Dynamic Documents for Data Analytic Science

by

Gabriel Becker
B.S. (California Polytechnic State University at San Luis Obispo) 2008

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Statistics

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA at DAVIS

Approved:

Duncan Temple Lang, Chair

Deborah Nolan

Paul Baines

Committee in Charge
2014

Acknowledgements

My deepest gratitude goes out to my advisor, Duncan Temple Lang, for showing me that research in our field can and must go beyond the programming with which we implement our ideas. I am also deeply grateful to Deborah Nolan, whose support, collaboration, and guidance went far beyond the duties strictly prescribed to a committee member. I would not be where I am today without the ideas, advice, opportunities, and constructive criticism provided by Duncan and Deborah throughout my time performing and later describing this research.

My thanks also go out to the remaining members of my committee, Paul Baines, Wolfgang Polonik, and Alexander Aue. Their insightful comments and discussion were invaluable in allowing the following work to reach its potential.

No graduate career could possibly succeed without the dedicated work of many beyond the professors and their students. I gratefully acknowledge the tireless efforts of Pete Scully and Pat Aguilera, without whom our department would surely have ceased to function entirely.

Finally, and with great love, I thank my wife Erin. Without her unending support throughout this process, the following thesis - and my life while producing it - would have been greatly diminished.

Abstract

The need for reproducibility in computational research has been highlighted by a number of recent failures to replicate published data analytic findings. Most efforts to ensure reproducibility involve providing guarantees that reported results can be generated from the data via the reported methods, with a popular avenue being dynamic documents. This insurance is necessary but not sufficient for full validation, as inappropriately chosen methods will simply reproduce questionable results. To fully verify computational research we must replicate analysts' research processes, including: choice of and response to exploratory or intermediate results, identification of potential analysis strategies and statistical methods, selection of a single strategy from among those considered, and finally, the generation of reported results using the chosen method.

We present the concept of comprehensive dynamic documents. These documents represent the full breadth of an analyst's work during computational research, including code and text describing: intermediate and exploratory computations, alternate methods, and even ideas the analyst had which were not fully pursued. Furthermore, additional information can be embedded in the documents such as data provenance, experimental design, or details of the computing system on which the work was originally performed. We also propose computational models for representing, processing, and programmatically operating on such documents within R.

These comprehensive documents act as databases, encompassing both the work that the analyst has performed and the relationships among specific pieces of that work. This allows us to investigate research in a number of ways difficult or impossible to achieve given only a description of the final strategy. We can explore the choice of methods and whether due diligence was performed during an analysis. Secondly, we can compare alternative strategies either side-by-side or interactively. Finally, we can treat these complex documents as data about the research process and analyze them programmatically.

We also present a proof-of-concept set of software tools for working with comprehensive dynamic documents. This includes an R package which implements a framework for comprehensive documents in R, an extension of the IPython Notebook platform which allows users to author and interactively view them, and a caching mechanism which provides the efficiency necessary for interactive, self-updating views of such documents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Research should be reproducible. A number of recent, high-profile incidents have highlighted the need for reproducibility of computational research in fields ranging from bioinformatics (Baggerly and Coombes, 2009) to economics (Herndon et al., 2013).

In the experimental sciences, work must generally meet two requirements in order to be considered reproducible. First, enough detail must be given in the description of the work that an independent laboratory could replicate the experiment. Secondly, these independent recreations of the experiment, when performed, must give scientifically equivalent results.

The primary purpose of reproducing an experimental result is to validate and confirm the conclusions of the original work. Most obviously, reproducing the experiment itself does this by generating independent corroborative data which strengthen the body of evidence for the original conclusion.

Reproducing the full experimental process from data generation to conclusions also acts as a corroboration of the *choice of experimental methods*, though more indirectly. As an independent group of scientists work through a series of experiments, they might find fault with the experimental design or statistical methodology chosen by the original analyst, even if repeating the original steps precisely leads to results which agree with the original. It is this methodological corroboration which is key to fully verifying research results.

We can validate the results of computational analyses in a manner similar to the validation of experimental results: by inspecting the choices made by the analyst as well as confirming that the chosen methods generate the reported results when applied to the data. This is true even when generating new data is impossible (e.g., analyzing election data for fraud, or tracking immediate response to a current event via Twitter or Facebook posts).

The importance of inspecting an analyst's choice of methods is compounded by the fact that data analysis can be somewhat subjective, with multiple valid ways of approaching a question. There are many wrong methods of approaching any given statistical question, but there may be multiple "correct" approaches as well. Furthermore, the appropriateness of particular methods for a given data set is often not discernible a priori. We will see this shortly in the context of an example analysis where the analyst seeks to build a classifier which can predict the intended numeric digit given a handwritten character.

Dynamic documents have gained interest recently as a tool for ensuring that reported results correctly correspond to the computations performed by an analyst. Conceptually, a dynamic document is a document which combines code and text such that the code can be run on command, with text, code, and output automatically inserted into an article or report intended for human consumption. This automatic generation and insertion of output ensures that the correct output – e.g plots, tables, and numerical results – are included in the final result, and that the actual code used to generate the results in the report is available.

