descendant of both the *Blues* and *Classic Rock* parent genres in Figure 8.4).

These two modifications do complicate the organizational simplicity offered by traditional trees, but such modifications are necessary if one is to deal with certain fundamental realties: the boundaries between different genres are often vague, sub-genres are often the result of a complex amalgamation of potentially disparate parent genres and many recordings do not fall unambiguously into single genre classes.

The various branches of the modified tree used in the Bodhidharma dataset are permitted to vary in terms of both depth and breadth. This was necessary in order to accommodate the different degrees to which some real-life genres can be split into narrow sub-genres and others simply exist as undivided broad categories.

The particular genres included in the Bodhidharma dataset are shown in Figure 8.4, along with the structuring interrelating the classes This ontology includes 38 unique leaf genres, 9 root genres and eight intermediate genres, for a total of 55 unique genre labels.

A number of sources of information were consulted when designing this ontology. The final ontology is an amalgamation of the genre classes used by on-line and brick and mortar retailers, information found in scholarly writings on popular music, popular music magazines, music critic reviews, schedules of radio and video specialty shows, fan web sites and the personal knowledge of the author.

Particular use was made of the All Music Guide,[241] an excellent on-line resource, and of the Amazon on-line store.[242] These sites are widely used by people with many different musical backgrounds, and their ontologies provide among the best available representations of the types of genres that people actually use. These two sites are also complimentary, in a sense. The All Music Guide contains detailed and well-researched information, but does not establish clear relationships between genres. Amazon, in contrast, has a clear genre structure, but no informative descriptions of individual genres.

Given the limitations on the number and types of music that have been encoded into MIDI and made available on-line, it was unfortunately only practical to use a subset of the classes that would have ideally been included in the Bodhidharma dataset's ontology. The amount of time needed to manually find, download and classify recordings individually also imposed limitations on the number of classes that could be included. This ontology is, however, significantly larger and more diverse than that used by any other MIR research dataset known to the author, with the exception of Codaich (Section 8.6).

The ontology shown in Figure 8.4 is not always perfectly logical or consistent. This is necessary in order to truly evaluate MIR systems, as the types of genre structures that humans actually use are also usually illogical and inconsistent. The Bodhidharma ontology is not presented as perfect or as complete, but rather as a structuring that is useful for realistic research evaluations. The ontology encapsulates many of the difficulties, ambiguities and inconsistencies inherent to any realistic genre ontology, and is large and sophisticated enough that it provides a significantly more difficult and realistic test bed than is used in most past of contemporary MIR classification research.

In practice, much MIR research has involved only ten or so genre classes. Although such ontologies do not in general give a true representation of how well a system deals with a realistically diverse range of music, it can be useful to have access to a genre ontology of a similar size to that used in other research for the purpose of roughly comparing the performance of one's own system with published results on similarly sized ontologies. A subset of the Bodhidharma dataset has therefore been used in a smaller 9-

[241] www.allmusic.com
[242] www.amazon.com

class ontology, as shown in Figure 8.5. This reduced dataset includes only 225 of Bodhidharma’s 950 recordings. The choice of the particular genres to include in this reduced ontology was made such that both relatively similar and dissimilar genres would be included, so that one might evaluate how well a system can make both relatively easy and relatively difficult classifications.

**Country**
- Bluegrass
- Contemporary
- Traditional

**Jazz**
- *Bop*
  - Bebop
  - Cool
- *Fusion*
  - Bossa Nova
  - Jazz Soul
  - Smooth Jazz
- Ragtime
- Swing

**Modern Pop**
- Adult Contemp.
- *Dance*
  - Dance Pop
  - Pop Rap
  - Techno
- Smooth Jazz

**Rap**
- Hardcore Rap
- Pop Rap

**Rhythm and Blues**
- *Blues*
  - Blues Rock
  - Chicago Blues
  - Country Blues
  - Soul Blues
- Funk
- Jazz Soul
- Rock and Roll
- Soul

**Rock**
- *Classic Rock*
  - Blues Rock
  - Hard Rock
  - Psychedelic
- *Modern Rock*
  - Alternative Rock
  - Hard Rock
  - Metal
  - Punk

**Western Classical**
- Baroque
- Classical
- *Early Music*
  - Medieval
  - Renaissance
- Modern Classical
- Romantic

**Western Folk**
- Bluegrass
- Celtic
- Country Blues
- Flamenco

**Worldbeat**
- *Latin*
  - Bossa Nova
  - Salsa
  - Tango
- Reggae

**Figure 8.4:** The genres of music found in the Bodhidharma MIDI dataset and their associated ontological structuring.

**Jazz**
- Bebop
- Jazz Soul
- Swing

**Popular**
- Rap
- Punk
- Country

**Western Classical**
- Baroque
- Modern Classical
- Romantic

**Figure 8.5:** The Bodhidharma MIDI dataset’s reduced genre ontology.

### 8.5.3 Methodology used to collect and select recordings

This section describes the methodology that was used to acquire and select the particular recordings that make up the Bodhidharma dataset. Fortunately, there is a large community of individuals that encode commercially released music of many types into MIDI files and post them on-line, thus making it possible to harvest them for inclusion in this dataset. Although the quality of these encodings can vary widely, there are fortunately many encoders who take the time to generate good encodings. There are also a number of MIDI files that are professionally produced and sold, although these are much fewer in number and tend to be limited primarily to certain genres of music, such as Classical.

The first step towards assembling the Bodhidharma MIDI dataset was the compilation of a catalogue of web sites from which MIDI files could be downloaded. Although an emphasis was placed on sites that individually focused on particular genres, such sites were insufficient in number to meet the demands of Bodhidharma's large 38-genre ontology, so a number of general purpose sites and MIDI search engines were utilized as well.

The sites in this catalogue were then surveyed manually, and all MIDI recordings that sounded like they belonged to the genre classes in the ontology were downloaded. These collected recordings were then supplemented by searches for specific recordings chosen as prototypical examples of their genres. This was done by constructing lists of ten to twenty pieces that were particularly typical of each genre according to the All Music Guide web site and the experience of the author. At this point, 30 to 45 MIDI recordings were available for most of the leaf genres. Deficits in any individual genres were remedied by performing further searches for model recordings.

It was decided to use a combination of prototypical examples as well as more general examples in order to make it possible to evaluate how well MIR systems could deal with both typical examples and outliers. Using only prototypical examples would have risked biasing a trained models towards the author's potentially erroneous perception of genre classes, and would also have risked training models that would be unable to deal with recordings that are somewhat atypical of genres that they nonetheless clearly belong to.

Care was also taken to include examples of a range of different styles within each genre. So, for example, the Baroque genre includes recordings that are examples of operas, keyboard music, orchestral music, chamber music, etc. Of course, this approach can significantly increase the difficulty of classification, particularly since there are only 25 recordings per class to be divided into disjoint training, testing and validation sets, but it is necessary to achieve realistic success rates and avoid overtraining classifiers so that they are unable to recognize the full range of musical styles that one might find in any particular genre.

A number of past music classification experiments have taken the step of ensuring that their datasets do not include multiple recordings by the same artist in a given genre, in order to avoid deceptive success rates influenced by artist classification rather than genre classification. Flexer (2007), for example, provides experimental justification for such *artist filters*. The Bodhidharma MIDI dataset goes further by including some pieces by the same artists in different genre classes when pieces by artists who have performed in different genres are available. So, for example, the Bodhidharma MIDI dataset includes one song by Tupac Shakur in the Pop Rap genre as well as another in the Hardcore Rap genre. This approach is important for ensuring that a genre classification system is able to deal correctly with musicians who perform multiple genres of music, as often happens in practice.

The MIDI files for each genre were taken from a variety of sources whenever possible. This was done in order to even out any encoding particularities, as recordings from a single source might have encoding characteristics that a machine learning system could use as a basis for classification rather than actual musical characteristics. This could potentially artificially inflate classification performance and compromise the generality of leaned models.

All of the downloaded files were then re-reviewed one by one by the author, and classified based on the author’s experience, the All Music Guide and the genre label of the piece on the site that it was downloaded from, if available. Recordings were rejected if they were of a poor technical quality. Twenty-five pieces were then selected from the available pool for each leaf genre. Single pieces were permitted to belong to more than one genre, when appropriate.

The particular ceiling of twenty-five recordings per leaf genre was selected because of the time requirements involved in manually finding, downloading and classifying recordings, particularly given the large number of genres. An additional problem was that MIDI files are much harder to find in some genres than others. It was decided to use an equal amount of recordings for all leaf genres, as failure to stratify the dataset in this way could cause pattern recognition systems to find local minima in the solution space by simply ignoring genres with few recordings.

As a side note, many MIDI files contain metadata indicating genre. This metadata is not universally present, however, and there is not any consistent set of genre labels or classification standards that are used. When present, these labels tend to be far noisier even than the contents of the ID3 tags of MP3s found on the Internet. Genre identifications stored in MIDI files were therefore largely ignored during ground-truth labelling.

## *8.6 Codaich: An audio dataset*

### 8.6.1 Overview

Codaich is a very large collection of carefully labelled MP3 recordings intended for use in MIR research. The term *Codaich* is both the Gaelic word for share and a combination of the first names (COry, DAniel, ICHiro) of the authors of the paper in which Codaich and jMusicMetaManager were originally published (McKay, McEnnis and Fujinaga 2006). Codaich was assembled both for testing the jMIR components and as a prototype implementation of many of the design principles presented in Section 8.2.4.

Section 8.6.2 describes Codaich in some detail and provides background on how it was designed and assembled. Section 8.6.3 describes OMEN, a proposed system for legally sharing audio datasets among research groups. It is hoped that OMEN will eventually be used to make features extracted from Codaich available to the MIR community at large. Finally, Section 8.6.4 informally describes some of the most common types of metadata errors that were observed while assembling Codaich.

Those wishing to download metadata on the recordings that make up Codaich may do so by accessing jmir.sourceforge.net/index_Codaich.html. The information posted there includes the iTunes XML file for the collection and a number of profiling inventory reports generated by jMusicMetaManager.

### 8.6.2 Details of Codaich and the methodology used to assemble it

Codaich exists in two versions: the release version and the current version. Codaich is constantly growing, and this is reflected in the current version. Although efforts are made to keep Codaich's metadata correct and consistent as recordings are added, it is inevitable that some errors will initially be missed due to human fallibility, so the metadata in the current version is not always perfect. The release version, on the other hand, is that version of Codaich that has most recently had its metadata checked by jMusicMetaManager and then been correspondingly corrected. The release version is therefore more reliable for research purposes, and it is the metadata for this version that is posted on jmir.sourceforge.net. All of the statistics on Codaich that are reported in this section therefore refer to the release version, rather than to the current version, even though the current version is significantly larger.[243]

The release version of Codaich consists of 26,420 MP3 recordings belonging to 55 different leaf genres of music. This includes tracks by 2213 performing artists from 2036 different albums.[244] Codaich consists primarily of commercially produced music, since this is the type of music consumed by most listeners, but it also includes a significant number of recordings from independent labels.

The MP3 audio format in particular was chosen because it is by far the most popular digital audio file format in use among contemporary listeners. In order to realistically reflect the variety of file encodings found in practical use cases and the related difficulties from the perspective of automatic music classification (see Section 3.2.5), Codaich includes music digitized using a variety of encoders at many different bit rates.

The motivations, principles, rationales and techniques used to build Codaich's genre ontology and choose the kinds of recordings to include for each genre were similar to those used in constructing the Bodhidharma MIDI dataset, as described in Sections 8.5.2 and 8.5.3. One essential difference between the two datasets, however, is that audio files

[243] There are 32,066 recordings in the current version of Cocaich at the time of this writing, compared to 26,420 recordings in the release version.
[244] Mutli-volume albums are only counted once in this statistic. Also, 144 of these albums are compilation albums that each include tracks by multiple artists.

are much more widely and easily available than MIDI files. This means that many more recordings belonging to a greater variety of classes could be harvested for inclusion in Codaich than was possible for the Bodhidharma MIDI dataset.

As a result, an entirely different genre ontology was designed for Codaich, as illustrated in Figures 8.6 and 8.7. There are 55 unique leaf genres of music in Codaich, as opposed to 38 in the Bodhidharma dataset. Although there is certainly some overlap in the genre classes found in the two datasets, many of the individual classes also differ. The genres are structured differently as well, in order to more accurately conform to the particular music that is included in Codaich.

Much like the Bodhidharma MIDI dataset, Codaich is designed to be as realistically difficult as possible. This means that, for example, there are cases where pieces that belong to different genres but are recorded by the same artist are included in Codaich. The greater availability of audio recordings meant that this approach could be taken a step further with Codaich than was possible with the Bodhidharma dataset, however. For example, Codaich also includes some different versions of the same piece in different genres performed by different performers. So, for example, there is one Modern Blues performance of *Key to the Highway* by Derek and the Dominoes and another Traditional Blues performance by Brownie McGhee.

Codaich includes an unequal amount of recordings per genre, another thing that differentiates it from the Bodhidharma MIDI dataset. This was reasonable because the comparatively large number of audio files available in many genres made it possible to have a fairly large number of recordings in even the smallest genres in the Codaich ontology. Such unbalanced class sizes have the advantage of accommodating researchers who might wish to use only certain subsets of all of the genres available, and would not wish to have the number of recordings per genre limited by the number of recordings available in genres in which they may not be interested. Having said this, it is of course possible, and recommended for many scenarios, that stratified learning be used during actual training.

Codaich was originally seeded with an early version of the in-house music database of Douglas Eck's lab at the Université de Montréal and with the personal music collections of several graduate students at McGill University's Schulich School of Music. Lists were then drawn up of particularly desired recordings belonging to each of Codaich's genres, including recordings representing a range of styles within each genre as well as both prototypical examples of each genre and outliers. This was done in ways similar to those used in the building of the Bodhidharma dataset, as described in Section 8.5.3. As many of these recordings as possible were then harvested from both the extensive collection of the Marvin Duchow Music Library and resources that are available on the Internet.

In their original state after being imported into Codaich, all recordings were labelled with metadata based on either the Gracenote[245] labels accessed by iTunes, in the case of tracks imported from CDs, or, in the case of digital music files, based on pre-existing ID3 tags. In the case of tracks digitized from analog sources, such as cassette tapes or LPs, the tags were entered manually.

Of course, as discussed previously, such pre-existing tags tend to be very noisy and inconsistent. All tags from all recordings were thus manually checked and corrected by the author, using a number of sources, the most significant of which was the All Music Guide. Any relevant metadata fields that were originally empty were also filled in, so that at the end of this process, the following metadata fields were annotated for all recordings (when available):

- Recording Title
- Performer/Artist Name
- Composer Name[246]
- Genre(s)
- Recording Year[247]
- Album Title
- Track Number
- Total Tracks in Album
- Disc Number
- Discs In Set
- Compilation Status

[245] www.gracenote.com
[246] Composer Names have to date only been consistently annotated for the Classical portion of Codaich. The composer labeling of the remainder is an ongoing process.
[247] The Recording Year field is not yet fully annotated for all recordings in Codaich.

- Bit Rate
- Track Comments
- Recording File Path

Finally, this metadata was all processed by jMusicMetaManager in order to detect probable errors and inconsistencies. Appropriate corrections were then made.

**Blues**
- Contemporary Blues
- Country Blues
- Urban Blues

**Country**

**Dance Pop**

**Electronica**

**Hip Hop / Rap**

**Instrumental Pop**

**Modern Folk**
- Alternative Folk
- Singer / Songwriter

**R&B**
- Contemporary R&B
- Funk
- Gospel
- Rock & Roll
- Soul

**Reggae**

**Rock**
- Alternative Pop / Rock
- Alternative Metal / Punk
- Classic Rock
- Metal
- Roots Rock

**Spoken**

**Figure 8.6:** Popular Music genres included in the Codaich audio dataset.

**Classical**
- 20th Century Classical
- Baroque
- Classical
- Renaissance & Medieval
- Romantic

**Jazz**
- Acid Jazz
- Avant-Garde Jazz
- Bebop
- Cool Jazz
- Dixieland
- Hard Bop
- Latin Jazz
- Post-Bop
- Soul Jazz
- Swing

**World**
- African
- Americas
- Arabic
- Asian
- Calypso
- Celtic
- Chanson
- Cuban
- European
- Flamenco
- Fusion
- Gypsy
- Indian
- Klezmer
- Latin American
- Mixed Traditional
- Tango
- U.S. Traditional

**Figure 8.7:** Classical, Jazz and World Music genres included in the Codaich audio dataset.

## 8.6.3 OMEN: A framework for overcoming copyright limitations

As discussed in Section 8.2, legal restrictions on distributing audio data pose a serious obstacle to the sharing of musical datasets between research labs. Such limitations can greatly increases the work needed to build new systems, since each lab must assemble its own training and testing dataset. Legal obstacles to dataset distribution also make it much more difficult to comparatively evaluate different MIR systems, since doing so properly requires that they be trained and tested on the same datasets.

The *On-Demand Metadata Extraction Network* (OMEN) (Fujinaga and McEnnis 2006; McEnnis 2006; McEnnis, McKay and Fujinaga 2006b) offers a framework that can provide one possible solution to these problems. OMEN, which was implemented entirely by Daniel McEnnis, essentially proposes the notion of a networked infrastructure that allows researchers to submit custom feature extraction requests to sites that have local legal access to the particular digitized musical recordings that are desired. Examples of such sites include libraries with digitized music collections and MIR research centres. These sites can then locally extract the requested features and return the resulting feature values to the user. This can all be done without explicitly violating copyright laws, since users are never provided with access to the audio samples themselves.

Since feature values and metadata are essentially what many MIR researchers are interested in, publicly distributing this data rather than the music itself is a useful way of legally circumventing copyright limitations. The motivation behind copyright laws is not to hinder research, but rather to protect intellectual property and prevent pirating. Discussions with legal experts at past ISMIR conferences have indicated that there is likely no legal obstacle to the distribution of information extracted from recordings, as long as such information cannot be used to reconstruct the music itself with a listenable degree of fidelity. This means that many of the features that are typically extracted form music in MIR research can in fact be publicly distributed, even when the music itself cannot. Even features such as MFCCs can only be used to reconstruct a very poor facsimile of the originals under typical feature parameterizations.

It is important to note that simply extracting and publicly posting stock features would be an insufficiently flexible approach, since an important part of MIR research involves developing new and specialized features. Furthermore, different researchers will want features extracted using different feature and pre-processing parameters, such as window size, window overlap, downsampling, amplitude normalization, etc.

OMEN addresses these issues through integration with the jAudio audio feature extractor (see Chapter 3). jAudio not only allows users to customize the parameters of the features that are extracted, but also allows new features to be added remotely as plug-ins without the need for recompiling jAudio itself. Each site providing access to music via OMEN would therefore have copies of jAudio installed locally, and would allow users to set extraction parameters and add features as needed as part of their feature extraction requests.

Extracting features from audio can be a computationally intensive task. The load on any one OMEN site is diminished by the likelihood that multiple sites will hold copies of at least some recordings from which features are to be extracted, so extraction tasks can be divided up between sites. Additionally, libraries in particular have the special advantage of owning large networks of computers around the world that go unused outside of opening hours, and that typically use only a small percentage of their processing capability even during business hours. Such networks of underutilized computers could be taken advantage of using grid computing to speed up feature extraction.

OMEN's architecture consists of three different types of *nodes,* each of which communicates with other nodes using web services:

- **Master node:** A centralized server that coordinates all OMEN feature extraction requests. It maintains a regularly updated directory of which recordings are available at which sites and has searchable access to their metadata. The master node also provides users who wish to extract features with an interface allowing them to specify and parameterize the features that they wish to extract and choose the recordings that they want to extract them from, but hides the actual locations of the recordings.

- **Library nodes:** The coordinating server at each site that has legal access to audio samples. Library nodes receive feature extraction requests from the master node, coordinate the actual extraction of features by their subordinate worker nodes and communicate extracted feature values to users via the master node. Each library node distributes recordings from which features are to be extracted among worker nodes as appropriate.

- **Worker nodes:** The individual computer or computers that perform the actual work of feature extraction at each site that has legal access to audio samples. This is done by running jAudio to extract features from the audio recordings that are assigned by a library node.

OMEN provides a framework that can potentially be used to effectively make Codaich available to all MIR researchers. However, although McEnnis has informally tested OMEN on Codaich using resources available at McGill University, there are a number of efficiency and security issues that remain to be addressed before Codaich can be made available to the public via OMEN.

### 8.6.4 Informal observations on common metadata errors

As mentioned previously, Codaich's metadata was originally seeded with information extracted from the contents of ID3 tags and from the Gracenote database. This seed metadata was then all checked for errors, both manually and using jMusicMetaManager, a process that quickly revealed certain common types of errors in the seed metadata. Although a systematic study of these errors was beyond the scope of the work being done, informal records were kept for the purpose of determining if additional kinds of error checking needed to be added to jMusicMetaManger. Some of the most common errors are briefly summarized[248] here in order to serve as a potential starting point for more formal studies in the future, and as a list of particularly common problems errors to be cautious of when manually correcting metadata found in the "wild":

- Inappropriate spaces at the beginning of strings.
- Inappropriate use of all caps.
- Track titles missing entirely, or replaced by entries such as *Track 01* rather than actual recording titles.

[248] Additional types of common errors are implicit in the jMusicMetaManager error detection options.

- Track numbers included in the beginning of otherwise correctly annotated recording titles (e.g., *07 Mercy*).
- Titles cut off after an insufficient number of characters.
- Placement of artist names in the Title field as one combined string with titles instead of separately in the Artist field.
- Inclusion of guest artist names in the Title field, even when the Artist field specifies the name of the main artist(s). This is particularly common in Hip-Hop and R&B.
- Use of *Various Artists* or other such labels in the Artist field of compilation albums.
- Particularly unreliable composer annotations in Popular music.
- Use of composers' names in the Artist field of Classical music.
- Variable spelling of non-English names, particularly with respect to the use of non-English accents and characters.
- General confusion over whether the Year field should specify the year of recording or the year of album release (e.g., in compilations or posthumous releases).
- Disc numbers often included in the Album field rather than in the Disc Number field.
- Track numbers often missing.
- Great inconsistency in the Genre field. Interestingly, many recordings that are obviously not Blues recordings are annotated with the Blues label, likely because it corresponds to the first ID3 code and is often the default choice supplied by audio encoding software.
- Accidental and non-systematic spelling errors of all kinds.

## 8.7 The SAC dataset: Combining symbolic, audio and cultural musical data

The SAC (Symbolic Audio Cultural) dataset (McKay and Fujinaga 2008) was assembled in order to provide matching symbolic, audio and cultural data that could be analyzed and studied together, such as in the experiments described in Section 9.4. SAC consists of 250 MIDI recordings, 250 matching MP3s, and metadata for each of these recordings (e.g., Title, Artist, Genre, etc.). This metadata is stored in an iTunes XML file that can be parsed by jWebMiner in order to extract cultural features from the web.

The combination of the MIDI files, MP3 files and the iTunes XML file therefore provide sufficient information for extracting features from matching symbolic, audio and cultural sources for each of the 250 different pieces of music. jMIRUtilities, described in Section 8.8, can be used to facilitate the combination of such different feature types, each extracted using jSymbolic, jAudio and jWebMiner, respectively.

It should be noted that the MIDI and audio recordings were acquired separately, rather than simply synthesizing the audio files based on the MIDI recordings. Although this made acquiring the dataset significantly more difficult and time consuming, it was considered necessary in order to truly test the value of combining symbolic and audio data, since audio generated from MIDI by its nature does not include any additional information other then the very limited data encapsulated by synthesis algorithms.

The Bodhidharma MIDI dataset and Codaich were using as starting points for building SAC. Unfortunately, there was insufficient overlap between the recordings in these two source datasets to build SAC to an acceptable size using a reasonable genre ontology. It was therefore necessary to acquire some new recordings in order to attain a sufficient amount of music in each of the desired genres. The process of finding MIDI files of a sufficient quality that met the constraints of the problem was a very time consuming task, and it is for this reason that SAC currently contains only 250 pieces of music.

SAC is divided into 10 genres, with 25 pieces of music per genre. These 10 genres consist of 5 pairs of similar genres, as shown in Figure 8.8. This arrangement has the advantage of making it possible to perform both 10-class and 5-class genre classification experiments simply by combining the instances in each pair of related genres into one class. It is therefore possible to experimentally evaluate how good a classification system is at distinguishing between relatively dissimilar classes as well as relatively similar classes. An additional advantage is that it becomes possible to measure an indication of how serious misclassification errors are in 10-class experiments by examining how many misclassifications are in an instance's partner genre compared to how many are in one of the other 8 genres. These two characteristics of SAC's genre ontology were taken advantage of in the experiments described in Section 9.4.

The particular total of 10 genres was chosen for the purpose of making rough comparisons with the performance of the many previously published genre classification systems that have used genre ontologies of a similar size, probably because of the influential early work of Tzanetakis and Cook (2002). Also, the most recent audio genre classification MIREX evaluation[249] involved 10 genres, and the most recent symbolic genre classification MIREX evaluation[250] involved 9. As emphasized in Section 8.2, however, in the long term it will be necessary to significantly expand the number of genres in SAC in order to perform evaluations whose results are meaningful in the context of practical applications.

The basic methodologies used in devising SAC's particular genre ontology, choosing appropriate pieces and labelling them were based on those used in constructing the Bodhidharma MIDI dataset, as described in Sections 8.5.2 and 8.5.3, with the obvious difference that it was possible to take advantage of some of the recordings already in the Bodhidharma dataset and Codaich. As was the case with these two source datasets, steps were taken to make SAC particularly difficult and realistic, such as including multiple versions of the same pieces in different genres and including examples of different pieces in different genres by the same artist.

Those wishing to acquire more specific details on the SAC dataset should contact Cory McKay at cory.mckay@mail.mcgill.ca.

[249] www.music-ir.org/mirex/2009/index.php/Audio_Genre_Classification_%28Mixed_Set%29_Results
[250] www.music-ir.org/evaluation/mirex-results/sym-genre/index.html

| | | |
|---|---|---|
| **Blues** | **Classical** | **Jazz** |
| Modern Blues | Baroque | Bop |
| Traditional Blues | Romantic | Swing |
| **Rap** | **Rock** | |
| Hardcore Rap | Alternative Rock | |
| Pop Rap | Metal | |

**Figure 8.8:** The genres of music found in the SAC dataset.

## *8.8 jMIRUtilities: Tools for labelling, linking and pre-processing musical data*

jMIRUtilities is a set of tools for performing miscellaneous tasks relating to preparing musical data for processing by the jMIR components. Although there are plans to expand jMIRUtilities to have a much larger range of functionality, at the time of this writing it may only be used to perform the following tasks:

- **Manual instance labelling:** A simple GUI is provided for quickly batch labelling sets of audio, MIDI or other files and generating a corresponding ACE XML Instance Label file (see Chapter 7).
- **Automatic instance labelling:** An ACE XML Instance Label file may be automatically generated based on a tab-delimited text file containing labelled instances. This is convenient for model classifications that are stored in a Microsoft Excel table, for example, which may easily be exported to such a tab-delimited text files.
- **Extract labels from iTunes files:** An iTunes XML file may be parsed and a delimited text file generated that lists the File Path, Recording Title, Artist Name, Composer Name, Album Title and/or Genre Names for each recording in the file, as requested by the user. This can be a useful pre-processing step for generating ACE XML Instance Label files from iTunes XML files via jMIRUtilities' automatic instance labelling functionality.
- **Feature merging:** ACE XML Feature Value files (see Chapter 7) that contain different features for the same instances may be merged into a single ACE XML

Feature Value file. ACE XML Feature Description files that correspond to these Feature Value files may also be merged. This might be used, for example, to combine feature values extracted by jAudio with feature values extracted by jWebMiner for the same pieces of music.

jMIRUtilities is, like all jMIR components, implemented in Java, open-source and free. The development and expansion of jMIRUtilities is an ongoing project, but a prototype version may be acquired by contacting Cory McKay at cory.mckay@mail.mcgill.ca.

## *8.9 Summary of original contributions*

### 8.9.1 Datasets

Section 8.2 presents an original theoretical discussion of essential issues that should be considered when designing, building and labelling musical datasets that are to be used as ground-truth in MIR research. Among other important issues, this discussion emphasises in particular the design of effective and realistic class ontologies. Many of the issues raised in this section are vital, considering the poor quality of the datasets that are used in much of the published MIR research, and given the essential role that ground-truth plays in the training and testing of classification algorithms. Section 8.2.4 consists of an original list of proposed guidelines to follow when building MIR research datasets.

The Bodhidharma MIDI dataset, Codaich and SAC are presented as three different original datasets designed for general use in MIR research. They are each alternative prototype implementations of many of the music research dataset design principles proposed in Section 8.2.4. These three datasets are each designed to be particularly difficult to classify in order to increase the realism of evaluations experiments that are performed with them.

Codaich is a large, carefully labelled and diverse collection of MP3 recordings. The Bodhidharma MIDI dataset is the largest and most diverse research collection of MIDI files known to the author. The SAC dataset combines matched audio recordings, MIDI recordings and cultural metadata for the purpose of performing research on combining features extracted from audio, symbolic and cultural sources.

jMIRUtilities offers a set of miscellaneous tools for facilitating the labelling and merging of musical data.

### 8.9.2 jMusicMetaManager

jMusicMetaManager is an original powerful automatic music metadata error detection tool. Although there are a number of services that attempt to correct metadata by identifying recordings by matching them to their own databases and then replacing the recordings' metadata with the metadata stored in these databases, the metadata on these databases itself tends to be error-prone and inconsistent. jMusicMetaManager is fully self-sufficient and does not rely on any external sources of information that could themselves be erroneous.

jMusicMetaManager includes a number of novel error-detection methodologies and implementation algorithms. These include original variants of classic edit distance thresholding, the use of specialized find/replace operations for reducing potential edit distances in ways that are particular to music, novel reordered word subset strategies and the use of false-positive filters specifically designed for music. The merge-based algorithms described in Section 8.4.4 that provide the central error-detection infrastructure are also themselves entirely novel.

Finally, many of the profiling reports generated by jMusicMetaManager for organizing music collections and publishing information about them present such information in novel ways.

## *8.10 Future research*

### 8.10.1 Datasets

This chapter includes a reasonably extensive theoretical discussion on constructing MIR research datasets, including a number of general guidelines. There is a need for empirical studies on many of the issues that are raised here, such as on the comparative reliability of ground-truth labels derived from different types of sources or on the relative incidences of different types of metadata errors. The data resulting from such studies would be very helpful in ultimately formulating more concrete and specific guidelines for building musical datasets.

Although the Bodhidharma MIDI dataset, Codaich and SAC are very high-quality datasets relative to the norm in MIR research, there is still certainly potential for further improvement. For example, although Codaich is fairly large and is constantly growing, both SAC and the Bodhidharma dataset remain much smaller. A priority in the future will thus be to increases their size and, particularly in the case of SAC, diversity.

As noted in Section 8.5, collecting research-appropriate MIDI files can be a very work intensive prospect. Efforts will therefore be made in the future to study the viability of generating MIDI files by automatically transcribing audio files. Although transcription technology has traditionally been quite error-prone when dealing with polyphonic audio, improvements continue to be made. There have been some promising recent developments, such as Celemony Software's Melodyne *Direct Note Access*[251] technology, although the performance of this software remains to be publicly verified. In any case, if reliable transcription technology were to become available then the Bodhidharma dataset and SAC could easily be dramatically expanded to include all of the recordings in Codaich.

There are also certain improvements that can be made to Codaich's metadata. Specifically, Codaich's Composer field remains poorly annotated for non-Classical recordings, and the Recording Year field has not yet been annotated for all recordings. Some of the improvements to jMusicMetaManager suggested in Section 8.10.2 could also help to detect potential metadata errors in Codaich that have not yet been detected.

The addition of more metadata fields for at least some of the recordings would help to extend the range of MIR research areas to which Codaich could be applied. Such additions could include general labels such as mood or listening scenario, for example, as well as more content-oriented metadata such as sectional divisions or note onsets.

Relating to this, the integration of multiple parallel ontologies into Codaich, as discussed in Section 8.2.2, is an additional area for further research. The development of web-mining software for automatically and reliably acquiring both ontologies themselves and the labels to apply to individual instances would be very useful in this respect.

Readying OMEN for public release is another priority. Although OMEN's developer, Daniel McEnnis, is currently involved in other projects, work on it could be continued by

[251] http://www.celemony.com/cms/

others. Particular priorities include testing the framework using truly distributed remote locations and the incorporation of security to prevent features from being extracted that could be used to reconstruct music at a listenable level of quality.

There are also plans to add a significant amount of additional functionality to jMIRUtilities so that it can be used for a diverse range of data pre- and post-processing tasks.

### 8.10.2 jMusicMetaManager

Perhaps the most useful type of functionality that could be added to jMusicMetaManager would be the ability to automatically apply corrections to iTunes XML files and MP3 ID3 tags. This is not as simple a task as it might sound, however, as an interface would be needed to allow users to approve changes before they are made so that false positives do not corrupt metadata, as well as to resolve cases where there might be ambiguity as to what the appropriate correction would be to a detected error.

One intriguing way to address such issues would be to have jMusicMetaManager access audio fingerprinting services such as MusicBrainz. This would make it easy to acquire good recommendations for the correct metadata values to use when errors are detected. This would also have the further advantage of providing default tags for recordings that lack metadata labels in the original collection.

The use of such fingerprinting services would also provide an entirely new way of detecting errors, namely by simply comparing metadata entries in the dataset being analyzed with the tags suggested by MusicBrainz for each recording. There are a number of potential problems with this approach, such as metadata errors in the MusicBrainz database itself, rare recordings that may not be in the MusicBrainz dataset, erroneous fingerprinting identifications and dramatically increased processing times due to the necessity of processing audio to extract fingerprints. Nonetheless, this approach does also have significant potential advantages, and certainly warrants investigation.

There are also other resources that could be consulted by jMusicMetaManager to derive model tags that could be used to detect additional errors or propose appropriate corrections. These include simple dictionaries, data mined from on-line sources like the All Music Guide and library authority controls (DiLauro 2001), such as that provided by the U.S. Library of Congress. Potential metadata corrections could also be ranked by comparing hit counts for different candidates using search engines such as Google.

There are also additional new types of music collection profiling reports that could usefully be added to jMusicMetaManager. Further variants of the error-checking algorithms already in use could also be added, such as new find/replace transformations. For example, jMusicMetaManager does not yet transform diacritical markings like umlauts. jMusicMetaManager is also not yet capable of correctly dealing with non-European alphabets, such as Hiragana. A number of default error detection parameterizations could also be distributed with the software, each suited to different types of musical datasets.

Improvements could also be made to some of jMusicMetaManager's low-level processing. A number of optimizations could decrease processing times, for example. It would also be preferable to use a better ID3 parser if one becomes available, since all of the open-source ones that were available at the time of jMusicMetaManager's implementation were unable to parse some of the MP3 files in Codaich.

Another useful addition would be to make jMusicMetaManager able to parse metadata from file formats other than MP3s. Although such metadata can currently be accessed by jMusicMetaManager indirectly by parsing iTunes XML files, the software should ideally also be able to parse formats such as AAC recordings directly.

Also, the iTunes software itself has a number of limitations with respect to the kinds of metadata that it can be used to edit. These limitations include a limited number of fields, only one genre allowed per recording,[252] no functionality for representing ontological structuring and a lack of segmented annotation of recordings, all types of information which may be freely expressed in ACE XML files. A useful addition to jMusicMetaManager in the future would therefore be a GUI that could be used to edit metadata in ways that are more expressive and structured than permitted by iTunes.

A final priority is the design a sleeker jMusicMetaManager API so that researchers can access its functionality from their own software more easily. Although jMusicMetaManager is already open-source and is implemented using well-documented

[252] jMusicManager bypasses the single genre limitation by using the plus sign as a delimiting code that permits multiple genres to be stored in a single string.

and modular code, the architectural emphasis was originally on providing functionality via a GUI rather than through third-party software.

# 9. Evaluations and experiments

## *9.1 Overview of evaluations*

This chapter describes several sets of evaluations that were conducted using the jMIR software suite. These experiments were designed with the intention of verifying the suite's ability to effectively perform music classification tasks, both as separate modules and as units operating together linked by the ACE XML file formats. Comparative experiments were also performed investigating the potential for increasing classification performance by combining features extracted from audio, symbolic and cultural information sources.

It is important to note that all of the experiments described in this chapter were performed without any special fine tuning or custom modifying of the jMIR components. Such modifications were avoided in order to more accurately simulate use cases where users with limited technical backgrounds might wish to use jMIR simply as is. It is likely that improved experimental results could have been attained with task-specific parameterizations, but such an approach would have been inconsistent with this chapter's goal of attempting to evaluate jMIR's general applicability to MIR classification research.

The experiments described in Section 9.2 were performed in the early stages of jMIR's development, and involved applying jAudio and ACE to a number of different MIR and general classification tasks in order to verify that good results could be attained. Results were compared to results on the same datasets published by other researchers when possible.

Section 9.3 describes the submission of the Bodhidharma system (McKay 2004) to the MIREX Symbolic Genre Classification competition for comparison with systems made by other research groups. Bodhidharma is the original framework that jSymbolic is built upon.

A set of exploratory experiments with the release version of jMIR are described in Section 9.4. These experiments investigated the classification utility of combining features extracted from audio, symbolic and cultural sources of musical information by performing a series of genre classification experiments using all seven possible combinations and subsets of the three corresponding types of features. The results are presented and discussed in detail. These experiments also served as validation of the ability of the different jMIR components to be effectively used together.

## 9.2 Preliminary experiments

Four sets of experiments were performed during the early development of the jMIR software in order to roughly evaluate the performance of ACE and jAudio. These experiments were designed simply to verify that jMIR could perform at least as well as other published research when applied to MIR-related tasks. The results of the first three sets of experiments are shown in Table 9.1.

| Experiment | Previous Research | Success Rate (Previous Research) | Success Rate (jMIR) |
|---|---|---|---|
| Drum stroke | Tindale et al. 2004 | 94.9 | 96.3 |
| Beatboxing | -- | -- | 95.6 |
| Speech/music | Scheirer & Slaney 1997 | 94 | 98 |

**Table 9.1:** jMIR's classification success rates on various MIR datasets. All values are percentages, and they are all averages calculated across cross-validation folds.

The first experiment consisted of repeating Tindale et al.'s (2004) snare drum stroke identification experiment, using Tindale's data but replacing the feature extraction and classification software that he used with jAudio and ACE, respectively. This dataset encompassed seven drum stroke classes, namely centre, half centre, halfway, halfedge, edge, rimshot and brush. jMIR achieved a success rate of 96.3%, as compared to Tindale et al.'s best rate of 94.9%, a proportional reduction in error rate of 27.5%.

The second experiment involved using jMIR to perform five-class beat-box classification (Sinyor et al. 2005). The five classes were kick, open, closed, k-snare and p-snare. Although no other research has been published based on this data that could be used for comparison, jMIR achieved a success rate of 95.6%, a value approaching a level of accuracy usable in performance scenarios.[253]

The third experiment consisted of two-class speech/music discrimination based on Scheirer and Slaney's classic data set (1997). jMIR achieved a success rate of 98% at this

[253] jMIR does not currently necessarily operate in realtime, however.

task, a proportional decrease of 67% in error rate compared to Scheirer and Slaney's success rate of 94%.

The fourth set of experiments involved running ACE on ten UCI feature sets (Blake and Merz 1998) from a variety of research domains. This served as a test of ACE's meta learning approach. The results, along with a baseline taken from previously published machine learning research on some of the same datasets (Kotsiantis and Pintelas 2004), are shown in Table 9.2 and Figure 9.1.

It can be seen that ACE performed very well, particularly given the difficulty of some of these feature sets. This is emphasized by ACE's excellent performance relative to the results published by Kotsiantis and Pintelas, which were themselves shown by the authors to be better than a wide variety of alternative algorithms. Although statistical uncertainty makes it impossible to claim that ACE's results are inherently superior, it does show that ACE can achieve results as good as or better than relatively recent sophisticated state-of-the-art algorithms.

Furthermore, in this last experiment, ACE was forced to restrict each of its learning algorithms to one minute or less for both training and testing on a now relatively obsolete desktop computer (a 2.8 GHz single core Pentium 4). The time limit was imposed in order to investigate ACE's ability to rapidly evaluate a wide variety of classifiers. Although even higher success rates could likely have been achieved with more training time, the performance achieved by ACE in this limited time demonstrates its efficiency in exploratory research.

Table 9.2 is also revealing in that it demonstrates how a variety of different classifiers can perform best given a variety of feature sets. Furthermore, the AdaBoost algorithm was selected by ACE four times out of ten, demonstrating the efficacy of the ensemble classification approaches that are incorporated into ACE.

The results of all four of these sets of experiments, which demonstrated that similar or better than published results could be achieved by jMIR, were encouraging, and served as the initial validation justifying the further development of the software. It is important to re-emphasize that these results were all achieved without any manual tuning applied based on each particular application under consideration, as is consistent with jMIR's

| UCI Data Set | ACE's Selected Classifier | Kotsiantis' Success Rate | ACE's Success Rate |
|---|---|---|---|
| anneal | AdaBoost | -- | 99.6 |
| audiology | AdaBoost | -- | 85.0 |
| autos | AdaBoost | 81.7 | 86.3 |
| balance scale | Naïve Bayes | -- | 91.4 |
| diabetes | Naïve Bayes | 76.6 | 78.0 |
| ionosphere | AdaBoost | 90.7 | 94.3 |
| iris | FF Neural Net | 95.6 | 97.3 |
| labor | k-NN | 93.4 | 93.0 |
| vote | Decision Tree | 96.2 | 96.3 |
| zoo | Decision Tree | -- | 97.0 |

**Table 9.2:** ACE's classification success rate on ten UCI feature sets compared to a published baseline (Kotsiantis and Pintelas 2004). All values are percentages, and they are all averages calculated across cross-validation folds.

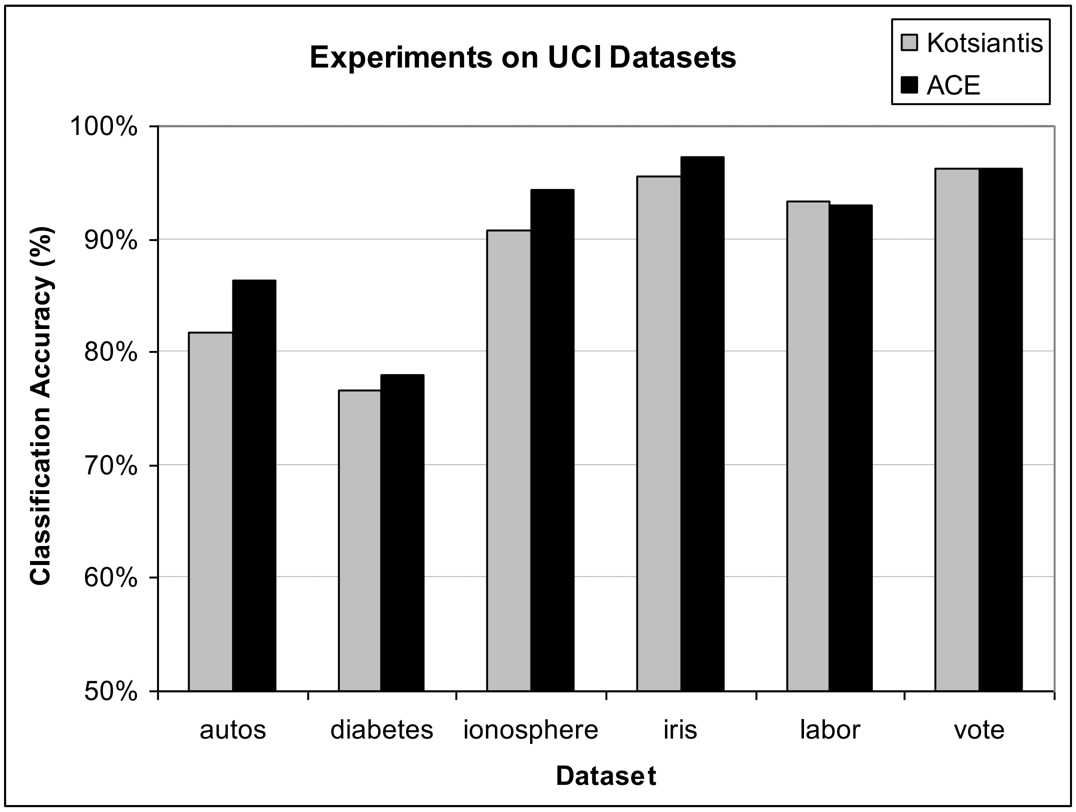


**Figure 9.1:** Comparison of ACE's performance with that of Kotsiantis and Pintelas (2004) on six UCI feature sets. Based on the results described in Table 9.2.

design priority of providing software that can be applied immediately to MIR research problems without the need for user customization, unless desired.

## 9.3 MIREX evaluation of Bodhidharma

The Music Information Retrieval Evaluation eXchange (MIREX) (Downie 2006) is a yearly competition associated with the ISMIR conference. It provides an opportunity for a variety of MIR algorithms and approaches to be independently evaluated and compared using the same data sets. Evaluation areas (e.g., artist identification, audio onset detection, beat tracking, etc.) are proposed by the community, and evaluations are performed independently and objectively at the International Music Information Retrieval Systems Evaluation Laboratory (IMIRSEL).

The 2005 edition of MIREX included a Symbolic Genre Classification category, and the Bodhidharma system was submitted to it for evaluation. Bodhidharma (McKay 2004) consists of a symbolic feature extraction system combined with an ensemble machine learning classification system. The feature library used by jSymbolic was originally identical to the feature library implemented in Bodhidharma, although the jSymbolic library has since been improved. Bodhidharma also served as the basic framework that jSymbolic was built on, although Bodhidharma's machine learning component was excluded. The evaluation of Bodhidharma was therefore equivalent to an evaluation of the prototype version of jSymbolic combined with Bodhidharma's machine learning algorithms.

The 2005 MIREX Symbolic Genre Classification competition involved evaluations on two hierarchical genre taxonomies, one consisting of three parent classes and nine leaf classes, and the second consisting of nine parent classes and thirty-eight unique leaf classes. Each recording was assigned exactly one leaf genre label.