A common form of dynamic document is a file containing a linear sequence of interspersed code and text chunks which respectively define a series of computational steps and provide discussion of those steps and their results. Most existing dynamic document systems limit themselves to this linear formulation of dynamic documents, including the popular SWeave (Leisch, 2002) and **knitr** (Xie, 2013a) systems for the R statistical programming language (R Core Team, 2014), as well as the ActivePapers (ActivePapers Development Team, 2013) and IPython Notebook (Pérez et al., 2013a) projects for Python (Python Development Team, 2013).

These linear dynamic documents act as templates for narratives constructed around a sequence of computations and their results. In the context of an article-like narrative, computations which are not necessary to directly generate the final results from data are typically omitted. When the template is processed, the code is evaluated and the output generated is used to construct the report. This fits with the view proposed by Claerbout (Claerbout and Karrenfach, 1992) and later championed by Buckheit and Donoho (Buckheit and Donoho, 1995) that the deliverable in computational research is the combination of output with any algorithms, scripts, or software necessary to generate it, rather than solely the output itself.

The merit of computational and statistical results must be viewed as contingent on the researcher's choices of which methods and strategies to apply. These choices are nearly always based on exploratory computations which investigate the data, but which are not required to generate the results once a method is chosen. These computations – and the insights they grant – are a key part of the analysis process which is often not reported when describing the work.

To fully assess or validate computational results we must be able to replicate all intermediate computations and results which led the analyst to choose the final methods and parameters used to generate those results. Rossini recognized this, advocating data analyses be described via documents which *"describe results and lessons learned, both substantive and for statistical practice,* ***as well as a means to reproduce all steps, even those not used in a concise reconstruction, which***

were taken in the analysis" (emphasis ours) (Rossini, 2001).

A mistake in crucial intermediate computations or their interpretation can invalidate results as completely as a mistake in applying the chosen method would. Beyond detecting mistakes that invalidate a set of results, information about these intermediate steps also allows us to understand why an author made the choices she did. Furthermore, we can consider whether we agree with her reasoning, or would have chosen a different strategy. Without knowledge of the strategies considered and access to the intermediate results that the analyst used to make her final decisions, this type of assessment is much more challenging. Examples of computations which are important to the research process but unnecessary to regenerate that research's final results include:

- determining that the data violate the distributional assumptions for the relevant parametric methods(s), leading to a focus by the analyst on non-parametric approaches,

- performing a regression analysis and detecting autocorrelation in the residuals, leading the analyst to abandon ordinary regression as a strategy,

- trying multiple data transformations (e.g, square-root, logarithm, etc.) before choosing one to apply.

We propose a type of dynamic document which more accurately encompasses the entire research process while retaining the ability to dynamically generate concise, article-like reports. Specifically, we feel the following are essential for documents to fully capture, communicate, and provide reproducibility for the research process:

- text and code content for all computations which contributed to the generation of final results *or the choice of methods or strategies used;*

- relationships between groups of content, e.g., the set of methods and computations considered when choosing how to answer a particular question;

- semantic information about content, e.g., that applying a particular method generated an error, or failed its diagnostics when applied to the data, and even the reasoning for choosing or not choosing a particular alternative.

In the remaining sections of this chapter we motivate the concepts which will underlie our proposed documents in the context of two data analysis projects. For each project, we briefly discuss the actions, decisions, and reasoning a data analyst approaching the problem might perform. Our two examples are chosen specifically to highlight different aspects of the research process which would not necessarily be captured in a concise narrative such as a published journal article.

Finally, we conclude the chapter with an outline for the remainder of this thesis. We briefly summarize the relevance of each of the three software tools we will present: the **DynDocModel** and **RCacheSuite** R packages, and our modified version of the IPython Notebook (Pérez et al., 2013a) interactive computing environment.

1.1 Examples of Data Analyses

Data analysis is often a sprawling, non-linear process involving much more than applying a single statistical method to data. We illustrate these more involved aspects of the research process via two examples: an investigation of classification strategies for handwritten numeric digits, and an exploratory data analysis of a census of housing sales in the San Francisco Bay Area between 2003 and 2006 with a focus on properties sold more than once during that time.

We "ride along" with the data analyst performing each analysis, looking to both their actions and reasoning as they explore the data, consider the best way to proceed, and eventually generate their final results. We use these two hypothetical analyses to identify key aspects of the research process not typically reflected in journal articles or other write-ups of analysis results. We then use our observations to motivate a discussion of the structure of dynamic documents and how it might be generalized to more fully reflect the research process.

1.1.1 Classifying handwritten digits

We first step into the mind of an applied statistician tasked with developing a classifier to identify single, hand-drawn digits. He first obtains a manually classified dataset for use as a training set. The data are digitized, black-and-white 28x28 pixel images of handwritten numbers (0-9) that have been centered and normalized for size. Each observation in the data consists of a value between 0 and 255 for each of the 784 individual pixels as well as a label identifying the true digit drawn. The pixel values range between 0 representing white space (no ink) and 255 representing black (darkest ink).