A separate experiment was performed for each of the two taxonomies, with each experiment using stratified cross-validation. Two different classification success rates were calculated for each experiment:

- **Raw accuracy:** Each system was given a full point for each correct leaf genre and zero points for each incorrect classification. This evaluated the ability of the systems to make precise classifications.

- **Hierarchical accuracy:** Each system was given a full point for each correct leaf genre. Partial points were given if the selected leaf genre was incorrect, but was in a correct branch of the taxonomy tree. This approach therefore included a measure of whether the systems made minor mistakes or major mistakes when they were incorrect.

A total of five systems were submitted to IMIRSEL for MIREX evaluation. The results are shown in Tables 9.3 and in Figure 9.2. The detailed results are reported at http://www.music-ir.org/evaluation/mirex-results/sym-genre/index.html.

These results demonstrate excellent performance by the Bodhidharma system and, by extension, jSymbolic. Bodhidharma placed first in all four evaluation categories. Although some of the other submissions achieved success rates that were statistically comparable to those of Bodhidharma on the 38-class taxonomy, Bodhidharma performed significantly better than any of the other systems on the 9-class taxonomy, with a raw accuracy 12.4% higher than the second place system and a hierarchical accuracy 8.4% higher.

Although Bodhidharma's results on the small taxonomy were arguably good enough for some practical application, none of the submissions achieved practically viable results on the larger taxonomy. So, while it is certainly encouraging that results far better than chance[254] were attained by all of the systems, it was clear that there was still much work to be done before automatic symbolic genre classification could be used in a real-world context involving large taxonomies.

There has not been another Symbolic Genre Classification Evaluation at MIREX since 2005, so it is difficult to say for certain if superior results have since been attained by an alternative system. However, based on a review of the literature, the best symbolic genre classification results to date appear to have been achieved by DeCoro, Barutcuoglu and Fiebrink (2007). Using the same dataset as the one used in MIREX 2005, they achieved a raw classification accuracy of 60.1% on the 38-class taxonomy, an improvement of 14.0% over Bodhidharma's performance. This improvement was achieved by using a Bayesian Aggregation approach to perform the classification as an

[254] Random classification would yield averages of 11.1% on the 9-class taxonomy and 2.6% on the 38-class taxonomy.

|  | Raw Accuracy 9-Class | Hierarchical Accuracy 9-Class | Raw Accuracy 38-Class | Hierarchical Accuracy 38-Class |
|---|---|---|---|---|
| **Bodhidharma** | 84.4 | 90.0 | 46.1 | 64.3 |
| **Basili et al. (NB)** | 72.0 | 81.6 | 45.0 | 62.6 |
| **Li** | 72.0 | 80.2 | 41.0 | 57.6 |
| **Basili et al. (J48)** | 65.3 | 76.7 | 39.8 | 54.9 |
| **Ponce de Leon** | 37.8 | 50.7 | 15.3 | 24.9 |

**Table 9.3:** MIREX 2005 Symbolic Genre Classification results. All values are percentages, and they are all averages calculated across cross-validation folds.

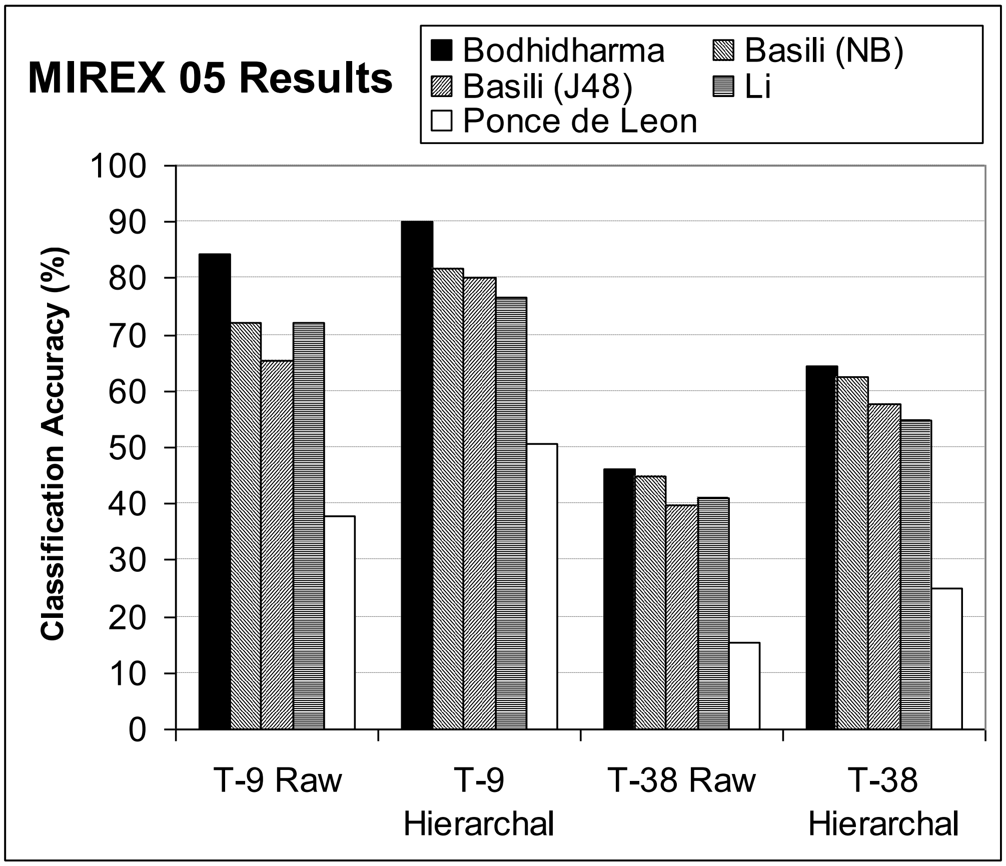


**Figure 9.2:** MIREX 2005 Symbolic Genre Classification results.

alternative to the neural net and k-nearest neighbour ensemble approach used by Bodhidharma. What is encouraging from the perspective of jMIR is that DeCoro, Barutcuoglu and Fiebrink used Bodhidharma/jSymbolic's feature library as the input to their classification algorithms.

The combination of the MIREX results and the work of DeCoro, Barutcuoglu and Fiebrink demonstrate that jSymbolic's feature library has achieved higher symbolic genre classification success rates than any other symbolic feature libraries,[255] both at the most recent MIREX Symbolic Genre Classification Evaluation and in the general research published to date.

## 9.4 Experiments on combining feature types

### 9.4.1 Goals and overview

The final set of experiments consisted of using the release version of jMIR to classify music by genre using features extracted from audio, symbolic and cultural sources. Classifications were performed using each feature type individually, all three possible pairs of feature types and all three feature types together. These experiments were performed with three goals in mind:

- To verify the ability of the mature jMIR components to effectively perform music classification.
- To verify the intercompatibility of the jMIR components.
- To perform exploratory research investigating the potential benefits of combining features extracted from audio, symbolic and cultural sources.

In the past, MIR research has tended to focus on studying either audio, symbolic or cultural sources of musical information. In recent years, as described in Section 9.4.2, MIR researchers have increasingly begun to study these sources of information in combination, with a particular emphasis on research combining audio and cultural sources

[255] Although tests were not performed at MIREX isolating the feature extraction and machine learning libraries so that they could be independently evaluated.

of data. However, there has yet to be a systematic study investigating the relative levels of automatic classification efficacy corresponding to all three feature types operating together, or of each of the seven possible combinations of the three feature types.

The experiments described in this section[256] were therefore devised to fill this gap. In essence, this research is intended to address the degree of orthogonal independence of the three feature types. If the feature orthogonality is high, then significant performance boosts can potentially be attained by combining feature types. If it is low, and there is therefore significant redundancy between the feature types, then combining the different sources of musical information may be wasted effort.

As noted above, this investigation was performed via sets of automatic genre classification experiments. Genre classification in particular was chosen because it is a complex and difficult task that combines diverse musical variables and requires classification systems to successfully deal with problematic labelling and ground-truth issues (McKay and Fujinaga 2006b), thereby serving as an appropriately difficult test case. The experiments could just as easily have been performed using other types of classification, however, such as mood or artist classification. The essential question being investigated remains the degree of utility attained by combining features extracted from the three types of musical data.

Before proceeding to the details of the experiment, it is appropriate to first briefly discuss the three types of musical data. Features extracted from all three can arguably provide valuable information for use in music classification and similarity research. Audio is clearly useful because it corresponds to the essential way that music is consumed, and cultural data external to musical content is musicologically well-known to be highly influential on our interpretations and experiences of music (e.g., Fabbri 1981).

Symbolic data has recently been receiving less attention from MIR researchers than it did in the past, with the obvious exception of music theoretical researchers. The value of symbolic data should not be overlooked, however, as much of the information relating to high-level musical abstractions that can be relatively easily extracted from such formats is

[256] A condensed version of these experiments have also been published at the ISMIR conference (McKay and Fujinaga 2008).

currently poorly encapsulated by the types of features that are typically extracted from audio, which tend to focus primarily on timbral information

Symbolic formats can thus, at the very least, be a powerful representational tool in automatic music classification. This characteristic will become increasingly valuable as polyphonic audio to symbolic transcription algorithms continue to improve. The end result of successful transcription technology may well be that symbolic formats in essence become important intermediate data structures. Even though such transcription technologies are still error-prone, it has been found (e.g., Lidy et al. 2007) that classification systems can be relatively robust to such errors. Also, the advance material promoting Celemony Software's Melodyne *Direct Note Access*[257] technology seems to indicate that fairly high-quality audio-to-symbolic transcription will soon become available, although the performance of the technology remains to be publicly verified.

The usefulness of combining features extracted from all three sources of musical information therefore seems intuitively apparent. However, as stated above, this remained to be experimentally verified prior to the experiments described in this section. These experiments also provided an excellent test scenario for combining the jAudio, jSymbolic, jWebMiner and ACE jMIR components, as well as validating the intercompatibility of the ACE XML file formats, thereby meeting all three goals outlined at the beginning of this section.

### 9.4.2 Related research

There has been a significant amount of research on combining audio and cultural data. Whitman and Smaragdis (2002) and Baumann, Klüter and Norlien (2002) performed some particularly important early work on combining audio features with cultural data mined from the web, and achieved substantial performance gains when doing so. Ellis and his colleagues (2002) also combined audio and cultural data, but with an emphasis on acquiring ground-truth. Dhanaraj and Logan (2005) took a more content-based approach by combining information extracted from lyrics and audio. Knees and his colleagues (2006) combined audio and cultural data for the purpose of generating a music browsing space, as did Pampalk and Goto (2007). Aucouturier and Pachet (2007) combined audio

[257] http://www.celemony.com/cms/

features with cultural metadata in order to achieve improved genre classification performance. Research has also been done on using audio data to generate or make correlations with cultural labels, which can in turn improve other kinds of classification (e.g., Eck, Bertin-Mahieux and Lamere 2007; Reed and Lee 2007).

There has been much less work on combining symbolic data with audio data. One of the exceptions is the outstanding work of Lidy and his colleagues (2007), who reported improvements in three different sets of experiments when audio and symbolic data were combined, compared to when only audio data was used.

Much of the research on combining audio and symbolic data has simply involved *MIDIstration,* which is to say the synthesis of audio files based on MIDI files. Although this can be useful for some purposes, it is nonetheless much less powerful than separately acquired audio and symbolic data, since the only additional information attained via MIDIstration is the timbral information stored in the synthesis algorithms used. MIDIstration is in general less useful than audio to symbolic transcription, even though the transcription process arguably does not add any information that is not already in the audio signal, since the transcription process has the advantage of making high-level features much more accessible than they are in raw audio.

To the best knowledge of the author, no previous research has been performed on combining symbolic and cultural features or on combining all three.

The literature on automatic genre classification is far too large to cite with any completeness here, but the work of McKay and Fujinaga (2006) and McKay (2004) include citations of some of the most influential research in the field as well as discussions of essential issues.

### 9.4.3 Experimental procedure

The experiments were performed using the SAC (Symbolic Audio Cultural) dataset, which is described in detail in Section 8.7. This dataset was assembled for the specific purpose of performing these experiments, although it can of course be used for other research as well. The most pertinent aspects of SAC are reviewed in the following paragraphs.

The SAC dataset consists of 250 MP3 files, 250 MIDI files matched to each audio file and cultural metadata[258] associated with each piece. Each matching MIDI file was downloaded from the Internet independently, and was not transcribed from the audio, nor was the audio derived from the MIDI via MIDIfication. Although this made acquiring the dataset significantly more difficult and time consuming, it was considered necessary in order to truly test the value of combining symbolic and audio data, since audio generated from MIDI by its nature does not include any additional information other then the very limited data encapsulated by the synthesis algorithms, as discussed is Section 9.4.2.

The pieces of music making up SAC are divided into ten different genres, and each piece is assigned a ground-truth genre label. These ten genres consist of five pairs of similar genres, as shown in Figure 9.3. This arrangement has the advantage of making it possible to perform both 10-class and 5-class genre classification experiments simply by combining the instances in each pair of related genres into one class.

This arrangement also makes it possible to experimentally evaluate how good a classification system is at distinguishing between relatively dissimilar classes as well as relatively similar classes. A related advantage is that it also becomes possible to measure an indication of how serious misclassification errors are in 10-class experiments by examining how many misclassifications are in an instance's partner genre compared to how many are in one of the other eight genres.

| **Blues** | **Classical** | **Jazz** | **Rap** | **Rock** |
|---|---|---|---|---|
| Modern Blues | Baroque | Bop | Hardcore Rap | Alternative Rock |
| Traditional Blues | Romantic | Swing | Pop Rock | Metal |

**Figure 9.3:** The ten musical genres in the SAC dataset. The columns demonstrate how the genres can be grouped into pairs in order to result in a 5-genre parent taxonomy.

The first part of the experiment was to use jMIR's feature extractors to acquire features from each matched audio recording and MIDI recording, as well as from the Internet, using the cultural metadata associated with each piece. To provide a clarifying example, features might be extracted from a Duke Ellington MP3 recording of *Perdido*, from an independently acquired MIDI encoding of the same piece, and from automated

[258] The most significant fields of which, from the perspective of this experiment, are *Artist, Genre* and *Title*.

search engine queries based on recording metadata such as *Artist, Title* and candidate genres. Three types of features, one corresponding to each data type, were therefore extracted for each piece.

These three types of features were then grouped into all seven possible combinations and subsets using jMIRUtilities.[259] This was done once for each of the two genre taxonomies. This resulted in a total of fourteen sets of features,[260] as shown in Table 9.4. These fourteen sets of features were used to perform fourteen corresponding automatic genre classification experiments.

| Feature Type | 5-Genre Code | 10-Genre Code |
|---|---|---|
| Symbolic | S-5 | S-10 |
| Audio | A-5 | A-10 |
| Cultural | C-5 | C-10 |
| Symbolic + Audio | SA-5 | SA-10 |
| Audio + Cultural | AC-5 | AC-10 |
| Symbolic + Cultural | SC-5 | SC-10 |
| Symbolic + Audio + Cultural | SAC-5 | SAC-10 |

**Table 9.4:** The identifying codes used for each of the fourteen experiments performed.

ACE was trained on and used to classify each of these fourteen feature type combinations in fourteen independent 10-fold cross-validation experiments. This resulted in two classification accuracy rates for each of the seven feature type combinations, one for each of the two genre taxonomies.

It should incidentally be emphasized that ACE includes dimensionality reduction functionality, which is essential in overcoming the *curse of dimensionality,* an important consideration given that most of the feature sets included well over one hundred features. ACE was therefore able to effectively choose only those features that were most orthogonal or showed the most discriminatory power from all of the available features when training its actual classifiers.

[259] jMIRUtilities is a set of tools designed for coordinating the jMIR components and for performing miscellaneous tasks described in Section 8.8.
[260] Although the symbolic and audio features were the same regardless of which of the two genre taxonomies were used, the cultural features differed between the two taxonomies. This variance was due to the fact that jWebMiner queries incorporated the names of the candidate genres. So, even though A-5 and A-10 used the same features, they are assigned different experiment codes for the sake of clarity.

It was desirable not only to determine how effective each of the feature type combinations was at achieving correct classifications, but also how serious misclassifications were. Two classifiers with similar raw classification accuracy rates can in fact be of very different values if the mistakes made by one are much more serious than those made by the other. A *normalized weighted classification accuracy rate* was therefore calculated for each of the 10-genre experiments in order to provide insights on error types. This was calculated by weighting a misclassification within a genre pair (e.g., *Alternative Rock* instead of *Metal*) as 0.5 of an error, and by weighting a misclassification outside of a pair (e.g., *Swing* instead of *Metal*) as 1.5 of an error. This contrasts with the simple classification accuracy rate, where both types of errors counted simply as 1 error.

It is important to note that this weighting approach avoided simply inflating the accuracy rates because "bad" misclassifications were overweighted by just as much as "good" misclassifications were underweighted. This normalized approach contrasts with the approaches used in the past in some MIREX evaluations, for example, which just reduced the weight of "good" errors without increasing the weight of "bad" errors.

### 9.4.4 Results

The average classification accuracy rates across cross-validation folds for each of the fourteen experiments outlined in Table 9.4 are shown in Table 9.5. Both weighted and unweighted results are included. Figures 9.4 and 9.5 illustrate the unweighted results for the 5-genre and 10-genre classifications respectively. These results are summarized in Table 9.6 and Figure 96, which show the average results for all experiments using one feature type, all experiments using two feature types and all experiments using three feature types.

|  | S | A | C | SA | AC | SC | SAC |
|---|---|---|---|---|---|---|---|
| **5-UW** | 86.4 | 82.8 | 87.2 | 92.4 | 95.2 | 94 | 96.8 |
| **10-UW** | 66.4 | 67.6 | 61.2 | 75.6 | 78.8 | 75.2 | 78.8 |
| **10-W** | 66.4 | 67.4 | 66.6 | 78.6 | 84.6 | 81.2 | 84.2 |

**Table 9.5:** The unweighted classification accuracy rates for the 5-genre taxonomy (5-UW) experiments and both the unweighted (10-UW) and weighted (10-W) accuracy rates for the 10-genre taxonomy experiments. Results are reported for each feature type combination, as described in Table 9.4. All values are percentages, and they are all averages calculated across cross-validation folds.

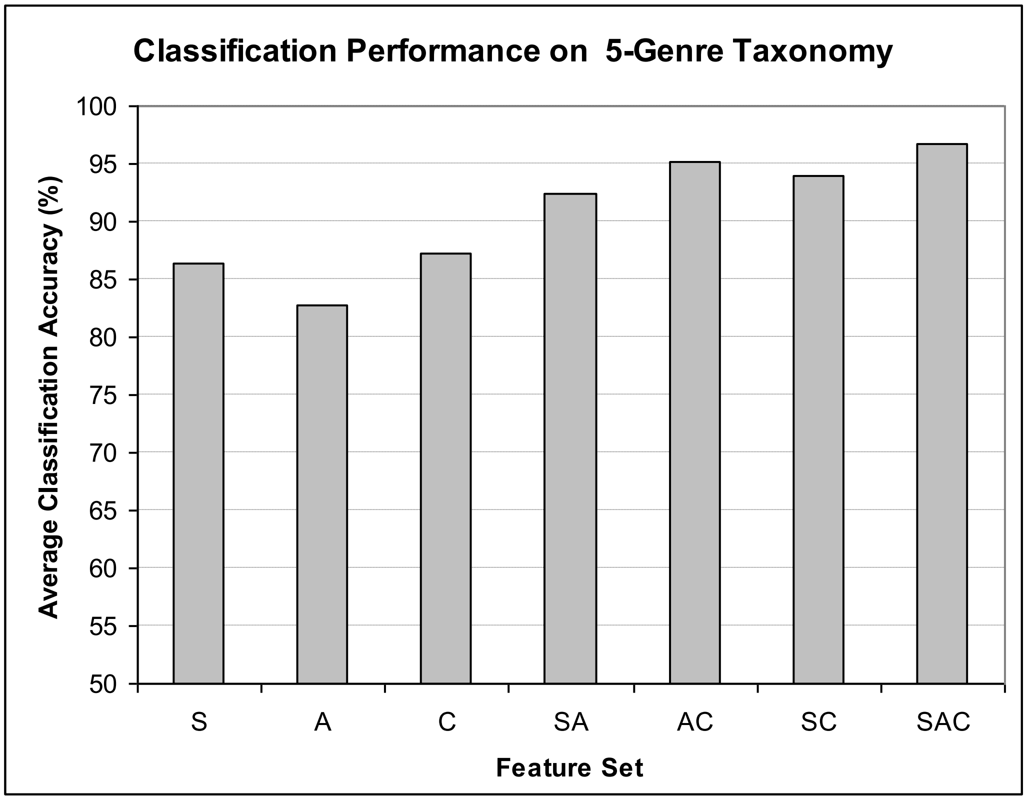


**Figure 9.4:** The classification accuracy rates for the 5-genre taxonomy, as described in Tables 9.4 and 9.5.

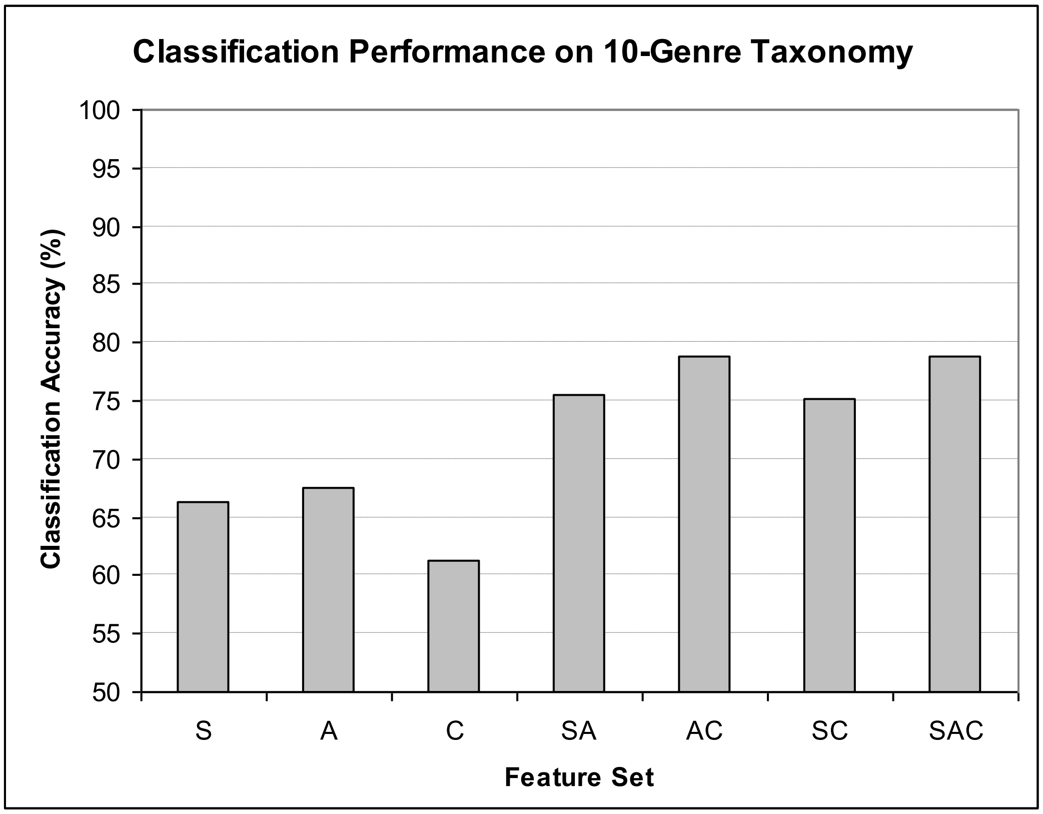


**Figure 9.5:** The unweighted classification accuracy rates for the 10-genre taxonomy, as described in Tables 9.4 and 9.5.

| | 1 Feature Type | 2 Feature Types | 3 Feature Types |
|---|---|---|---|
| **5-UW** | 85.5 | 93.9 | 96.8 |
| **10-UW** | 65.1 | 76.5 | 78.8 |
| **10-W** | 66.8 | 81.5 | 84.2 |

**Table 9.6:** The average classification accuracy rates for all experiments employing just one type of feature (S, A and C), two types of features (SA, AC and SC) or all three types of features (SAC). Results are specified for the 5-genre taxonomy (5-UW), the unweighted 10-genre taxonomy (10-UW) and the weighted 10-genre taxonomy (10-W). All values are percentages, and are calculated as simple mean averages from Table 9.5.

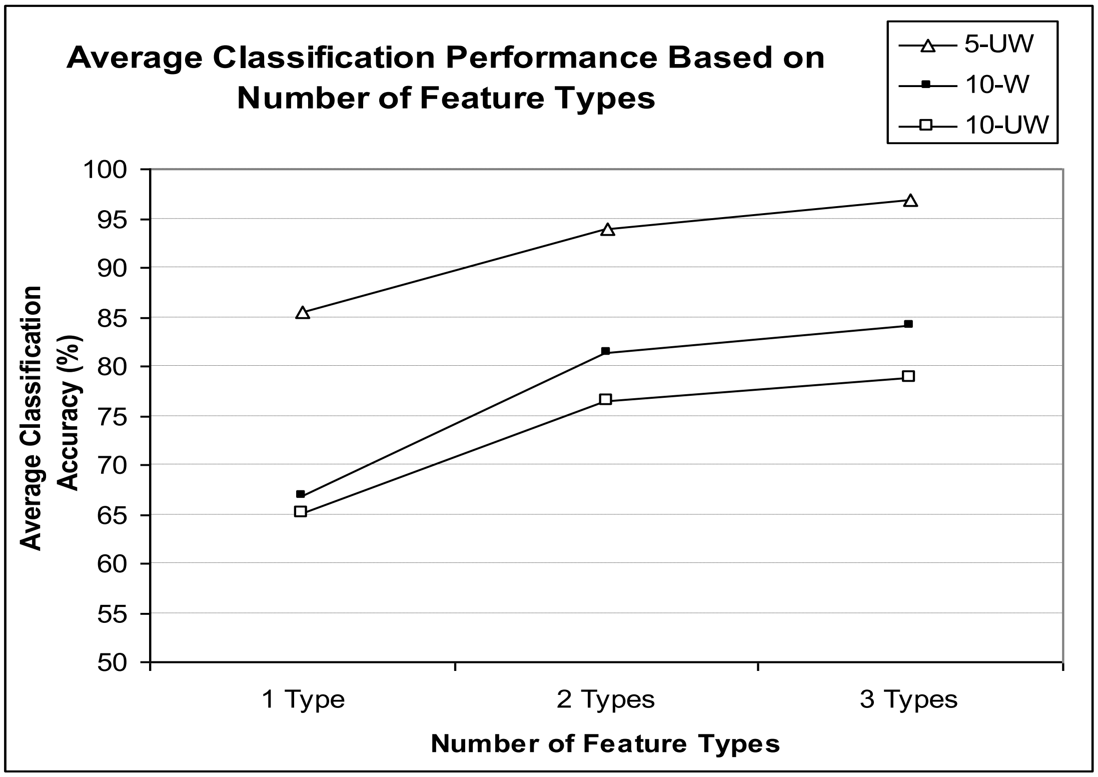


**Figure 9.6:** The average classification accuracy rates for all experiments employing just one type of feature (S, A and C), two types of features (SA, AC and SC) or all three types of features (SAC). The three trend lines refer to the 5-genre taxonomy (5-UW), the unweighted 10-genre taxonomy (10-UW) and the weighted 10-genre taxonomy (10-W). This data corresponds to Table 9.6.

### 9.4.5 Discussion of the effects of combining feature types on classification accuracy

As can be seen in Figures 9.4 and 9.5, all combinations of two feature types performed better than all single feature types classified independently. Furthermore, combining all three feature types for the most part resulted in better performance than all pairs of feature types, although this increase was less dramatic.

These results are illustrated in Figure 9.6, which shows strong average increases in performance when feature types are combined. On average, combining all feature types resulted in an increase in performance of 11.3% in the 5-genre taxonomy and 13.7% in the 10-genre unweighted case compared to the average performance of each of the single feature types classified individually. Considered in terms of proportional reductions in error rate, this corresponds to decreases of 78.0% and 39.3% for the 5 and 10-genre taxonomies respectively.

The Wilcoxon signed-rank test indicates that, with a significance level of 0.125, the improvements in performance of two or three feature types over one type were statistically significant in all cases. However, the improvements when three feature types were used instead of two were not substantial enough to conclude with a high degree of statistical certainty that three feature types are always better than two, although this may still be the case. The corresponding average increases in performance were only 2.3% and 2.7% for the 5 and 10-genre taxonomies, respectively. What is clear, however, is that both two and three feature types consistently resulted in better classification success rates than only one feature type.

All of this provides a strong indication that the three different types of features contain orthogonally independent information, and can therefore be fruitfully combined for a variety of purposes. This helps to confirm the intuitively reasonable suspicions that cultural features hold useful information not accessible from either symbolic or audio data and that audio files contain information that MIDI files do not.

What may be more surprising to some, however, is that the symbolic data appears to contain information that is not accessed by the features extracted from the audio files. This is implied by the facts that the AS feature set performed 9.6% better than the A feature set in 5-UW, and 8% better in 10-UW. This is potential confirmation of the suspicions expressed above that symbolic data has important representational advantages over raw audio data with respect to making some types of information more conveniently accessible to machine learning algorithms. This, in turn, implies that mappings from audio to MIDI resulting from improved audio transcription technology could result in substantially improved classification performance as a side benefit.

As an additional point, it is interesting to note that the standard deviation across single feature types was only 2.3 for the 5-genre taxonomy and 3.4 for the 10-genre taxonomy, and would have been even lower if not for the relatively low performance of the cultural features on the 10-genre taxonomy. This may indicate that the three types of features are roughly equivalent in terms of their genre distinguishing ability when considered individually.

### 9.4.6 Discussion of types of misclassification

As described in Section 9.4.3, normalized weighted classification accuracy rates were calculated for the experiments on the 10-genre taxonomy in order to evaluate the seriousness of the particular misclassifications that were made. The results, and how they compare to the unweighted classification accuracies, are shown in Table 9.5 and Figure 9.7.

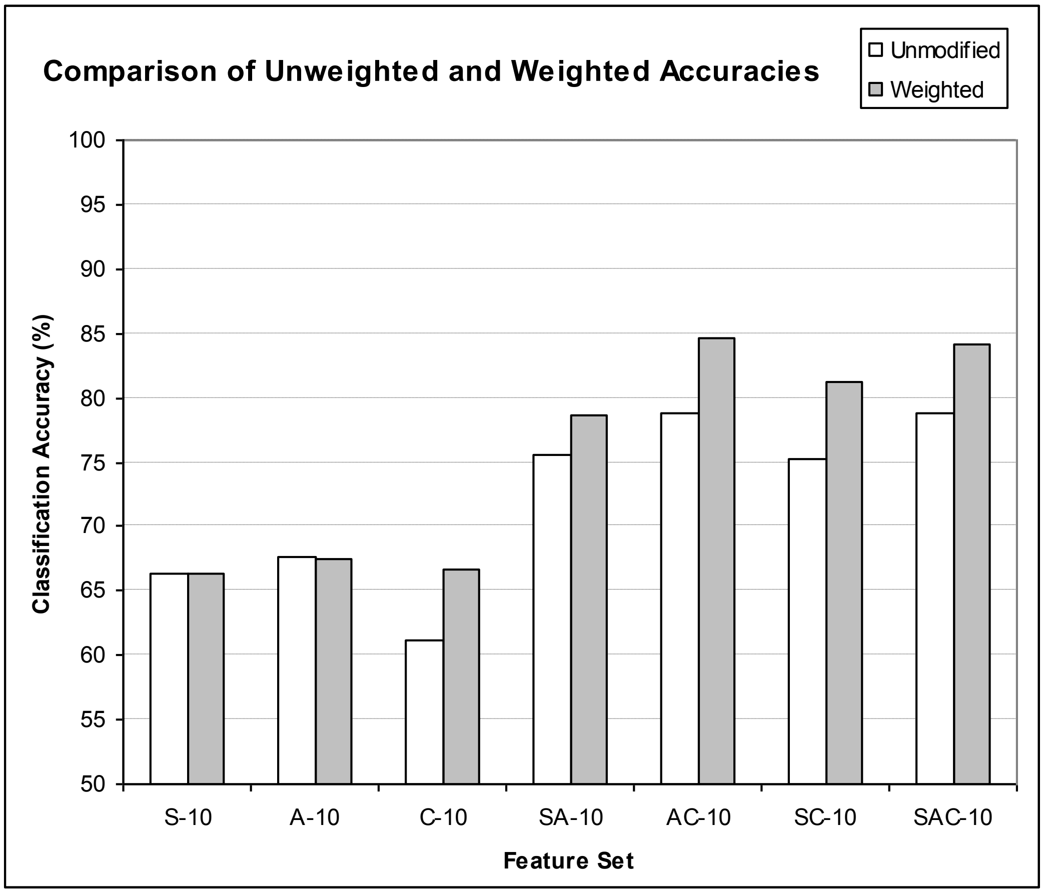


**Figure 9.7:** The difference between the unweighted and weighted classification accuracies on the 10-genre taxonomy for each of the seven feature type combinations (Table 9.4). This data correspond to the last two rows of Table 9.5.

As can be seen in Figure 9.7, the weighted and unweighted accuracies were not significantly different when the audio and symbolic features were processed individually. However, the weighted performance was 3% higher than the unweighted performance when the two feature types were combined. Although this is not a dramatic increase, it is an indication that combining these feature types may make those misclassifications that do occur less serious in addition to increasing classification accuracy overall. Recall, also, that the normalized weighting does not artificially inflate results, as very different misclassifications are overpenalized just as much as not too different misclassifications are underpenalized.

The differences between the weighted and unweighted classification accuracies were greater in all feature sets that included cultural features. These weighted rates were higher than the unweighted rates by an average of 5.7%, a difference that, based on Student's paired t-test, is statistically significant with a significance level of 0.005. This indicates that the types of misclassifications that occur when cultural features are used are likely less serious than when audio or symbolic features are used alone. Quite encouragingly, it also appears that this improvement in error quality carries through when cultural features are combined with both audio and symbolic features, as the SAC-10 weighted success rate was 5.4% higher than the unweighted success rate.

One interesting note is that many of the classification errors in general occurred in the two Rap genres. These genres were correctly classified with only a 68% success rate for the SAC-10 experiment, for example, compared to 81.5% for recordings belonging to the other eight genre classes. This relatively poor performance on Rap was for the most part consistent across feature types and experiments, and the was emphasized by the relatively similar classification success rates on average for the other eight genre classes. This is surprising, given that Rap would intuitively seem particularly easy to distinguish from other genres. This is an intriguing result, and bears further investigation.

### 9.4.7 Discussion of absolute genre classification performance

In order to put the experimental results presented here in context, it is appropriate to compare them with classification accuracies achieved by alternative high-performing specialized genre classification systems. It is important, however, to keep in mind the essential caveat that different classification systems can perform dramatically differently on different datasets, so direct comparisons of classification accuracies calculated on different datasets can give only a very rough indication of comparative performance. The size and depth of the genre taxonomy, the particular genre labels chosen as candidates, the quality of the labelling of the ground-truth, the particular instances chosen to include in the dataset and the relationships between them are just some of the many factors that can have a deep impact on the difficulty of a dataset.

The MIREX evaluations offer the best benchmarking reference points available. Although no evaluations of genre classification based on cultural data have been carried out yet at MIREX, both audio and symbolic genre classification evaluations have been held, most recently in 2009 and 2005, respectively. The highest classification accuracy attained in audio classification was 73.3%, achieved by Cao and Li on a 10-genre ontology.[261] The highest accuracy for symbolic classification was 84.4%, attained on a 9-genre taxonomy by McKay and Fujinaga's Bodhidharma system.[262]

The experiments described in this thesis achieved classification accuracies of 67.6% using only features extracted from audio and 66.4% using only features extracted from symbolic data. This is at least comparable to the best MIREX audio result of 73.3%, but significantly lower than the best MIREX symbolic result of 84.4%, which were achieved on a taxonomy only smaller by one class.

This latter result is intriguing, as jSymbolic uses the same features and feature implementations as Bodhidharma. The difference is likely due at least in part to the specialized and sophisticated hierarchical and round robin learning classification ensemble algorithms used by Bodhidharma (McKay 2004), whereas ACE only experiments with general-purpose machine learning algorithms. This offers strong support for future research experimenting with sophisticated classification schemes such as Bodhidharma's.

When all three feature types were combined, the jMIR experiments described in this paper achieved a success rate of 78.8% which was still lower than Bodhidharma's performance, but significantly better than the best audio MIREX results to date.

Even though 78.8% is still much too low for practical applications, these results are particularly encouraging given the particular difficulty of the SAC dataset compared to datasets used in many other genre classification experiments. SAC consists of pairs of closely related genres, includes different versions of the same pieces performed in different genres, includes different pieces by the same artist in different genres, includes a

[261] www.music-ir.org/mirex/2009/index.php/Audio_Genre_Classification_%28Mixed_Set%29_Results
[262] www.music-ir.org/evaluation/mirex-results/sym-genre/index.html

diverse variety of subtypes within each genre, etc., all in order to more closely approximate realistic genre conditions. These aspects of SAC that make it a particularly difficult dataset to correctly classify are discussed in more detail in Section 8.7. Furthermore, only 25 recordings were available per genre in SAC, and these needed to be divided into training, testing and validation sets, which left only relatively few recordings for classifiers to train on.

Taken in the context of the particular difficulty of the SAC dataset, and considering that the accuracy on the 10-genre taxonomy improves to 84.2% when weighted, the results attained here are encouraging, and may be an indication that the ultimate ceiling on performance (Aucouturier and Pachet 2004) might not be as low as some have worried. It may well be that the use of more sophisticated machine learning approaches, such as those used by Bodhidharma or by DeCoro, Barutcuoglu and Fiebrink (2007), combined with the development of new features, could significantly improve performance further.

### 9.4.8 Conclusions

The results of these experiments indicate that it is indeed substantively beneficial to combine features extracted from audio, symbolic and cultural data sources, at least in the case of automatic genre classification. Further research remains to be performed investigating whether these benefits generalize to other areas of music classification and similarity research, although the outlook seems good.

All feature groups consisting of two feature types performed significantly better than any single feature types classified alone. Combining all three feature types resulted in still further improvements over the feature type pairs on average, but these additional improvements were not as uniformly dramatic.

The results also indicated that combining feature types tends to cause those misclassifications that do occur to be less serious, as the misclassifications are more likely to be to a more similar class. Such improvements were particularly pronounced when cultural features were involved.

Encouragingly high genre classification accuracy rates were attained. The results of the experiments as a whole provide hope that any ultimate performance ceiling on genre classification performance might not be as low as has been worried.

In terms of meeting the first two experimental goals outlined in Section 9.4.1, the jMIR software suite was demonstrated to be an effective and convenient tool for performing feature extraction and classification research and for effectively combining information extracted from the different musical data sources. Although results were clearly improved when feature types were combined, it was also shown that each of the jMIR feature extractors was able to produce features that could be used individually to perform classifications with success rates over six times better than chance in all 10-genre classification experiments.

## 9.5 Summary of original contributions

The empirical evaluations in this chapter demonstrated the utility of ACE, jAudio, jSymbolic and jWebMiner as tools that can be used to produce excellent classification results when applied a variety of MIR research areas. It was also shown that the jMIR components could be used together effectively in order to combine features extracted from audio, symbolic and cultural sources and build corresponding combined classification models using ACE.

The preliminary experiments described in Section 9.2 demonstrated that jAudio and ACE can be used without customization or specialized parameterization to quickly and easily achieve similar or better than previously published classification success rates when applied to both MIR and general classification tasks. The independently evaluated MIREX results discussed in Section 9.3 demonstrate that jSymbolic's features are the most effective to date at classifying symbolic recordings by genre to date.

To the best knowledge of the author, the experiments described in Section 9.4 are the first to combine features extracted from all three sources of musical data for the purpose of automatic music classification. They are also the first to systematically compare the performance of all possible combinations and subsets of the feature types.

These experiments provided strong evidence that combining features extracted from audio, symbolic and cultural information sources can substantively improve classification results compared to only using one type of data. Classification accuracies of 96.8% and 78.8% were attained respectively on 5- and 10-class genre taxonomies when all three feature types were combined, compared to average accuracies of 85.5% and 65.1% when features extracted from only one of the three sources of data were used. It was also found that combining feature types decreased the seriousness of those misclassifications that were made, on average, particularly when cultural features were included.

## 9.6 Future research

Now that symbolic and audio data have been found to be capable of providing at least partially orthogonal features, an important next step is to supplement the experiments performed in Section 9.4 by repeating them with MIDI files transcribed from audio, in order to investigate more practically significant use cases where MIDI files do not have to be manually harvested. This would also enable experiments on much larger audio datasets, such as Codaich, rather than limiting one to the much smaller universe of MIDI files. Preliminary investigations of existing and forthcoming audio transcription technologies will be the first step towards this goal. Although less directly useful from a practical value, it would also be interesting to investigate the relative classification performance when audio files are synthesized from MIDI files, in order to attain a comparative baseline.

There are also plans to perform experiments where feature types are combined in more sophisticated ways, such as by segregating them among different specialist classifiers collected into blackboard ensembles. Further experiments with more sophisticated classification techniques in general, such as those utilized in Bodhidharma (McKay 2004) or by DeCoro, Barutcuoglu, and Fiebrink (2007), also bear further investigation, and could fruitfully be incorporated into ACE.

Since one of the essential goals of the jMIR package is to provide a framework applicable to a wide variety of MIR applications with little or no fine tuning being necessary, a key priority in the future is to perform experiments with the mature jMIR release in a variety of MIR research areas relating to both classification and similarity. Submitting jMIR to future editions of the MIREX competitions in a number of different areas would be an excellent opportunity to gain feedback on its performance relative to state-of-the-art specialized approaches.

# 10. Conclusions and future research

## *10.1 Summary of dissertation and original contributions*

This dissertation presented a suite of software applications called jMIR that are intended for use in performing research involving automatic music classification. jMIR is the only existing unified music information retrieval research toolset that includes software for performing symbolic, audio and cultural feature extraction; meta learning software for selecting, training and applying machine learning algorithms; large and well-labelled musical research datasets; software for profiling and detecting errors in musical metadata; and standardized file formats for representing feature values, instance class labels, class ontologies and associated metadata in flexible and expressive ways.

Certain essential objectives are emphasized in the design and implementation of each of the jMIR components, especially:

- Providing software that may be used to apply automatic music classification to research problems in ways that are accessible to researchers from a diverse range of music-related research disciplines.
- Providing a framework for researchers to develop, test and share new automatic music classification algorithms, particularly new features and machine learning strategies.
- Facilitating research that combines information extracted from different types of musical data.

The jMIR components are designed such that users may utilize them either as separate independent units or as an integrated whole, as they prefer. The jMIR components are as follows:

- **jAudio:** A feature extractor for extracting information from audio files.
- **jSymbolic:** A feature extractor for extracting information from MIDI files.
- **jWebMiner:** A feature extractor for extracting cultural information from the Internet.

- **ACE:** Meta learning software for experimenting with, selecting, training and applying pattern recognition algorithms.
- **ACE XML:** A set of standardized file formats that can be used to store information such as extracted feature values, feature metadata, class labels associated with instances, miscellaneous instance metadata and class ontologies.
- **jMusicMetaMangaer:** A tool for profiling large musical datasets as well as automatically detecting metadata errors, inconsistencies and redundancies.
- **Codaich, Bodhidharma MIDI and SAC:** Three labelled musical datasets for use in debugging, validating and benchmarking new automatic music classification technologies and for performing general exploratory computational musical research.
- **jMIRUtilities:** A set of tools for performing miscellaneous tasks associated with jMIR.

The provision of these jMIR software tools to music researchers is itself a substantial contribution to the field of music information retrieval, as there is currently no other toolset designed specifically for automatic music classification research with anything approaching the breadth and scope of jMIR. The jMIR project has resulted in the implementation of a great deal of functionality. The software consists of over 95,000 lines of original code[263] at the time of this writing, not including the original Bodhidharma code or the work currently underway on the ACE XML 2.0 support libraries. Moreover, this code is based around a carefully designed architecture that emphasizes extensibility and modularity.

All of the jMIR code is also open-source, so the processing and analysis functionality that it implements can be examined directly and extended by those inclined to do so. jMIR thus performs the dual roles of providing music researchers in general with ready-to-use tools and of providing algorithm developers with a framework for implementing new approaches and then testing and distributing their work to other researchers.

[263] And over 3.8 million characters.

A number of important additional contributions to the field of music information retrieval have also been produced during the course of the jMIR project, beyond simply the essential functionality offered by the software itself. Certain highlights are summarized below, with more detailed information available in the individual chapters of this dissertation:

- General limitations and problems associated with the current state of automatic music classification research are discussed, and guidelines are proposed for addressing these concerns and moving past them.[264]
- Automatic music classification is considered in the context of theoretical and experimental insights from the fields of music psychology, music theory and musicology. This contrasts with the more typical MIR approach of simply viewing automatic music classification as an engineering and machine learning problem. From the perspective of the author, it is essential that MIR researchers move past this view, as music classification is fundamentally associated with the human understanding of music, not raw audio data devoid of context. It therefore seems intuitively clear that automatic music classification should best be approached with a musically and psychologically informed awareness.[265]
- Although all jMIR components are designed to be extensible and modular, jAudio is presented as a particularly developed example of how such design goals can be fully expressed in the context of music classification software. This is exhibited by jAudio's metafeatures, aggregagators, automatic handling of feature dependencies, automatic extraction scheduling, full feature plug-in support and functionality for generating audio for the purpose of testing and debugging new features.[266]
- A number of important principles associated with the design and choice of new high-level musical features are discussed. This analysis and the resultant

[264] Chapter 1
[265] Chapter 2
[266] Chapter 3

suggestions can be of use to those designing high-level musical features of their own.[267]

- The jSymbolic feature catalogue is much larger than any other existing set of high-level musical features. Moreover, many of these features are entirely original. Taken together, these 111 implemented features and 42 additional proposed features provide a solid basis for extracting musically meaningful statistical information from music. This feature catalogue has only begun to be taken advantage of in music classification research, and it also holds the potential to be applied to more analytically oriented work.[268]
- The jMIR components are designed to have interfaces that make them accessible to music researchers in general, including researchers with little or no experience in computational research. The jMIR interfaces and the design principles behind them are therefore themselves useful contributions to MIR research. For example, jWebMiner and ACE respectively implement data mining and machine learning algorithms that have largely already been used previously by others, but package this functionality in ways that make it useful and accessible specifically to music researchers in ways that it previously was not.[269]
- A relatively detailed introduction to and overview of machine learning is provided. Although this particular section of the dissertation certainly does not describe original research, it is presented in such a way as to make it accessible even to music researchers with only a minimal background in statistics and computer science. It also places a particular emphasis on how machine learning techniques can be applied to music research. Most existing machine learning texts, in contrast, tend to presume a significant mathematical and computational background on the part of the reader, and also tend to focus more on detailed theoretical aspects of machine learning that are useful when designing new algorithms and less on the practical problems that one encounters when actually using machine learning. This is precisely the wrong focus from the perspective of

[267] Chapter 4
[268] Chapter 4
[269] Chapters 3, 4, 5 and 6

the typical music researcher, who lacks the expertise necessary for developing new algorithms, and who simply wishes to use existing algorithms effectively. The consequence of this is that existing machine learning texts tend to be alienating to the majority of music researchers who, as a consequence, have a tendency to simply blindly use existing off-the-shelf implementations without a true understanding of the algorithms that they are choosing or of whether they are in fact well-suited to their particular problems. The machine learning summary presented in this dissertation is a useful contribution because it provides music researchers with an accessible quick-and-easy yet still reasonably in-depth introduction to machine learning that focuses on the particular practical issues that one actually encounters in automatic music classification research. This will hopefully allow music researchers in general to approach machine learning a little less naïvely and more effectively. ACE itself also still provides the option of automatically selecting algorithms to use via meta learning based on empirical evidence.[270]

- A theoretical discussion is provided of problems and issues to consider when choosing or designing data representation formats associated with automatic music classification. This includes the proposal of an original list of related design principles.[271]
- ACE XML is an entirely original standard for representing information specifically related to automatic music classification. It is presented to the MIR community as a general standard for storing and communicating such information between different music software applications and research groups, not just as the native format used by the jMIR components.[272]
- A number of algorithms are proposed (and implemented in jMusicMetaManager) for detecting metadata errors and redundancies in musical collections. These include both purely original algorithms, such as the reordered word subset operations and the merge-based coordination algorithm, as well as original

[270] Chapter 6
[271] Chapter 7
[272] Chapter 7

variants of established edit distance-based techniques. For example, the find/replace transformations facilitate the detection of certain types of errors that are particularly prevalent in musical metadata. The jMusicMetaManager algorithms do not rely upon fingerprinting, as do most musical metadata systems, and can in fact be used to clean the often inconsistent metadata provided by fingerprinting services.[273]

- A reasonably extensive theoretical discussion is provided of problems and concerns relating to MIR research datasets, and a number of general guidelines are proposed for designing, building and using such datasets effectively. A particular emphasis is placed on classes and class ontologies.[274]
- The Codaich, SAC and Bodhidharma MIDI datasets provide diverse, well-labelled and clean musical data in both audio and symbolic forms. These musical datasets have already been used in research at McGill University, where there is legal access to them, and it is hoped that the datasets will be made accessible to the MIR research community in general once OMEN or some other system like it allows features extracted from the data to be distributed legally.[275]

The experiments[276] performed as part of the jMIR project also provide a useful general contribution to MIR research, particularly the results of those experiments involving features extracted from multiple different types of musical data. More details are provided below in Section 10.2.

It is hoped that the jMIR components will be used and extended by a wide variety of music researchers in the future. Both the software and its associated documentation may be freely downloaded from jmir.sourceforge.net. As of 23 April 2010, there have been 3897 logged downloads of jMIR components from the jMIR site, and jMIR components have also been used in several research projects performed independently of the author.[277] It therefore appears that the first of these hopes has begun to be realized. Unfortunately,

[273] Chapter 8
[274] Chapter 8
[275] Chapter 8
[276] Chapter 9
[277] Examples include the work of Bellaachia and Jimenez (2009); DeCoro, Barutcuoglu and Fiebrink (2007); and Hillewaere, Manderick and Conklin (2009).

the only jMIR development work to date has been performed by the author and his research collaborators at McGill University.[278] It is therefore clear that additional work remains to be done on promoting jMIR as a development framework, an issue that is discussed further in Section 10.3.

## 10.2 Conclusions drawn from experimental results

The experiments described in Chapter 9 have, to begin with, demonstrated that that the jMIR components may serve as useful and effective tools with respect to a variety of automatic music classification research projects. In particular, the winning submission of Bodhidharma to the MIREX Symbolic Genre Classification contest, as described in Section 9.3, demonstrated that jMIR technology can perform favourably when compared to other systems.

The experiments on combining features extracted from different types of musical data, as described in Section 9.4, also helped to demonstrate the effectiveness of jMIR. Not only were good classification results achieved on the hard SAC dataset when various jMIR components were used separately, but significantly improved results were achieved when features were extracted and combined using two or three jMIR feature extractors together. This demonstrated the ability of the jMIR components to work well both individually and together.

The results from the experiments described in Section 9.4 also have an importance beyond just the scope of jMIR itself. It was found that combining features extracted from audio, symbolic and cultural sources of musical data in any combination significantly improved results compared to when features were only extracted from one type of data. Moreover, it was found that combining features extracted from either audio or symbolic data, or both, with features extracted from cultural data decreased the seriousness of those misclassifications that did occur, in addition to reducing the number of misclassifications overall.

These results support the utility of producing a framework specifically designed to facilitate the combination of data extracted from multiple types of musical information, which was one of the core design principles underlying jMIR. These results also suggest

[278] Some development work using jMIR was also done independently of the author at Sun Microsystems, but this work was closed source, and is therefore not available to the public.

that work should continue on finding effective ways of combining information extracted from diverse sources. Although it remains to be determined whether combining additional types of musical data will improve classification performance still further, this is an area that bears further investigation. Ideas for future research related to this are discussed in Sections 10.3.9 and 10.3.10.

## *10.3 Future research*

This section proposes only areas of future research and development that are related to jMIR as a whole. Future work that is specifically relevant to particular jMIR components is not included here, as Chapters 3 through 9 each contain their own sections proposing research of this kind.

### 10.3.1 Adding functionality to the existing jMIR components

As noted in the individual Future Research sections of Chapters 3 through 9, there are a number of important types of functionality that could be added to each of the jMIR components. Moreover, it is certain that still further types of useful functionality will become apparent in the future as MIR research progresses. The addition of functionality of both kinds is therefore a priority of future jMIR development.

The development of new features and machine learning algorithms are two areas of particular interest, as jMIR has been designed to make it especially easy to add functionality of this kind. As noted above, it is hoped that many researchers in MIR-related fields, not just the author, will eventually use the jMIR framework to develop such technologies and distribute them communally so that they may be added to the jMIR distributions used by others. Approaches to accomplishing this are discussed below.

### 10.3.2 User studies and interface improvements

One of the core goals of jMIR is to make sophisticated automatic music classification technologies available to researchers of all kinds, including both users with extensive backgrounds in computational research and researchers with little or no background of this kind. Although efforts have certainly been made to help achieve this via the current jMIR interfaces, these efforts have ultimately been limited by the scope of a single dissertation research project with many varied facets.

The next step towards fully realizing this goal will involve performing detailed user studies of diverse groups of music researchers. Such users will not only be able to highlight how the current jMIR interfaces succeed and fail to meet each of their particular needs, but will also likely be able to propose new ideas that can be incorporated into jMIR.

### 10.3.3 Software testing and support

Efforts have certainly been made to test the jMIR components extensively, both individually and together. Such tests have involved standard software engineering methodologies, such as unit testing, as well as actual use of the jMIR components in research, as described in Chapter 9.

Having noted this, it is inevitable in any reasonably large-scale and complex software development project such as jMIR that some problems will evade initial detection, as they will only become evident under specific conditions that can be difficult to anticipate. Such problems can be particularly difficult to discover exhaustively in cross-platform software like jMIR. Furthermore, as time goes by many changes can occur in the computing environments in which software operate, such as changes to the Java Runtime Environment or to operating systems. Even as a problem is detected, the changes made to correct the issue can potentially themselves lead to other new unanticipated problems, even if one is very careful. Such problems are a common general issue in modern application development, and can be particularly evident in academic projects where just one or a very few developers are responsible for all design, implementation and testing.

An important area of further work is therefore the continuing support and updating of jMIR. User studies such as those discussed in Section 10.3.2 will be helpful in discovering potential bugs, as will the general use of jMIR in research projects. Many users have already downloaded jMIR components from the jMIR SourceForge page and used them in their own work, and correspondence with several of them has helped to reveal bugs that could then be corrected. Work will continue on addressing issues as they are reported, as well as on performing further formal error testing as jMIR is expanded.

There are also plans to use jMIR to submit entries to the annual MIREX annual MIR competition in as broad a range of categories as possible. This will help to further validate jMIR's performance in a variety of research applications, and will also help to highlight potential areas of weakness resulting in poor performance, which can then be addressed.

### 10.3.4 Diversifying ways of accessing jMIR's functionality

As noted previously, extensive efforts have already been made to make jMIR's functionality as accessible as possible, in terms of both jMIR's code base and the direct usability of its components as a set of ready-to-use applications. There are, however, a number of additional steps that can be taken to increase jMIR's accessibility still further. For example, jMIR in its current state is intended specifically for use in research applications. Much of its functionality could also be usefully adapted in the future to more casual applications, however.

One of the most effective ways of making jMIR's functionality accessible to casual users would be to port aspects of jMIR to various popular plug-in formats. This would allow users of all kinds to access this functionality via software applications that they already have access to and are already familiar with. For example, jAudio and jSymbolic's feature extraction functionality might be made available in the form of VST plug-ins so that it could be accessed from any of the many digital audio software systems with VST compatibility. Even systems without direct VST compatibility, such as Audacity,[279] could make use of such jMIR VST plug-ins via a VST shim.

It would also be useful to implement some aspects of jMIR in the form of plug-ins for common media players, such as iTunes,[280] Winamp[281] or Windows Media Player.[282] For example, many users might find it useful to access aspects of ACE and jMusicMetaManager's functionality to predict and correct metadata associated with their music collection. jAudio, jSymbolic and/or jWebMiner feature extraction functionality could also be added to media players via plug-ins.

Although the jMIR code is very well documented, and is designed to be as modular and extensible as possible, some developers might prefer to access jMIR's functionality without needing to access the jMIR code directly. It would therefore be useful to embed the jMIR components in a web services wrapper so that jMIR functionality could be

[279] audacity.sourceforge.net
[280] developer.apple.com/sdk/
[281] www.winamp.com/plugins
[282] www.microsoft.com/windows/windowsmedia/player/plugins.aspx

accessed over a network connection without the need to install jMIR libraries locally. McEnnis has already begun work on such an approach as part of the OMEN project (McEnnis 2006), and it would be valuable in the future to complete this work and expand its scope.

### 10.3.5 Public algorithm repository

One of the most effective ways of encouraging researchers to share their algorithm implementations with others is to provide simple and effective ways of allowing researchers to access and use these implementations. jMIR's plug-in oriented architecture provides a first step in this direction, as researchers can implement algorithms in the jMIR framework in such a way that that others can then add these implementations to their own jMIR distributions relatively easily. It would be even more helpful, however, if researchers also had a common data space to which they could publish such jMIR algorithm implementations.

The creation of such a repository on-line is an important part of the planned future development of jMIR. This would allow developers to upload their original jMIR algorithm implementations to the repository, including the original Java source code, compiled Java bytecode, associated documentation and any other relevant information. Others could then download the algorithm implementations that they are interested in, plug them into their jMIR distribution and use them in their own research. It will also be important to add a version control framework to the repository, in order to make it easier for researchers to work collaboratively on algorithms if they wish to.

### 10.3.6 Distributed computing

Many of the tasks associated with automatic music classification can be computationally expensive, thereby limiting the amount of data that can be processed quickly.[283] Furthermore, as noted in Chapter 8, there are legal barriers limiting the extent to which musical datasets can be shared amongst researchers. Both of these issues can significantly impact the ability of different research groups to compare the effectiveness of their solutions to MIR problems with solutions found by others. This is because proper comparative evaluations require that different approaches be applied to the same data, and

[283] Automatic classification still tends to be much faster than manual human classification, however.

that a large amount of data be processed in order to verify the musical scope of different approaches and achieve sufficient statistical certainty.

Distributed computing can help solve these problems. The computational problem is addressed simply by having many computers process different aspects of a problem at once, so that a solution is achieved sooner. The problem of overcoming legal barriers in order to test one's approach on the same musical data used by others is addressed by allowing features to be extracted on-site at those potentially geographically distributed locations that do have legal access to the music, and then communicating only the extracted features, not the actual music itself. This is exactly the approach taken by OMEN (McEnnis 2006).

The incorporation of distributed computing functionality into each of the jMIR components is therefore a priority for future development. Initial work in this direction has already begun with respect to jAudio and ACE, and the framework begun as part of the OMEN project specifically with respect to jAudio could be helpful in continuing it.

Modifying the jMIR components to take advantage of distributed computing in such a fashion would also mesh well with the plans to implement a web services wrapper for jMIR and create a shared public algorithm repository, as discussed above in Sections 10.3.4 and 10.3.5, respectively. The resources used to run the repository could also be used to coordinate a cloud of participating computers, each ideally with legal access to musical data, and distributed experiments could be initiated on this cloud using web services.

### 10.3.7 NEMA integration

Work is currently underway on integrating jMIR into the *Networked Environment for Music Analysis (NEMA)*[284] project for the express purpose of achieving the objectives discussed in Section 10.3.6. The goal of NEMA is to create an open and extensible web services-based framework that facilitates the integration and use of music data and processing tools by the global MIR communities on a basis that is independent of location. This framework is being built by taking advantage of the *Software Environment*

[284] nema.lis.uiuc.edu

*for the Advancement of Scholarly Research (SEASR)*[285] project's distributed computing and application integration infrastructure.

Additional general improvements to jMIR are also continuing as part of the NEMA project. Significant work has already been done on developing ACE 2.0 and ACE XML 2.0 thanks to NEMA funding, and NEMA-related work on jMIR will continue on past the publication of this dissertation.

### 10.3.8 New jMIR components

There are several potentially important types of music classification-related tasks that are not yet addressed by any of the jMIR components. There are therefore plans to implement additional components that will broaden the scope of jMIR:

- **jLyrics:** A feature extractor for extracting information from song lyrics. jLyrics will extract features consisting of simple statistics, such as word frequency counts and word transition probabilities, as well as more sophisticated features, including both original features and features published as part of the rich existing literature on text mining. An additional goal of jLyrics is to include functionality for automatically acquiring the lyrics from the Internet based on metadata like song and artist names. This latter task will be performed via web services offered by existing lyrics repositories and, if necessary, via web scraping functionality. Hu, Downie and Ehman (2009) have already achieved good results when using lyrical features alone and in combination with audio features, and it will be interesting to investigate this area further.

- **jImage:** A feature extractor for extracting information from images, with a particular focus on album covers. This software will provide the ability to extract both standard features from the image processing literature and features customized for the particularities of musical images. jImage will also include functionality for extracting images of album art from various sources available on the Internet based on identifying metadata.

[285] seasr.org

- **jVideo:** A feature extractor for extracting information from music videos, as well as for mining the videos themselves from the Internet based on identifying metadata.
- **jStructure:** Software for automatically segmenting audio streams in time, both in terms of partitioning entire pieces of music within a single stream and in terms of structural divisions within individual pieces. The software will also attempt to classify the overall form of each piece and label each segment with the appropriate corresponding structural labels.
- **jTags:** A feature extractor that will take advantage of web services offered by a variety of on-line resources in order to extract listener tags of various kinds from data on the Internet. This software will also include functionality for combining tags from different sources and removing noise from these tags.
- **jMusicVisualiser:** Software for visually examining and exploring relationships between musical instances, features and classes in either two or three dimensional spaces. Users will be able to fully customize what is to be plotted along each axis and how data points are to be colour coded in order to allow them to view the data in the ways that are the most convenient to them. jMusicVisualiser will also provide users with the ability to explore music that has been automatically clustered once unsupervised learning functionality is incorporated into ACE.
- **jClassOntology:** An application for mining class ontologies from the web. As noted in Chapter 8, the class ontologies used in many automatic music classification experiments tend to be unrealistic and simplistic. jClassOntology will use data mining techniques to empirically build candidate class ontologies from both information available on the Internet in general and from particular selected sources of information.

All of these proposed new jMIR components will, of course, follow the same design principles of jMIR as whole, and will use ACE XML as a means of storing and communicating data.

### 10.3.9 Further experiments on combining different types of data

jAudio, jSybmolic and jWebMiner currently extract features from the types of musical data that have received the most attention from the MIR research community, which is to say, respectively, audio, symbolic and general textural information available on-line. There are, however, many other types of musical data from which potentially useful features can be extracted, including lyrics, still images and video. Moreover, these kinds of information are readily accessible on-line. As can be observed from an examination of the proposed new jMIR components described in Section 10.3.8, there are plans to build jMIR tools for extracting features from just these kinds of data sources.

The next step once such features are accessible will be to perform the same types of experiments described in Section 9.4, but this time examining the relative performance of features extracted from all possible combinations of matching audio, symbolic, cultural, lyrical, visual and video data for each musical instance. It is hoped that this will help find the best types of data to combine for solving various types of automatic music classification problems and, consequently, increase performance.

### 10.3.10 Automatic mining and integration of data sources

In practice, it can be very difficult and time consuming to manually acquire matching instances belonging to different types of musical data in order to perform the types of experiments described in Sections 9.4 and 10.3.9. Collecting the matching audio, symbolic and metadata information for the SAC dataset was very time consuming, and manually adding additional types of data like lyrics, images and video would compound this issue further.

A successful multimodal approach to automatic music classification will therefore likely require that the harvesting of different types of data for each given musical instance be automated. Ideally, it should be possible to provide a successful multimodal system with just one type of data, such as just audio or just lyrics, and then have the system automatically acquire the corresponding data of other kinds, such as symbolic data or album art. However, since one cannot in practice assume that one will have *a priori* identifying information about a given musical instance, one must therefore have a way of acquiring the necessary data without presupposing that one begins with this identifying knowledge.

The solution to this problem may actually be closer than one might think. Let us consider, for example, a scenario where one is given a set of audio recordings to classify, but where one does not have any other additional explicit information about the recordings. One could begin by using fingerprinting technology such as MusicBrainz[286] to acquire metadata such as Title and Artist Name from the audio. This metadata could then be used to extract cultural features from the Internet using software such as jWebMiner. It would also be possible to use this metadata to automatically download information such as lyrics, album art and music videos using web services offered by, respectively, LyricWiki,[287] Amazon[288] and YouTube,[289] for example. Indeed, there are also a number of web services that could be used to access the audio itself using identifying metadata if one is in fact originally provided with this metadata rather than the audio.

Although automatically acquiring a symbolic version of each audio recording is not quite as simple as accessing the other kinds of musical information, advances in automatic music transcription technology are bringing the state of the art ever closer to the point where symbolic recordings can be automatically transcribed from audio with error rates that are low enough at least for the purposes of feature extraction, if not actual musical performance. Technologies such as Celemony Direct Note Access,[290] for example, seem to offer the promise of effective transcription algorithms, although they remain to be proven in a research context.

Of course, there can be certain limitations with solutions of the type discussed above, in terms of both accuracy and robustness to services going off-line. However, this example does illustrate a methodology that could be fruitful to pursue further if experiments of the type discussed in Section 10.3.9 do in fact indicate that combining features extracted from different types of musical data does have a significant positive impact on classification performance. If this is the case, then an important future step in the jMIR project will be the implementation of a framework for automatically acquiring and extracting features from a variety of different types of musical data.

[286] musicbrainz.org
[287] lyrics.wikia.com
[288] aws.amazon.com
[289] code.google.com/apis/youtube/overview.html
[290] www.celemony.com/cms/index.php?id=dna_qa

## *10.4 Using jMIR*

The jMIR project has SourceForge pages located at jmir.sourceforge.net and sourceforge.net/projects/jmir/. Each of the jMIR components may be downloaded from SourceForge, including both fully compiled and ready to use Java bytecode as well as the source code itself. Extensive documentation is also available at this site, including manuals, Javadoc APIs and copies of published papers. Basic installation and user instructions are also packaged with each component. All of the jMIR code is fully open-source, and is distributed under a GNU General Public License.[291]

As noted previously, jMIR is intended not just as a set of applications to be used as is, but also as a framework to be expanded and a platform for building new technologies. Contributions and improvements to jMIR from other researchers and developers are appreciated and encouraged.

[291] www.gnu.org/licenses/gpl.html

# 11. Bibliography

Aarden, B., and D. Huron. 2001. Mapping European folksong: Geographical localization of musical features. *Computing in Musicology* 12: 169–83.

Abdi, H., D. Valentin, and B. Edelman. 1999. *Neural Networks*. Thousand Oaks, CA: Sage Publications.

Adams, C. 1976. Melodic contour typology. *Ethnomusicology* 20 (2): 179–215.

Adeli, H., and S. L. Hung. 1995. *Machine learning: Neural networks, genetic algorithms and fuzzy systems*. New York: John Wiley & Sons.

Agrawal, R., and R. Srikant. 1996. Mining sequential patterns: Generalizations and performance improvements. *Proceedings of the International Conference on Extending Database Technology*. 3–17.

von Ahn, L., and L. Dabbish. 2004. Labeling images with a computer game. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 319–26.

Alpaydin, E. 2004. *Introduction to machine learning*. Cambridge, MA: MIT Press.

Amatriain, X., P. Arumi, and M. Ramirez. 2002 CLAM: Yet another library for audio and music processing? *Proceedings of the ACM Conference on Object Oriented Programming, Systems, and Applications*. 22–3.

Amatriain, X., and A. Pau. 2005. Developing cross-platform audio and music applications with the CLAM framework. *Proceedings of the International Computer Music Conference*. 403–10.

Anagnostopoulou, C., and G. Westermann. 1997. Classification in music: A computational model for paradigmatic analysis. *Proceedings of the International Computer Music Conference.* 125–8.

Anderson, C. 2006. *The long tail: Why the future of business is selling less of more*. New York, NY: Hyperion.

Anderson, S. 2009. *Sten's Blog >> Music Explorer FX*. Retrieved 14 December 2009, from http://blogs.citytechinc.com/sanderson/?p=105.

Arumi, P., and X. Amatriain. 2005. CLAM, an object oriented framework for audio and music. *Proceedings of the International Linux Audio Conference.*

Ash, T. 1989. Dynamic node creation in backpropagation networks. *Connection Science* 1: 365–75.

Ashby, F. G. 1989. Stochastic general recognition theory. In *Human information processing: Measures, mechanisms and models,* eds. D. Vickers and P. L. Smith, 435–57. Amsterdam: Elsevier.

Ashby, F. G. 1992. Multidimensional models of categorization. In *Multidimensional models of perception and cognition,* ed. F. G. Ashby, 449–84. Hillsdale, NJ: Lawrence Erlbaum Associates.

Ashby, F. G., and W. T. Maddox. 1990. Integrating information from separable psychological dimensions. *Journal of Experimental Psychology: Human Perception and Performance* 16: 598–612.

Aucouturier, J. J., and F. Pachet. 2003. Representing musical genre: A state of the art. *Journal of New Music Research* 32 (1): 83–93.

Aucouturier, J. J., and F. Pachet. 2004. Improving timbre similarity: How high is the sky? *Journal of Negative Results in Speech and Audio Sciences:* 1 (1).

Aucouturier, J. J., and F. Pachet. 2007. Signal + context = better. *Proceedings of the International Conference on Music Information Retrieval*. 425–30.

Backer, E., and P. Van Kranenburg. 2005. On musical stylometry: A pattern recognition approach. *Pattern Recognition Letters* 26: 299–309.

Barutcuoglu, Z., and C. DeCoro. 2006. Hierarchical shape classification using Bayesian aggregation. *IEEE International Conference on Shape Modelling and Applications*. 44–8.

Basili, R., A. Serafini, and A. Stellato. 2004. Classification of musical genre: A machine learning approach. *Proceedings of the International Conference on Music Information Retrieval.* 505–8.

Baumann, S., and O. Hummel. 2003. Using cultural metadata for artist recommendations. *Proceedings of the International Conference on WEB Delivery of Music*. 138–41.

Baumann, S., A. Klüter, and M. Norlien. 2002. Using natural language and audio analysis for a human-oriented MIR system. *Proceedings of the International Conference on WEB Delivery of Music*.

Bellaachia, A., and E. Jimenez. 2009. Exploring performance-based music attributes for stylometric analysis. *World Academy of Science, Engineering and Technology* 55: 509–11.

Bengio, Y. 1996. *Neural networks for speech and sequence recognition*. London: International Thomson Computer Press.

Berenzweig, A., B. Logan, D. P. W. Ellis, and B. Whitman. 2004. A large-scale evaluation of acoustic and subjective music-similarity measures. *Computer Music Journal* 28 (2): 63–76.

Bergeron, M., and D. Conklin. 2008. Structured polyphonic patterns. *Proceedings of the International Conference on Music Information Retrieval.* 69–74.

Bergstra, J., N. Casagrande, D. Erhan, D. Eck, and B. Kegl. 2007. Aggregate features and AdaBoost for music classification. *Machine Learning* 65 (2–3): 473–84.

Bergstra, J., A. Lacoste, and D. Eck. 2006. Predicting genre labels for artists using freedb. *Proceedings of the International Conference on Music Information Retrieval*. 85–8.

Bernstein, A., F. Provost, and S. Hill. 2005. Toward intelligent assistance for a data mining process: An ontology-based approach for cost-sensitive classification. *IEEE Transactions on Knowledge and Data Engineering* 17 (4): 503–18.

Bhattacharya, B., K. Mukherjee, and G. Toussaint. 2005. Geometric decision rules for high dimensions. *Proceedings of the 55th Session of the International Statistics Institute*.

Bigand, E., and B. Tillmann. 1996. Does formal musical structure affect perception of musical expressiveness? *Psychology of Music* 24 (1): 3–17.

Bishop, C. M. 2007. *Pattern recognition and machine learning.* New York: Springer.

Blackburn, S., and D. De Roure. 1998. Musical part classification in content based systems. *Proceedings of ACM Multimedia*. 361–8.

Blake, C., and C. Merz. 1998. *UCI repository of machine learning databases*. Retrieved April 13, 2005, from www.ics.uci.edu/~mlearn/MLRepository.html. University of California, Irvine, Department of Information and Computer Sciences.

Boer, V. d., M. v. Someren, and B. J. Wielina. 2006. Extracting instances of relations from web documents using redundancy. *Proceedings of the European Semantic Web Conference.*

Boersma, P., and D. Weenink. 2009. Praat: Doing phonetics by computer (Version 5.1.05). *Web page*. Retrieved 11 December 2009, from http://www.praat.org.

Bogert, B., M. J. R. Healy, and J. W. Tukey. 1963. The quefrency alanysis of time series for echoes: Cepstrum, pseudoautocovariance, cross-cepstrum, and saphe cracking. *Proceedings of the Symposium on Time Series Analysis*. 209–43.

Brackett, D. 1995. *Interpreting popular music.* New York: Cambridge University Press.

Brackett, D. 2002. *(In search of) musical meaning: Genres, categories and crossover*. London: Arnold.

Bray, S., and G. Tzanetakis. 2005. Distributed audio feature extraction for music. *Proceedings of the International Conference on Music Information Retrieval.* 434–7.

Brazdil, P., C. Soares, and J. Costa. 2003. Ranking learning algorithms. *Machine Learning* 50 (3): 251–77.

Bregman, A. S. 1990. *Auditory scene analysis*. Cambridge, MA: MIT Press.

Brill, E. 1992. A simple rule-based part-of-speech tagger. *Proceedings of the Conference on Applied Natural Language Processing.* 152–5.

Brin, S., and L. Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the International Conference on World Wide Web*. 107–17.

Briscoe, G., and T. Caelli. 1996. *A compendium of machine learning, Volume 1: Symbolic machine learning*. Norwood, NJ: Ablex Publishing.

Brooks, L. 1978. Nonanalytic concept formation and memory for instances. In *Cognition and categorization,* eds. E. Rosch and B. B. Lloyd, 169–215. Hillsdale, NJ: Erlbaum.

Brown, J. C. 1993 Determination of meter of musical scores by autocorrelation. *Journal of the Acoustical Society of America* 94 (4): 1953–7.

Bruha, I. 2001. Pre- and post-processing in machine learning and data mining. In *Machine learning and its applications: Advanced lectures*, eds. G. Paliouras, V. Karkaletsis, and C. D. Spyropoulos, 258–66. New York: Springer.

Burred J. J., C. E. Cella, G. Peeters, A. Röbel, and D. Schwarz. 2008. Using the SDIF sound description interchange format for audio features. *Proceedings of the International Conference on Music Information Retrieval.* 427–32.

Byrd, D., and T. Crawford. 2001. Problems of music information retrieval in the real world. *Information Processing & Management* 38 (2): 249–72.

Calishain, T., and R. Dornfest. 2003. *Google hacks.* London: O'Reilly.

Cambouropoulos, E. 2001. Melodic cue abstraction, similarity, and category formation: A formal model. *Music Perception* 18 (3): 347–70.

Cannam, C., C. Landone, M. Sandler, and J. P. Bello. 2006. The Sonic Visualiser: A visualisation platform for semantic descriptors from musical signals. *Proceedings of the International Conference on Music Information Retrieval*. 324–7.

Cano, P., and M. Koppenberger. 2004. The emergence of complex network patterns in music artist networks. *Proceedings of the International Conference on Music Information Retrieval.* 466–9.

Cantú-Paz, E. 2000. *Efficient and accurate parallel genetic algorithms*. Boston: Kluwer Academic Publishers.

Carey, M. J., E. S. Parris, and H. Lloyd-Thomas. 1999. A comparison of features for speech, music discrimination. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. 149–52.

Casey, M. A., R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. 2008. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE 96* (4): 668–96.

Celma, O., M. Ramírez, and P. Herrera. 2005. Foafing the music: A music recommendation system based on RSS feeds and user preferences. *Proceedings of the International Conference on Music Information Retrieval.* 464–7.

Chai, W., and B. Vercoe. 2001. Folk music classification using hidden Markov models. *Proceedings of the International Conference on Artificial Intelligence*.

Cimiano, P., and S. Staab. 2004. Learning by Googling. *SIGKDD Explorations Newsletter* 6 (2): 24–33.

Cohen, M. M., and D. W. Massaro. 1992. On the similarity of categorization models. In *Multidimensional models of perception and cognition,* ed. F. G. Ashby, 395–448. Hillsdale, NJ: Lawrence Erlbaum Associates.

Cohen, W. W., and W. Fan. 2000. Web-collaborative filtering: Recommending music by crawling the web. *WWW 9 / Computer Networks* 33 (1–6): 685–98.

Collobert, R., S. Bengio, and J. Mariethoz. 2002. Torch: A modular machine learning software library. *Technical Report IDIAP-RR 02-46*. Idiap Research Institute, Switzerland.

Conklin, D., and M. Bergeron. 2008. Feature set patterns in music. *Computer Music Journal* 32 (1): 60–70.

Cook, N. 1987. *A guide to musical analysis*. London: J. M. Dent & Sons.

Cooper, G., and L. B. Meyer. 1960. *The rhythmic structure of music.* Chicago, IL: University of Chicago Press.

Cope, D. 1991a.Computer simulations of musical style. *Proceedings of the Computers in Music Research Conference*. 15–7.

Cope, D. 1991b. *Computers and musical style*. Madison, WI: A-R Editions.

Cope, D. 1996. *Experiments in musical intelligence*. Madison, WI: A-R Editions.

Crump, M. 2002. A principle components approach to the perception of musical style. *Honours Thesis*. University of Lethbridge, Canada.

Cumming, J. E. 1999. *The motet in the age of Du Fay*. Cambridge, UK: Cambridge University Press.

Dannenberg, R. B. 1993. Music representation issues, techniques, and systems. *Computer Music Journal* 17 (3): 20–30.

Dannenberg, R. B., and N. Hu. 2002. Pattern discovery techniques for music audio. *Proceedings of the International Symposium on Music Information Retrieval*. 63–70.

Dannenberg, R. B., B. Thom, and D. Watson. 1997. A machine learning approach to musical style recognition. *Proceedings of the International Computer Music Conference*. 344–7.

DeCoro, C., Z. Barutcuoglu, and R. Fiebrink. 2007. Bayesian aggregation for hierarchical genre classification. *Proceedings of the International Conference on Music Information Retrieval.* 77–80.

Deglise, F. 2007. Les pirates peuvent dormir tranquilles. *Le Devoir*, 8 November 2007.

Deliege, I. 1996. Cue abstraction as a component of categorization processes in music listening. *Psychology of Music* 24 (2): 131–56.

Deliege, I. 2001a. Prototype effects in music listening: An empirical approach to the notion of imprint. *Music Perception* 18 (3): 371–407.

Deliege, I. 2001b. Similarity perception <-> Categorization <-> Cue abstraction. *Music Perception* 18 (3): 233–43.

Demsar, J., B. Zupan, G. Leban, and T. Curk. 2004. Orange: From experimental machine learning to interactive data mining. *Proceedings of the European Conference on Principles and Practice of Knowledge Discovery in Databases*. 537–9.

Deutsch, D. 1999. The processing of pitch combinations. In *The psychology of music,* ed. D. Deutsch, 349–412. New York: Academic Press.

Dhanaraj, R., and B. Logan. 2005. Automatic prediction of hit songs. *Proceedings of the International Conference on Music Information Retrieval.* 488–91.

Dietterich, T. G. 2000. Ensemble methods in machine learning. In *Multiple classifier systems,* eds., J. Kittler, and F. Roli, 1–15. New York: Springer.

DiLauro, T., G. S. Choudhury, M. Patton, and J. W. Warner. 2001. Automated name authority control and enhanced searching. *D-Lib Magazine* 7 (4).

Downie, J. S. 2003. Music information retrieval. In *Annual review of information science and technology* 37, ed. B. Cronin, 295–340. Medford, NJ: Information Today.

Downie, J. S. 2005. MIREX 2005 contest results. Available on-line at http://www.music-ir.org/evaluation/mirex-results. Retrieved 9 June 2009.

Downie, J. S. 2006. The Music Information Retrieval Evaluation eXchange (MIREX). *D-Lib Magazine* 12 (12).

Downie, J. S., A. Ehmann, and D. Tcheng. 2005. Music-to-knowledge (M2K): A prototyping and evaluation environment for music information retrieval research. *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval.* 676.

Downie, J. S., J. Futrelle, and D. Tcheng. 2004. The international music information retrieval systems evaluation laboratory: Governance, access and security. *Proceedings of the International Conference on Music Information Retrieval*. 9–14.

Downie, J. S., K. West, A. Ehmann, and E. Vincent. 2005. The 2005 Music Information Retrieval Evaluation eXchange (MIREX 2005): Preliminary overview. *Proceedings of the International Conference on Music Information Retrieval.* 320–3.

Duda, R. O., P. E. Hart, and D. G. Stork. 2001. *Pattern classification.* New York: John Wiley & Sons Inc.

Duff, D., ed. 2000. *Modern genre theory*. New York: Longman.

Dunn, J. W., and M. Notess. 2002. Variations 2: The Indiana University digital music library project. *Proceedings o the Digital Library Federation Fall Forum*.

Dzeroski, S., L. Todorovski, and H. Blockeel. 2004. Relational ranking with predictive clustering trees. *Proceedings of the International Workshop on Active Mining*. 9–15.

Eaton, J. W., and J. B. Rawlings. 2003. Ten years of Octave—Recent developments and plans for the future. *Proceedings of the International Workshop on Distributed Statistical Computing*.

Eck, D., T. Bertin-Mahieux, and P. Lamere. 2007. Autotagging music using supervised machine learning. *Proceedings of the International Conference on Music Information Retrieval.* 367–8.

Eerola, T., T. Järvinen, J. Louhivuori, and P. Toiviainen. 2001. Statistical features and perceived similarity of folk melodies. *Music Perception* 18 (3): 275–96.

Eerola, T., and P. Toiviainen. 2004. MIR in Matlab: The MIDI Toolbox. *Proceedings of the International Conference on Music Information Retrieval*. 22–7.

Ellis, D. P. W., B. Whitman, A. Berenzweig, and S. Lawrence. 2002. The quest for ground truth in musical artist similarity. *Proceedings of the International Conference on Music Information Retrieval*. 170–7.

Ennis, D. M. 1992. Modeling similarity and identification when there are momentary fluctuations in psychological magnitudes. In *Multidimensional models of perception and cognition,* ed. F. G. Ashby, 279–98. Hillsdale, NJ: Lawrence Erlbaum Associates.

Epstein, J. 1996. *Growing artificial societies: Social science from the bottom up.* Cambridge, MA: MIT Press.

Eronen, A. 2001. Comparison of features for musical instrument recognition. *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. 753–6.

Essid, S., G. Richard, and B. David. 2004. Musical instrument recognition based on class pairwise feature selection. *Proceedings of the International Conference on Music Information Retrieval*. 560–8.

Evans, D., J. L. Klavans, and N. Wacholder. 2000. Document processing with LinkIT. *Proceedings of RIAO*.

Evans, D., and C. Zhai. 1996. Noun-phrase analysis in unrestricted text for information retrieval. *Proceedings of the Meeting of the Associations for Computational Linguistics*. 17–24.

Fabbri, F. 1981. A theory of musical genres: Two applications. In *Popular music perspectives,* eds. D. Huron, and P. Tagg, 52–81. Göteborg: IASPM.

Fabbri, F. 1982. What kind of music? *Popular Music* 2: 131–43.

Fabbri, F. 1999. Browsing music spaces: Categories and the musical mind. *Proceedings of the IASPM Conference*.

Fausett, L. 1994. *Fundamentals of neural networks: Architectures, algorithms and applications*. Englewood Cliffs, NJ: Prentice Hall.

Fiebrink, R., C. McKay, and I. Fujinaga. 2005. Combining D2K and JGAP for efficient feature weighting for classification tasks in music information retrieval. *Proceedings of the International Conference on Music Information Retrieval*. 510–3.

Fiebrink, R., and I. Fujinaga. 2006. Feature selection pitfalls and music classification. *Proceedings of the International Conference on Music Information Retrieval*. 340–1.

Fiebrink, R., G. Wang, and P. Cook. 2008a. Foundations for on-the-fly learning in the ChucK programming language. *Proceedings of the International Computer Music Conference*.

Fiebrink, R., G. Wang, and P. Cook. 2008b. Support for MIR prototyping and real-time applications in the ChucK programming language. *Proceedings of the International Conference on Music Information Retrieval*. 153–8.

Fingerhut, M. 2004. Music information retrieval, or how to search for (and maybe find) music and do away with incipits. Presented at the *IAML-IASA Congress*.

Flexer, A. 2007. A closer look on artist filters for musical genre classification. *Proceedings of the International Conference on Music Information Retrieval*. 341–4.

Fodor, J. A. 1983. *Modularity of mind: An essay on faculty psychology.* Cambridge, MA: MIT Press.

Foote, J. 1999a. An overview of audio information retrieval. *Multimedia Systems* 7 (1): 2–10.

Foote, J. 1999b. Methods for the automatic analysis of music and audio. *FXPAL Technical Report* FXPAL-TR-99-038. FXPAL. Palo Alto, CA.

Freed, A. 2006. Music MetaData quality: A multiyear case study using the music of Skip James. *Proceedings of the Audio Engineering Society Convention*.

French, J. C., and D. B. Hauver. 2001. Flycasting: On the fly broadcasting. *Proceedings of the WedelMusic Conference*.

Freund, Y., and R. E. Schapire. 1996. Experiments with a new boosting algorithm. *Proceedings of the International Conference on Machine Learning*. 148–56.

Frith, S. 1996. *Performing rites: On the value of popular music*. Cambridge, MA: Harvard University Press.

Fürnkranz, J., J. Petrak, P. Brazdil, and C. Soares. 2002. On the use of fast subsampling estimates for algorithm recommendation. *Technical Report TR-2002-36*. The Austrian Research Institute for Artificial Intelligence, Vienna.

Fujinaga, I. 1996. Exemplar-based learning in adaptive optical music recognition system. *Proceedings of the International Computer Music Conference*. 55–6.

Fujinaga, I. 1997. Adaptive optical music recognition. *Ph.D. Dissertation.* McGill University, Canada.

Fujinaga, I. 1998. Machine recognition of timbre using steady-state tone of acoustic musical instruments. *Proceedings of the International Computer Music Conference*. 207–10.

Fujinaga, I., and D. McEnnis. 2006. On-demand Metadata Extraction Network. *Proceedings of the Joint Conference on Digital Libraries*.

Fukunaga, K. 1972. *Introduction to statistical pattern recognition*. New York: Academic Press.

Gabura, A. J. 1965. Computer analysis of musical style. *Proceedings of the ACM National Conference.* 303–14.

Gallant, S. I. 1993. *Neural network learning and expert systems*. Cambridge, MA: MIT Press.

Gardner, H. 1973. Children's sensitivity to musical styles. *Merrill-Palmer Quarterly* 19: 67–77.

Geleijnse, G., and J. Korst. 2006a. Learning effective surface text patterns for information extraction. *Proceedings of the EACL Workshop on Adaptive Text Extraction and Mining*. 1–8.

Geleijnse, G., and J. Korst. 2006b. Web-based artist categorization. *Proceedings of the International Conference on Music Information Retrieval*. 266–71.

Geleijnse, G., and J. Korst. 2007. Tool Play Live: Dealing with ambiguity in artist similarity mining from the web. *Proceedings of the International Conference on Music Information Retrieval*. 119–20.

Geleijnse, G., M. Schedl, and P. Knees. 2007. The quest for ground truth in musical artist tagging in the social web era. *Proceedings of the International Conference on Music Information Retrieval*. 525–30.

Gentle, J. E. 2009. *Computational statistics*. New York: Springer.

Gingras, B., and I. Knopke. 2005. Evaluation of voice-leading and harmonic rules of J. S. Bach's chorales. *Proceedings of the Conference on Interdisciplinary Musicology*.

Giraud-Carrier, C., and J. Keller. 2002. Meta-learning. In *Dealing with the data flood: Mining data, text and multimedia*, ed. J. Meij, 832–44. The Hague, NL: STT/Beweton.

Gold, B., and N. Morgan. 2000. *Speech and audio signal processing: Processing and perception of speech and music.* Toronto: Wiley.

Goto, M. 2003. A chorus-section detecting method for musical audio signals. *Proceedings of International Conference on Acoustics, Speech and Signal Processing.* 437–40.

Goto, M. 2006. AIST annotation for the RWC music database. *Proceedings of the International Conference on Music Information Retrieval*. 359–60.

Goto, M., H. Hashiguchi, T. Nishimura, and R. Oka. 2002. RWC music database: Popular, classical and jazz music data-bases. *Proceedings of the International Conference on Music Information Retrieval*. 287–8.

Granitto, P. M., P. F. Verdes, H. D. Navone, and H. A. Ceccatto. 2002. Aggregation algorithms for neural network ensemble construction. *Proceedings of the Brazilian Symposium on Neural Networks*. 178–83.

Grant, B. K., ed. 2003. *Film genre reader III.* Austin, TX: University of Texas Press.

Guess, M. 2001. MusicXML for notation and analysis. In *The virtual score: Representation, retrieval, restoration,* eds. W. B. Hewlett, and E. Selfridge-Field, 113–24. Cambridge, MA: MIT Press.

Hallinan, J. 2001. Feature selection and classification in the diagnosis of cervical cancer. In *The practical handbook of genetic algorithms applications,* ed. L. Chambers, 167–202. New York: Chapman & Hall.

Hansen, L. K., and P. Salamon. 1990. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Learning* 12 (10): 993–1001.

Harris, F. J. 1978. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE* 66 (1): 51–83.

Hastie, T., R. Tibshirani, and J. Friedman. 2001. *The elements of statistical learning*. New York: Springer.

van der Heijden, F., R. P. W. Duin, D. de Ridder, and D. M. J. Tax. 2004. *Classification, parameter estimation and state estimation: An engineering approach using MATLAB*. New York: Wiley.

Herlocker, J. L., J. A. Konstan, and J. Riedl. 2000. Explaining collaborative filtering recommendations. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. 241–50.

Herlocker, J. L., J. A. Konstan, L. G. Terveen, and J. T. Riedl. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems* 22 (1): 5–53.

Herrera, P., G. Peeters, and S. Dubnov. 2003. Automatic classification of musical instrument sounds. *Journal of New Music Research* 32 (1): 3–21.

Hillewaere, R., B. Manderick, and D. Conklin. 2009. Global feature versus event models for fold song classification. *Proceedings of the International Society for Music Information Retrieval Conference.* 729–33.

Hoffman-Engl, L. 2001. Towards a cognitive model of melodic similarity. *Proceedings of the International Symposium on Music Information Retrieval*. 143–51.

Holland, J. H. 1975. *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press.

Homburg, H., I. Mierswa, B. Moller, K. Morik, and M. Wurst. 2005. A benchmark dataset for audio classification and clustering. *Proceedings of the International Conference on Music Information Retrieval*. 528–31.

Hoos, H. H., K. A. Hamel, K. Renz, and J. Kilian. 2001. Representing score-level music using the GUIDO music-notation format. In *The virtual score: Representation, retrieval, restoration,* eds. W. B. Hewlett, and E. Selfridge-Field. Cambridge, MA: MIT Press.

Hortaçsu, N., and H. G. Tekman. 2002. Aspects of stylistic knowledge: What are different styles like and why do we listen to them? *Psychology of Music* 30 (1): 28–47.

Hothker, K., D. Hornel, and C. Anagnostopoulou. 2001. Investigating the influence of representations and algorithms in music classification. *Computers and the Humanities* 35 (1): 65–79.

Hsu, J. L., C. C. Liu, and A. L. P. Chen. 2001. Discovering nontrivial repeating patterns in music data. *IEEE Transactions on Multimedia* 3 (3): 311–25.

Hu, X., M. Bay, and J. S. Downie. 2007. Creating a simplified music mood classification ground-truth set. *Proceedings of the International Conference on Music Information Retrieval.* 309–10.

Hu, X, J. S. Downie, and A. F. Ehman. 2009. Lyric text mining in music mood classification. *Proceedings of the International Society for Music Information Retrieval Conference*. 411–6.

Huotilainen, M., J. Louhivuori, R. Näätänen, M. Saher, M. Tervaniemi, and P. Toiviainen. 1998. Timbre similarity: Convergence of neural, behavioral, and computational approaches. *Music Perception* 16 (2): 223–41.

Huron, D. 1997. Humdrum and Kern: Selective feature encoding. In *Beyond MIDI: The handbook of musical codes,* ed. E. Selfridge-Field. Cambridge, 375–401. MA: MIT Press.

Huron, D. 1999. The new empiricism: Systematic musicology in a postmodern age. 1999 *Ernst Bloch Lecture*. University of California, Berkeley.

Huron, D. 2002. Music information processing using the Humdrum toolkit: Concepts, examples, and lessons. *Computer Music Journal* 26 (2): 11–26.

Hussein, F., R. Ward, and N. Kharma. 2001. Genetic algorithms for feature selection and weighting, a review and study. *International Conference on Document Analysis and Recognition.* 1240–4.

Ihaka, R., and R. Gentleman. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5: 299–314.

Jagannathan, V., R. Dodhiawala, and L. S. Baum, eds. 1989. *Blackboard architectures and applications*. New York: Academic Press.

Jain, A. K., R. P. W. Duin, and J. Mao. 1999. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (1): 4–37.

Jain, A. K., and D. Zongker. 1997. Feature selection: Evaluation, application and small sample performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19 (2): 153–8.

James, M. 1985. *Classification algorithms*. London: William Collins and Sons.

Jayant, N. S., and P. Noll. 1984. *Digital coding of waveforms: Principles and applications to speech and video*. Upper Saddle River, NJ: Prentice Hall.

Jennings, D. 2007. *Net, blogs and rock 'n' roll: How digital discovery works and what it means for consumers, creators and culture.* Boston: Nicholas Brealey.

Jensen, K. 1999. Timbre models of musical sounds. *Ph.D. Thesis.* University of Copenhagen, Denmark.

Jones, M. C., J. S. Downie, and A. F. Ehmann. 2007. Human similarity judgments: implications for t the design of formal evaluations. *Proceedings of the International Conference on Music Information Retrieval.* 539–42.

Joyce, J. 2006. Pandora and the Music Genome Project. *Scientific Computing* 23 (10): 40–1.

Kalousis, A., J. Gama, and M. Hilario. 2004. On data and algorithms: Understanding inductive performance. *Machine Learning* 54 (3): 275–312.

Kandel, A., and H. Bunke, eds. 2002. *Hybrid methods in pattern recognition*. London: World Scientific.

Karydis, I., A. Nanopoulos, and Y. Manolopoulos. 2006. Symbolic musical genre classification based on repeating patterns. *Proceedings of the ACM Workshop on Audio and Music Computing Multimedia*. 53–8.

Kashino, K., and H. Murase. 1999. A sound source identification system for ensemble music based on template adaptation and music stream extraction. *Speech Communication* 27 (3): 337–49.

Katayose, H., and S. Inokuchi. 1989. The KANSEI music system. *Computer Music Journal* 13 (4): 72–7.

Kendall, M. G., and A. Stuart. 1973. *The advanced theory of statistics. Volume II: Inference and Relationship*. London: Griffin.

Khoussainov, R., X. Zuo, and N. Kushmerick. 2004. Grid-enabled Weka: A toolkit for machine learning on the grid. *ERCIM News* 59.

Kirby, M. 2001. *Geometric data analysis: An empirical approach to dimensionality reduction and the study of patterns*. New York: John Wiley & Sons.

Kirlin, P. B., and P. E. Utgoff. 2005. VoiSe: Learning to segregate voices in explicit and implicit polyphony. *Proceedings of the International Conference on Music Information Retrieval.* 552–7.

Kitahara, T. 2008. A unified and extensible framework for developing music information processing systems. *Unpublished Manuscript.*

Kittler, J. 2000. A framework for classifier fusion: Is it still needed? *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*. 45–56.

Klapuri, A. 1999. Sound onset detection by applying psychoacoustic knowledge. *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing.*

Klapuri, A., and M. Davy, eds. 2006. Signal processing methods for music transcription. New York: Springer.

Knees, P., E. Pampalk, and G. Widmer. 2004. Artist classification with web-based data. *Proceedings of the International Conference on Music Information Retrieval.* 517–24.

Knees, P., M. Schedl, T. Pohle, and G. Widmer. 2006. An innovative three-dimensional user interface for exploring music collections enriched with meta-information from the web. *Proceedings of the ACM International Conference on Multimedia.* 17–24.

Knopke, I. 2008. The PerlHumdrum and PerlLilypond toolkits for symbolic music information retrieval. *Proceedings of the International Conference on Music Information Retrieval.* 147–52.

Koniari D., S. Predazzer, and M. Mélen. 2001. Categorization and schematization processes used in music perception by 10- to 11-Year-Old Children. *Music Perception* 18 (3): 297–324.

Kotek. B. 1998. Soft computing-based recognition of musical sounds. In *Rough sets in knowledge discovery*, eds. L. Polkowski, and A. Skowron, 193–213. Heidelberg: Physica-Verlag.

Kotsiantis, S., and P. Pintelas. 2004. Selective voting. *Proceedings of the International Conference on Intelligent Systems Design and Applications*. 397–402.

Krumhansl, C. L. 1978. Concerning the applicability of geometric models to similarity data: The interrelationship between similarity and spatial density. *Psychology Review* 85: 445–63.

Kuncheva, L. 2004. *Combining pattern classifiers*. Hoboken, NJ: Wiley.

Laine, P., and M. Kuuskankare. 1994. Genetic algorithms in musical style oriented generation. *Proceedings of the IEEE Conference on Evolutionary Computation*. 858–62.

Lakoff, G. 1987. *Women, fire, and dangerous things: What categories reveal about the mind.* Chicago, IL: University of Chicago Press.

Lamere, P., and D. Eck. 2007. Using 3D visualizations to explore and discover music. *Proceedings of the International Conference on Music Information Retrieval.* 173–4.

Lamont, A., and N. Dibben. 2001. Motivic structure and the perception of similarity. *Music Perception* 18 (3): 245–74.

Lartillot, O., S. Dubnov, G. Assayag, and G. Bejerano. 2001. Automatic modeling of musical style. *Proceedings of the International Computer Music Conference*. 447–54.

Lartillot, O., and P. Toiviainen. 2007. MIR in Matlab (II): A toolbox for musical feature extraction from audio. *Proceedings of the International Conference on Music Information Retrieval*. 127–30.

Lartillot, O., P. Toiviainen, and T. Eerola. 2008. A Matlab toolbox for music information retrieval. In *Data Aalysis, Machine Learning and Applications,* eds. C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker, 261–8. New York: Springer.

LaRue, J. 1992. *Guidelines for style analysis*. Warren, MI: Harmonie Park Press.

Law, E., L. von Ahn, R. Dannenberg, and M. Crawford. 2007. Tagatune: A game for music and sound annotation. *Proceedings of the International Conference on Music Information Retrieval*. 361–4.

Lee, J. H., and J. S. Downie. 2004. Survey of music information needs, uses, and seeking behaviours: Preliminary findings. *Proceedings of the International Conference on Music Information Retrieval*.

Leman, M. 1995. *Music and schema theory: Cognitive foundations of systematic musicology.* New York: Springer.

Lerch, A., G. Eisenberg, and K. Tanghe: 2005. FEAPI: A low level feature extraction plugin API. *Proceedings of the International Conference on Digital Audio Effects*.

Lerdahl, F., and R. Jackendoff. 1982. *A generative theory of tonal music*. Cambridge, MA: MIT Press.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10: 707–10.

Li, T., and M. Ogihara. 2003. Detecting emotion in music. *Proceedings of the International Symposium on Music Information Retrieval*. 239–40.

Lidy, T., A. Rauber, A. Pertusa, and J. M. Iñesta. 2007. Improving genre classification by combination of audio and symbolic descriptors using a transcription system. *Proceedings of the International Conference on Music Information Retrieval*. 61–6.

Lin, C. R., N. H. Liu, Y. H. Wu, and L. P. Chen. 2004. Music classification using significant repeating patterns. *Proceedings of Database Systems for Advanced Applications.* 506–18.

Lindner, G., and R. Studer. 1999. AST: Support for algorithm selection with a CBR approach. *Proceedings of the European Conference on the Principles of Data Mining and Knowledge Discovery*. 418–23.

Lippens, S, J. P. Martens, M. Leman, B. Baets, H. Meyer, and G. Tzanetakis. 2004. A comparison of human and automatic musical genre classification. *Proceedings of the IEEE International Conference on Audio, Speech and Signal Processing*.

Liu, D., L. Lu, and H. J. Zhang. 2003. Automatic mood detection from acoustic music data. *Proceedings of the International Symposium on Music Information Retrieval*. 81–7.

Llorà, X., B. Ács, L. S. Auvil, B. Capitanu, M. E. Welge, and D. E. Goldberg. 2008. Meandre: Semantic-driven data-intensive flows in the clouds. *Proceedings of the IEEE International Conference on eScience*. 238–45.

Lomax, A. 1968. *Folk song style and culture.* Washington, DC: American Association for the Advancement of Science.

Lowe, W., and S. McDonald. 2000. The direct route: Mediated priming in the semantic space. *Proceedings of the Annual Conference of the Cognitive Science Society*.

Maclin, R., and J. Shavlik. 1995. Combining the predictions of multiple classifiers: Using competitive learning to initialise neural networks. *Proceedings of the International Joint Conference on Artificial Intelligence.* 524–30.

MacMillan, K., M. Droettboom, and I. Fujinaga. 2001. Gamera: A structured document recognition application development environment. *Proceedings of the International Symposium on Music Information Retrieval*. 173–8.

Manaris B., J. Romero, P. Machado, D. Krehbiel, T. Hirzel, W. Pharr, and R. B. Davis. 2005. Zipf's law, music classification, and aesthetics. *Computer Music Journal* 25 (1): 55–69.

Mandel, M. I., and D. Ellis. 2007. A web-based game for collecting music metadata. *Proceedings of the International Conference on Music Information Retrieval*. 365–6.

Manning, C. D., and H. Schütze. 1999. *Foundations of statistical natural language processing*. Cambridge, MA: MIT Press.

Manuel, P. 1988. *Popular musics of the non-Western world*. Toronto: Oxford University Press.

Martin, K., and Y. Kim. 1998. Musical instrument identification: A pattern recognition approach. *Proceedings of the Acoustical Society of America*.

Maxwell, J. B., and A. Eigenfeldt. 2008 A music database and query system for recombinant composition. *Proceedings of the International Conference on Music Information Retrieval.* 75–80.

Mazzola, G., and O. Zahorka. 1994. The RUBATO performance workstation on NEXTSTEP. *Proceedings of the International Computer Music Conference*.

McAdams, S. 1999. Perspectives on the contribution of timbre to musical structure. *Computer Music Journal* 23 (3): 85–102.

McAdams, S., and D. Matzkin. 2003. The roots of musical variation in perceptual similarity and invariance. In *The cognitive neuroscience of music,* eds. I. Peretz, and R. Zatorre, 79–94. New York: Oxford University Press.

McClellan, J. H., R. W. Schafer, and M. A. Yoder. 1999. *DSP first: A multimedia approach.* Upper Saddle River, NJ: Prentice-Hall.

McEnnis, D. 2006. On-demand metadata extraction network (OMEN). *M.A. Thesis*. McGill University, Canada.

McEnnis, D., C. McKay, and I. Fujinaga. 2006a. jAudio: Additions and improvements. *Proceedings of the International Conference on Music Information Retrieval*. 385–6.

McEnnis, D., C. McKay, and I. Fujinaga. 2006b. Overview of OMEN. *Proceedings of the International Conference on Music Information Retrieval*. 7–12.

McEnnis, D., C. McKay, I. Fujinaga, and P. Depalle. 2005. jAudio: A feature extraction library. *Proceedings of the International Conference on Music Information Retrieval*. 600–3.

McKay, C. 2004. Automatic genre classification of MIDI recordings. *M.A. Thesis.* McGill University, Canada.

McKay, C. 2005. Approaches to overcoming problems in interactive musical performance systems. Presented at the *McGill Graduate Students Society Symposium*.

McKay, C. 2009. *ACE XML sample file appendix*. Retrieved 20 December 2009, from http://www.music.mcgill.ca/~cmckay/protected/ACE_XML_Sample_File_Appendix.pdf.

McKay, C., J. A. Burgoyne, J. Thompson, and I. Fujinaga. 2009. Using ACE XML 2.0 to store and share feature, instance and class data for musical classification. *Proceedings of the International Society for Music Information Retrieval Conference*. 303–8.

McKay, C., R. Fiebrink, D. McEnnis, B. Li, and I. Fujinaga. 2005. ACE: A framework for optimizing music classification. *Proceedings of the International Conference on Music Information Retrieval.* 42–9.

McKay, C., and I. Fujinaga. 2005a. Automatic music classification and the importance of instrument identification. *Proceedings of the Conference on Interdisciplinary Musicology.* CD-ROM.

McKay, C., and I. Fujinaga. 2005b. The Bodhidharma system and the results of the MIREX 2005 symbolic genre classification contest. Presented at the *International Conference on Music Information Retrieval*.

McKay, C., and I. Fujinaga. 2006a. jSymbolic: A feature extractor for MIDI files. *Proceedings of the International Computer Music Conference*. 302–5.

McKay, C., and I. Fujinaga. 2006b. Musical genre classification: Is it worth pursuing and how can it be improved? *Proceedings of the International Conference on Music Information Retrieval.* 101–6.

McKay, C., and I. Fujinaga. 2006c. Style-independent computer-assisted exploratory analysis of large music collections. Presented at the *Joint Meeting of the American Musicological Society and the Society for Music Theory*.

McKay, C., and I. Fujinaga. 2007a. jWebMiner: A web-based feature extractor. *Proceedings of the International Conference on Music Information Retrieval.* 113–4.

McKay, C., and I. Fujinaga. 2007b. Style-independent computer-assisted exploratory analysis of large music collections. *Journal of Interdisciplinary Music Studies* 1 (1): 63–85.

McKay, C., and I. Fujinaga. 2008. Combining features extracted from audio, symbolic and cultural sources. *Proceedings of the International Conference on Music Information Retrieval.* 597–602.

McKay, C., and I. Fujinaga. 2009a. Expressing musical features, class labels, ontologies and metadata using ACE XML 2.0. Submitted to *Structuring Music through Markup Language: Designs and Architectures,* ed. J. Steyn. Hershey: IGI.

McKay, C., and I. Fujinaga. 2009b. jMIR: Tools for automatic music classification. *Proceedings of the International Computer Music Conference*.

McKay, C., D. McEnnis, R. Fiebrink, and I. Fujinaga. 2005. ACE: A general-purpose classification ensemble optimization framework. *Proceedings of the International Computer Music Conference*. 161–4.

McKay, C., D. McEnnis, and I. Fujinaga. 2006. A large publicly accessible prototype audio database for music research. *Proceedings of the International Conference on Music Information Retrieval.* 160–3.

McKinney, M. F., and J. Breebaart. 2003. Features for audio and music classification. *Proceedings of the International Symposium on Music Information Retrieval.* 151–8.

McWilliams, R. 2005. Intra-judge consistency in ensemble performance adjudication. *Celebration of Voices: XV National Conference Proceedings*.

Mervis, C. B. 1980. Category structure and the development of categorization. In *Theoretical issues reading comprehension*, R. Spiro, B. C. Bruce, and W. F. Brewer eds. Hillsdale, NJ: Lawrence Erlbaum Associates.

Mervis, C. B., J. Catlin, and E. Rosch. 1976. Relationships among goodness-of-example, category norms, and word frequency. *Bulletin of the Psychonomic Society* 7: 283–4.

Meudic, B. 2003. Automatic pattern extraction from polyphonic MIDI files. *Proceedings of the Computer Music Modeling and Retrieval Conference.* 124–42.

Meyer, L. B. 1989. *Style and music: Theory, history and ideology*. Chicago: University of Chicago Press.

Michalski, R. S., I. Bratko, and M. Kubat, eds. 1999. *Machine learning and data mining: Methods and applications*. Toronto: John Wiley & Sons.

Michie, D., D. J. Spiegelhalter, and C. C. Taylor. 1994. *Machine learning, neural and statistical classification.* New York: Ellis Horwood.

Middleton, R. 1990. *Studying popular music*. Philadelphia: Open University Press.

Middleton, R. 2000. Popular music analysis and musicology: Bridging the gap. In *Reading pop: Approaches to textual analysis in popular music*, ed. R. Middleton, 104–21. New York: Oxford University Press.

MIDI Manufacturers Association. 2001. *Complete MIDI 1.0 detailed specification v96.1*. Los Angeles: International MIDI Association.

Mierswa, I., and K. Morik. 2005. Automatic feature extraction for classifying audio data. *Machine Learning Journal* 58: 127–49.

Mierswa, I., M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. 2006. YALE: Rapid prototyping for complex data mining tasks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 935–40.

Minaei-Bidgoli, B., G. Kortemeyer, and W. Punch. 2004. Optimizing classification ensembles via a genetic algorithm for a web-based educational system. *Proceedings of the International Workshop on Syntactical and Structural Pattern Recognition and Statistical Pattern Recognition.* 397–406.

Minda, J. P., and J. D. Smith. 2001. Prototypes in category learning: The effects of category size, category structure, and stimulus complexity. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 27 (3): 775–99.

Mitchell, T. M. 1997. *Machine learning*. New York: McGraw-Hill.

Moore, B. C. J. 2003. *An introduction to the psychology of hearing*. Boston: Academic Press.

Moscote Freire, A. 2007. Tuning into you: Personalized audio streaming services and their remediation of radio. *M. A. Thesis*. McGill University, Canada.

Motulsky, H. 1999. *Analyzing data with graphPad prism*. La Jolla, CA: GraphPad Software Inc.

Narmour, E. 1999. Hierarchical expectation and musical style. In *The psychology of music,* ed. D. Deutsch, 442–72. London: Academic Press.

Nattiez, J. J. 1990. *Music and discourse: Toward a semiology of music*. C. Abbate, trans. Princeton, NJ: Princeton University Press.

Negus, K. 1999. *Music genres and corporate cultures*. New York: Routledge.

Nettl, B. 1990. *Folk and traditional music of the western continents*. Englewood Cliffs, NJ: Prentice-Hall.

Nienhuys, H. W., and J. Nieuwenhuizen. 2003. LilyPond, a system for automated music engraving. *Proceedings of the XIV Colloquium on Musical Informatics*.

North, A. C., and D. J. Hargreaves. 1997. Liking for musical styles. *Music Scientae* 1 (1): 109–28.

Nosofsky, R. M. 1986. Attention, similarity, and the identification-categorization relationship. *Journal of Experimental Psychology: General* 115: 39–57.

Nosofsky, R. M., and S. R. Zaki. 2002. Exemplar and prototype models revisited: Response strategies, selective attention, and stimulus generalization. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 28 (5): 924–40.

Novello, A. and M. McKinney. 2007. Music similarity and perception in Western popular music. Presented at the *Society for Music Perception and Cognition Conference*.

Ockelford, A. 2004. On similarity, derivation and the cognition of musical structure. *Psychology of Music* 32 (1): 23–74.

Opitz, D., and R. Maclin. 1999. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research* 11: 169–98.

Oppenheim, A. V., R. W. Schafer, and J. R. Buck. 1999. *Discrete-time signal processing*. Upper Saddle River, NJ: Prentice Hall.

Pachet, F., J. J. Aucouturier, A. Burthe, A. Zils, and A. Beurive. 2006. The Cuidado Music Browser: An end-to-end electronic music distribution system. *Multimedia Tools and Applications* 30 (3): 331–49.

Pachet, F., and Z. Aymeric. 2004. Automatic extraction of music descriptors from acoustic signals. *Proceedings of the International Conference on Music Information Retrieval.* 353–6.

Pachet, F., and D. Cazaly. 2000. A taxonomy of musical genres. *Proceedings of the Content-Based Multimedia Information Access Conference.*

Pachet, F., G. Westermann, and D. Laigre. 2001. Musical data mining for electronic music distribution. *Proceedings of the International Conference on WEB Delivery of Music.* 101–6.

Pampalk, E., and M. Goto. 2007. MusicSun: A new approach to artist recommendation. *Proceedings of the International Conference on Music Information Retrieval.* 101–4.

Park, T. H. 2000. Salient feature extraction of musical instrument signals. *Master's Thesis.* Dartmouth College, USA.

Perrin, N. A. 1992. Uniting identification, similarity and preference: General recognition theory. In *Multidimensional models of perception and cognition,* ed. F. G. Ashby, 123–46. Hillsdale, NJ: Lawrence Erlbaum Associates.

Perrott, D., and R. O. Gjerdingen. 1999. Scanning the dial: An exploration of factors in the identification of musical style. *Research Notes*. Department of Music, Northwestern University, IL, USA.

Pestoni, F., J. Wolf, A. Habib, and A. Mueller. 2001. KARC: Radio research. *Proceedings of the WedelMusic Conference*.

Pfeiffer, S., and C. Parker. 2001. bewdy, Maaate!. Presented at the *Australian Linux Conference*.

Pin-Shan Chen, P. 1976. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems* 1 (1): 9–36.

Pohle, T., P. Knees, M. Schedl, and G. Widmer. 2007a. Building an interactive next-generation artist recommender based on automatically derived high-level concepts.

*Proceedings of the International Workshop on Content-Based Multimedia Indexing.* 336–43.

Pohle, T., P. Knees, M. Schedl, and G. Widmer. 2007b. Meaningfully browsing music services. *Proceedings of the International Conference on Music Information Retrieval.* 115–6.

Pollard-Gott, L. 1983. Emergence of thematic concepts in repeated listening to music. *Cognitive Psychology* 15: 66–94.

Ponce de León, P. J., and J. M. Iñesta. 2002. Musical style identification using self-organizing maps. *Proceedings of the International Conference on Web Delivery of Music*. 82–9.

Ponce de León, P. J., and J. M. Iñesta. 2004. Statistical description models for melody analysis and characterization. *Proceedings of the International Computer Music Conference.* 149–56.

Ponce de León, P. J., and J. M. Iñesta. 2007. A pattern recognition approach for music style identification using shallow statistical descriptors. *IEEE Transactions on Systems, Man and Cybernetics* 37 (2): 248–57.

Pope, S. T., F. Holm, and A. Kouznetsov. 2004. Feature extraction and database design for music software. *Proceedings of the International Computer Music Conference*. 596–603.

Pope, S. T., and A. Kouznetsov. 2004. FASTLab Music Analysis Kernel Library (FMAK). *FastLab Inc. Technical Report*.

Posner, M. I., and S. W. Keele. 1968. On the genesis of abstract ideas. *Journal of Experimental Medicine* 77: 353–63.

Posner, M. I., and S. W. Keele. 1970. Retention of abstract ideas. *Journal of Experimental Medicine* 83: 304–8.

Povel, O., and C. Giraud-Carrier. 2004. SwissAnalyst: Data mining without the entry ticket. *Proceedings of the TC12 First International Conference on Artificial Intelligence Applications and Innovations.* 393–406.

Pudil, P., F. Ferri, J. Novovicova, and J. Kittler. 1994. Floating search methods for feature selection with nonmonotonic criterion functions. *Proceedings of the IEEE International Conference on Pattern Recognition.* 279–83.

Punch, W., E. Goodman, M. Pei, L. Chia-Shun, P. Hovland, and R. Enbody. 1993. Further research on feature selection and classification using genetic algorithms. *Proceedings of the International Conference on Genetic Algorithms*. 557–64.

Quinlan, J. R. 1993. *C4.5: Programs for machine learning*. New York: Morgan Kaufmann Publishers.

Rabiner, L. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77 (2): 257–86.

Rabiner, L., and B. H. Juang. 1986. An introduction to hidden Markov models. *IEEE ASSP Magazine* 3 (1): 4–16.

Rabiner, L., and B. H. Juang. 1993. *Fundamentals of speech recognition.* Upper Saddle River, NJ: Prentice-Hall.

Raimond, Y. 2009. A distributed music information system. *Doctoral Dissertation*. Queen Mary, University of London, U.K.

Raimond, Y., S. Adbdallah, M. Sandler, and F. Giasson. 2007. The Music Ontology. *Proceedings of the International Conference on Music Information Retrieval.* 417–22.

Raimond, Y., and M. Sandler. 2008. A web of musical information. *Proceedings of the International Conference on Music Information Retrieval.* 263–28.

Raphael, C., and J. Stoddard. 2003. Harmonic analysis with probabilistic graphical models. *Proceedings of the International Symposium on Music Information Retrieval*. 177–81.

Reed, J., and C. H. Lee. 2007. A study on attribute-based taxonomy for music information retrieval. *Proceedings of the International Conference on Music Information Retrieval.* 485–90.

Reed, S. K. 1972. Pattern recognition and categorization. *Cognitive Psychology* 44: 522–45.

van Rees, R. 2003. Clarity in the usage of the terms ontology, taxonomy and classification. *CIB Report* 284: 432–9.

Reti, R. 1951. *The thematic process in music*. New York: Macmillan.

Rips, L. J., E. J. Shoben, and E. E. Smith. 1973. Semantic distance and the verification of semantic relationships. *Journal of Verbal Learning and Verbal Behavior* 12: 1–20.

Ritthoff, O., R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. 2001. YALE: Yet another machine learning environment. *Proceedings of the LLWA.*

Rizo, D., P. J. Ponce de León, C. Pérez-Sancho, A. Pertusa, and J. M. Iñesta. 2006. A pattern recognition approach for melody track selection in MIDI files. *Proceedings of the International Conference on Music Information Retrieval*. 61–6.

Rosch, E. 1973a. Natural categories. *Cognitive Psychology* 4: 328–50.

Rosch, E. 1973b. On the internal structure of perceptual and semantic categories. In *Cognitive development and the acquisition of language,* ed. T. E. Moore, 111–44. New York: Academic Press.

Rosch, E. 1975. Cognitive reference points. *Cognitive Psychology* 7: 532–47.

Rosch, E. 1978. Principles of categorization. In *Cognition and categorization,* eds. E. Rosch, and B. B. Lloyd, 27–48. Hillsdale, NJ: Erlbaum.

Rosch, E., C. Simpson, and R. S. Miller. 1976. Structural bases of typicality effects. *Journal of Experimental Psychology: Human Perception and Performance* 2: 491–502.

Rothstein, J. 1995. *MIDI: A comprehensive introduction*. Madison, WI: A-R Editions.

Rowe, R. 2001. *Machine musicianship*. Cambridge, MA: MIT Press.

Ruppin, A., and H. Yeshurun. 2006. MIDI music genre classification by invariant features. *Proceedings of the International Conference on Music Information Retrieval.* 397–9.

Russell, S., and P. Norvig. 2002. *Artificial intelligence: A modern approach*. Upper Saddle River, NJ: Prentice Hall.

Sakawa, M. 2002. *Genetic algorithms and fuzzy multiobjective optimization*. Norwell, MA: Kluwer Academic Publishers.

Sapp, C. S., Y. W. Liu, and E. Selfridge-Field. 2004. Search-effectiveness measures for symbolic music queries in very large databases. *Proceedings of the International Conference on Music Information Retrieval.* 266–73.

Sattath, S., and A. Tversky. 1977. Additive similarity trees. *Psychometrika* 42: 319–45.

Schedl, M., P. Knees, and G. Widmer. 2005a. A web-based approach to assessing artist similarity using co-occurrences. *Proceedings of the International Workshop on Content-Based Multimedia Indexing.*

Schedl, M., P. Knees, and G. Widmer. 2005b. Discovering and visualizing prototypical artists by web-based co-occurrence analysis. *Proceedings of the International Conference on Music Information Retrieval.* 21–8.

Schedl, M., P. Knees, and G. Widmer. 2006. Improving prototypical artist detection by penalizing exorbitant popularity. *Proceedings of the International Symposium on Computer Music Modeling and Retrieval*.

Schedl, M., T. Pohle, P. Knees, and G. Widmer. 2006. Assigning and visualizing music genres by web-based co-occurrence analysis. *Proceedings of the International Conference on Music Information Retrieval*. 260–5.

Schedl, M, G. Widmer, T. Pohle, and K. Seyerlehner. 2007. Web-based detection of music band members and line-up. *Proceedings of the International Conference on Music Information Retrieval*. 117–8.

Scheirer, E., and M. Slaney. 1997. Construction and evaluation of a robust multi-feature speech/music discriminator. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*.

Schütze, H. 1992. Dimensions of meaning. *Proceedings of Supercomputing*. 787–96.

Schwarz, D., and M. Wright. 2000. Extensions and applications of the SDIF Sound Description Interchange Format. *Proceedings of the International Computer Music Conference*. 481–4.

Selfridge-Field, E., ed. 1997. *Beyond MIDI: The handbook of musical codes*. Cambridge, MA: MIT Press.

Selfridge-Field, E. 1998. Conceptual and representational issues in melodic comparison. In *Melodic similarity: Concepts, procedures, and applications*, eds. W. B. Hewlett, and E. Selfridge-Field, 3–64. Cambridge, MA: MIT Press.

Shan, M. K., and F. F. Kuo. 2003. Music style mining and classification by melody. *IEICE Transactions on Information and Systems* E86-D (3): 655–9.

Shardanand, U., and P. Maes. 1995. Social information filtering: Algorithms for automating "word of mouth". *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 210–7.

Siedlecki, W., and J. Sklansky. 1989. A note on genetic algorithms for large-scale feature selection. *Pattern Recognition Letters* 10 (5): 335–47.

Sigurdsson, S. K. B. Petersen, and T. Lehn-Schioler. 2006. Mel frequency Cepstral coefficients: An evaluation of robustness to MP3 encoded music. *Proceedings of the International Conference on Music Information Retrieval.* 286–9.

Silla, Jr., C. N., C. A. A. Kaestner, and A. L Koerich. 2007. The Latin Music Database: Uma base de dados para a classificação automática de gêneros musicais. *Proceedings of the Eleventh Simpósio Brasileiro de Computação Musical*.

Sinyor, E., C. McKay, R. Fiebrink, D. McEnnis, and I. Fujinaga. 2005. Beatbox classification using ACE. *Proceedings of the International Conference on Music Information Retrieval.* 672–5.

Sjölander, K., and J. Beskow. 2000. WaveSurfer: An open source speech tool. *Proceedings of the International Conference on Spoken Language Processing.* 464–7.

Sleeman, D., R. Oehlman, and R. Davidge. 1990. Specification of Consultant–0 and a comparison of several learning algorithms. *Technical Report AUCS/TR9004*. Department of Computing Science, University of Aberdeen.

Smith, E. E., and D. L. Medin. 1981. *Categories and concepts.* Cambridge, MA: Harvard University Press.

van Someren, M. 2001. Model class selection and construction: Beyond the Procrustean approach to machine learning applications. In *Machine learning and its applications: Advanced lectures,* eds. G. Paliouras, V. Karkaletsis, and C. D. Spyropoulos, 196–217. New York: Springer.

Stevens, C., and C. Latimer. 1997. Music recognition: An illustrative application of a connectionist model. *Psychology of Music* 25 (2): 161–85.

Tagg, P. 1982. Analyzing popular music: Theory, method and practice. *Popular Music* 2: 37–67.

Tarasti, E. 2002. *Signs of music: A guide to musical semiotics*. New York: Mouton de Gruyter.

Tax, D., M. van Breukelen, R. Duin, and J. Kittler. 2000. Combining multiple classifiers by averaging or by multiplying? *Pattern Recognition* 33 (9): 1475–85.

Tekman, H. G., and N. Hortacsu. 2002. Aspects of stylistic knowledge: What are different styles like and why do we listen to them? *Psychology of Music* 30 (1): 28–47.

Temperley, D. 2001. *The cognition of basic musical structures*. Cambridge, MA: MIT Press.

Thompson, J., C. McKay, J. A. Burgoyne, and I. Fujinaga. 2009. Additions and improvements to the ACE 2.0 music classifier. *Proceedings of the International Society for Music Information Retrieval Conference*. 435–40.

Thorisson, T. 1999. Comparison of novice listeners' similarity judgments and style categorisation of Classic and Romantic piano examplars. *Psychology of Music* 26 (2): 186–96.

Tindale, A., A. Kapur, G. Tzanetakis, and I. Fujinaga. 2004. Retrieval of percussion gestures using timbre classification techniques. *Proceedings of the International Conference on Music Information Retrieval.* 541–4.

Towsey, M., A. Brown, S. Wright, and J. Diederich. 2001. Towards melodic extension using genetic algorithms. *Educational Technology & Society* 4 (2): 54–65.

Toynbee, J. 2000. *Making popular music: Musicians, creativity and institutions*. London: Arnold.

Tseng, Y. H. 1999. Content-based retrieval for music collections. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*. 176–82.

Turnbull, D., R. Liu, L. Barrington, and G. Lanckriet. 2007. A game-based approach for collecting semantic annotations of music. *Proceedings of the International Conference on Music Information Retrieval*. 535–8.

Tversky, A. 1977. Features of similarity. *Psychological Review* 84: 327–52.

Tversky, A., and I. Gati. 1982. Similarity, separability and the triangle inequality. *Psychological Review* 89: 123–54.

Tversky, A., and J. W. Hutchinson. 1986. Nearest neighbour evaluation of psychological spaces. *Psychological Review* 93: 3–22.

Typke, R., P. Giannopoulis, R. C. Veltkamp, F. Wiering, and R. van Oostrum. 2003. Using transportation distances for measuring melodic similarity. *Proceedings of the International Symposium on Music Information Retrieval*. 107–14.

Tzanetakis, G. 2002. Manipulation, analysis and retrieval systems for audio signals. *Ph.D. Dissertation.* Princeton University, USA.

Tzanetakis, G., and P. Cook. 2000. MARSYAS: A framework for audio analysis. *Organized Sound* 4 (3): 169–75.

Tzanetakis, G., and P. Cook. 2002. Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing* 10 (5): 293–302.

Tzanetakis, G., G. Essl, and P. Cook. 2001. Automatic musical genre classification of audio signals. *Proceedings of the International Symposium on Music Information Retrieval*. 205–10.

Tzanetakis, G., L. G. Martins, L. F. Teixeira, C. Castillo, R. Jones, and M. Lagrange. 2008. Interoperability and the Marsyas 0.2 runtime. *Proceedings of the International Computer Music Conference.*

Uitdenbogerd, A., and J. Zobel. 1998. Manipulation of music for melody matching. *Proceedings of the ACM International Multimedia Conference*. 235–40.

Vafaie, H., and I. Imam. 1994. Feature selection methods: Genetic algorithms vs. greedy-like search. *Proceedings of the International Conference on Fuzzy and Intelligent Control Systems*.

Verfaille, V. 2003. Effets audionumériques adaptatifs: théorie, mise en œuvre et usage en création musicale numérique. *Ph.D. Thesis*. l'Université Aix-Marseille II.

de Villiers, J., and E. Barnard. 1992. Backpropagation neural networks with one and two hidden layers. *IEEE Transactions on Neural Networks* 4 (1): 136–41.

Volk, A., P. van Kranenburg, J. Garbers, F. Wiering, R. C. Veltkamp, and L. P. Grijp. 2008. A manual annotation method for melodic similarity and the study of melody feature sets. *Proceedings of the International Conference on Music Information Retrieval.* 101–6.

Wanas, N. M., G. Auda, M. Kamel, and F. Karray. 1998. On the optimal number of hidden nodes in a neural network. *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*. 918–21.

Wanas, N. M., and M. S. Kamel. 2001. Decision fusion in neural network ensembles. *Proceedings of the International Joint Conference on Neural Networks*. 2952–7.

Wanas, N. M., and M. S. Kamel. 2002. Weighted combination of neural network ensembles. *Proceedings of the International Joint Conference on Neural Networks*. 1748–52.

Wang, G., R. Fiebrink, and P. R. Cook. 2007. Combining analysis and synthesis in the ChucK programming language. *Proceedings of the International Computer Music Conference*. 35–42.

Wasserman, L. 2004. *All of statistics.* New York: Springer.

Wenz, C. 2007. *Programming ASP.NET AJAX.* London: O'Reilly.

West, K., and S. Cox. 2004. Features and classifiers for the automatic classification of musical audio signals. *Proceedings of the International Conference on Music Information Retrieval.* 531–7.

Whitehead, P., E. Friedman-Hill, and E. Vander Veer. 2002. *Java and XML: Your visual blueprint for creating Java-enhanced Web programs.* Mississauga, Canada: Wiley Publishing Inc.

Whitman, B., and S. Lawrence. 2002. Inferring descriptions and similarity for music from community metadata. *Proceedings of the International Computer Music Conference*. 591–8.

Whitman, B., and P. Smaragdis. 2002. Combining musical and cultural features for intelligent style detection. *Proceedings of the International Conference on Music Information Retrieval*. 47–52.

Widmer, G. 1996. Recent developments in context-sensitive learning: The MetaL family of meta-learners. *Proceedings of Artificial Intelligence Techniques: The Third International Workshop*.

Witten, I. H., and E. Frank. 2005. *Data mining: Practical machine learning tools and techniques*. New York: Morgan Kaufman.

Wittgenstein, L. 1953. *Philosophical investigations.* New York: Macmillan.

Wright, M., A. Chaudhary, A. Freed, S. Khoury, and D. Wessel. 1999. Audio applications of the Sound Description Interchange Format standard. *Proceedings of the Audio Engineering Society Convention.* 276–9.

Wright, M., and A. Freed. 1997. Open Sound Control: A new protocol for communicating with sound synthesizers. *Proceedings of the International Computer Music Conference.*

Yip, C. L., and B. Kao. 1999. A study on musical features for melody databases. *HKU CSIS Technical Report* TR-99-05. Hong Kong University, Department of Computer Science and Information Systems, Hong Kong.

Zadel, M., and I. Fujinaga. 2004. Web services for music information retrieval. *Proceedings of the International Conference on Music Information Retrieval*. 478–83.

Zenobi, G., and P. Cunningham. 2001. Using diversity in preparing ensembles of classifiers based on different subsets to minimize generalization error. *Proceedings of the European Conference on Machine Learning*. 576–87.

Zhou, Z. H., J. X. Wu, Y. Jiang, and S. F. Chen. 2001. Genetic algorithm based selective neural network ensemble. *Proceedings of the International Joint Conference of Artificial Intelligence.* 797–802.

Ziv, N., and Z. Eitan. 2007. Themes as prototypes: Similarity judgments and categorization tasks in musical contexts. *Musicae Scientiae* Discussion Forum 4a: 99–133.

2005 MIREX Contest Results – Symbolic Genre Classification. Retrieved 14 December 2009, from http://www.music-ir.org/evaluation/mirex-results/sym-genre/index.html.

7digital. Retrieved 14 December 2009, from http://ca.7digital.com.

AllMusic. Retrieved 14 December 2009, from http://www.allmusic.com.

Amazon. Retrieved 14 December 2009, from http://www.amazon.com.

Apple - Download Music and More with iTunes. Retrieved 14 December 2009, from http://www.apple.com/itunes/.

Apple Developer Connection. Retrieved 16 December 2009, from http://developer.apple.com/sdk/.

Audacity. Retrieved 16 December 2009, from http://audacity.sourceforge.net.

Audio Genre Classification (Mixed Set) Results. Retrieved 14 December 2009, from http://www.music-ir.org/mirex/2009/index.php/Audio_Genre_Classification_%28Mixed_Set%29_Results.

Audiobaba Music Search. Retrieved 14 December 2009, from http://www.audiobaba.com.

BBC. Retrieved 14 December 2009, from http://www.bbc.co.uk.

Billboard. Retrieved 14 December 2009, from http://www.billboard.com.

Blip.fm. Retrieved 14 December 2009, from http://blip.fm.

Celemony Direct Note Access. Retrieved 16 December 2009, from http://www.celemony.com/cms/index.php?id=dna_qa.

Chroma Toolbox. Retrieved 23 January 2010, from http://www.mpi-inf.mpg.de/~mmueller/chromatoolbox/.

CrestMuse Project. Retrieved 30 December 2008, from http://www.crestmuse.jp/index-e.html.

Critical Metrics. Retrieved 14 December 2009, from http://criticalmetrics.com.

DBPedia. Retrieved 14 December 2009, from http://dbpedia.org.

Discogs. Retrieved 14 December 2009, from http://www.discogs.com.

Dublin Core Metadata Initiative. Retrieved 16 February 2009, from http://dublincore.org.

The Echo Nest. Retrieved 14 December 2009, from http://echonest.com.

Epitonic. Retrieved 20 December 2009, from http://www.epitonic.com.

ExploreMusic. Retrieved 14 December 2009, from http://www.exploremusic.com.

Facebook Developers. Retrieved 14 December 2009, from http://developers.facebook.com.

The Filter. Retrieved 14 December 2009, from http://www.thefilter.com.

Free Music Archive. Retrieved 14 December 2009, from http://freemusicarchive.org.

Freebase. Retrieved 14 December 2009, from http://www.freebase.com.

freeDB. Retrieved 14 December 2009, from http://www.freedb.org.

GarageBand. Retrieved 20 December 2009, from http://www.garageband.com.

Gigulate. Retrieved 14 December 2009, from http://gigulate.com.

The GNU General Public License. Retrieved 16 December 2009, from http://www.gnu.org/licenses/gpl.html.

Google Code. Retrieved 14 December 2009, from http://code.google.com.

Gracenote. Retrieved 14 December 2009, from http://www.gracenote.com.

Grooveshark. Retrieved 14 December 2009, from http://listen.grooveshark.com.

Idiomag. Retrieved 14 December 2009, from http://www.idiomag.com.

iLike. Retrieved 14 December 2009, from http://www.ilike.com.

The International Society for Music Information Retrieval. Retrieved 16 December 2009, from http://www.ismir.net.

Jamendo. Retrieved 20 December 2009, from http://www.jamendo.com/en/.

jMIR. Retrieved 1 December 2009, from http://jmir.sourceforge.net.

JSON. Retrieved 15 January 2009, from http://json.org/.

Last.FM. Retrieved 14 December 2009, from http://www.last.fm.

Listilpy. Retrieved 14 December 2009, from http://www.listiply.com.

LyricsFly. Retrieved 14 December 2009, from http://lyricsfly.com.

LyricWiki. Retrieved 14 December 2009, from http://lyrics.wikia.com.

Magnatagatune. Retrieved 20 December 2009, from http://tagatune.org/Magnatagatune.html.

Magnatune. Retrieved 20 December 2009, from http://magnatune.com.

The MathWorks. Retrieved 14 December 2009, from http://www.mathworks.com.

Meandre. Retrieved 7 January 2009, from http://seasr.org/meandre/.

MediaUnbound. Retrieved 14 December 2009, from http://www.mediaunbound.com.

MediaWikiAPI. Retrieved 14 December 2009, from http://en.wikipedia.org/w/api.php.

The Melisma music analyzer. Retrieved 9 June 2009, from http://www.link.cs.cmu.edu/music-analysis.

MIDI Manufacturers Association. Retrieved 9 June 2009, from http://www.midi.org.

MIREX. Retrieved 14 December 2009, from http://www.music-ir.org/mirex/2009/.

Music Ontology Specification. Retrieved 30 December 2008, from http://www.musicontology.com.

MusicBrainz. Retrieved 14 December 2009, from http://musicbrainz.org.

Musicovery. Retrieved 14 December 2009, from http://musicovery.com.

MySpace Developer Platform. Retrieved 14 December 2009, from http://developer.myspace.com/community/.

National Public Radio. Retrieved 14 December 2009, from http://www.npr.org.

NEMA. Retrieved 14 December 2009, from http://nema.lis.uiuc.edu.

Omnifone. Retrieved 14 December 2009, from http://www.omnifone.com.

Pandora Internet Radio. Retrieved 14 December 2009, from http://www.pandora.com.

People's Music Store. Retrieved 14 December 2009, from http://peoplesmusicstore.com.

Playdar. Retrieved 14 December 2009, from http://www.playdar.org.

Plug-ins for Windows Media Player. Retrieved 16 December 2009, from http://www.microsoft.com/windows/windowsmedia/player/plugins.aspx.

Rhapsody. Retrieved 14 December 2009, from http://www.rhapsody.com.

Rovi. Retrieved 14 December 2009, from http://www.rovicorp.com.

SDIF Sound Description Interchange Format. Retrieved 30 December 2008, from http://sdif.sourceforge.net.

SEASR. Retrieved 16 December 2009, from http://nema.lis.uiuc.edu.

Slacker Personal Radio. Retrieved 14 December 2009, from http://www.slacker.com.

Songkick. Retrieved 14 December 2009, from http://www.songkick.com.

Sonic Annotator. Retrieved 14 December 2009, from http://www.omras2.org/SonicAnnotator.

SoundCloud. Retrieved 14 December 2009, from http://soundcloud.com.

Spotify. Retrieved 14 December 2009, from http://www.spotify.com.

TunedIT. Retrieved 1 October 2009, from http://tunedit.org.

Twitter API Wiki. Retrieved 14 December 2009, from http://apiwiki.twitter.com.

Vamp Plugins. Retrieved 14 December 2009, from http://www.vamp-plugins.org.

Weka >> ARFF. Retrieved 30 December 2008, from http://weka.wiki.sourceforge.net/ARFF.

Winamp Plugins. Retrieved 16 December 2009, from http://www.winamp.com/plugins.

Yahoo! Developer Network Home. Retrieved 14 December 2009, from http://developer.yahoo.com.

Yahoo! Music. Retrieved 14 December 2009, from http://new.music.yahoo.com.

Yes. Retrieved 14 December 2009, from http://www.yes.com.