Initial exploration

The analyst first seeks to understand his data through some simple explorations. He first plots an individual observation, receiving the picture shown in Figure 1.1. He notes that the plot shows a clearly identifiable digit – either a six (‘6’) or a nine (‘9’) – but the orientation is wrong. Checking the label for the observation, he finds that it is a nine and uses this information to determine the transformation needed to get it to display correctly. Satisfied that he understands the data at the single observation level, he makes a note of this transformation to himself and moves on.

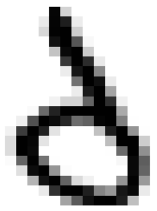


Figure 1.1: ***Plotting a single observation.*** The analyst first plots a single observation. He sees that the image is clearly a numeric digit, but that its orientation when plotted naively is incorrect. He uses this, along with the observation's label identifying it as a nine, to better understand the orientation of the data and how it should be plotted.

The analyst continues by becoming familiar with the data set. In particular, he investigates where the ink (non-zero pixel values) tends to be within the images. He first quickly confirms his intuition that because the data are scanned images of curves in black ink drawn on white paper, most pixel values will tend to be either small (less than 50) indicating white space or large (greater than 200) indicating ink. He finds that he was correct, with only about 6% of the pixels having intermediate values in the entire training dataset.

With the knowledge that the values tend to the extremes, he considers heatmaps of two different pixel-level measures: variance (Figure 1.2), and count of non-zero values in the data. The first gives him a sense of which areas of the grid tend to differ the most among observations, while the second indicates areas which most commonly contained ink. He sees that the variability in presence of ink is concentrated in a central area, with little to no difference between observations near the corners of the 28x28 grid. This will become important later, but for now he adds this to his understanding of the data and turns to identifying viable classification strategies.

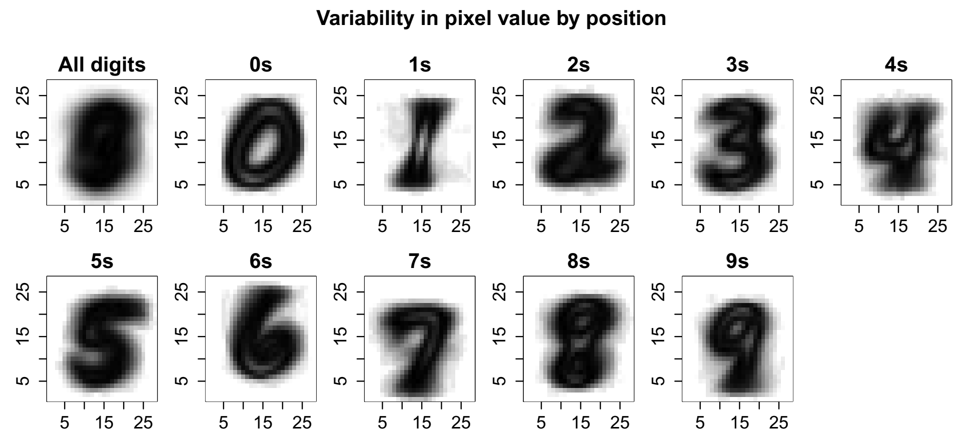


Figure 1.2: ***Investigating typical ink location.*** The analyst creates a heatmap of the individual pixel variances to see which areas differ across the data and for each digit separately. Differences between pixel values across different observations are common in the center of the 28x28 grid and essentially absent near its corners.

Exploring possible classification methods

Before pursuing a strategy for all ten digits, the analyst decides to see how various classification methods perform when asked to distinguish only two relatively distinct classes: ones (‘1’s) and eights (‘8’s). He reasons that analyzing only two digits is less computationally taxing, which will allow him to iterate and explore quickly without worrying about code optimization. He plans to exclude methods which have high error rates here from consideration, noting that the full set of ten digits contains pairs likely to be much more difficult to distinguish, including three and eight (‘3’ vs ‘8’), two and seven (‘2’ vs ‘7’), four and nine (‘4’ vs ‘9’), and five and six (‘5’ vs ‘6’). The analyst reasons that while the method which performs best on the restricted dataset is not guaranteed to prove optimal in the ten digit setting, methods which perform very poorly on only two classes – e.g., having misclassification rates several times higher than the best performing method – are unlikely to be competitive.

The analyst begins with a simple classical method, Linear Discriminant Analysis (LDA). R’s `lda()` function, from the **MASS** package (R Core Team, 2014), fails, however, throwing an error indicating that some variables "appear to be constant within groups." Considering the math underlying LDA, he recalls that it involves inverting the sample covariance matrix (S) and realizes that any pixels which have the same value across all observations within one of the classes (intended digits) would lead to S being singular.

The analyst thinks back to his exploration of typical ink location, recalling that he saw no variability near the corners of the plot. A quick calculation confirms that 135 of the 784 pixels are always white space (zero) within the full dataset, and that 296 pixels are non-zero in less than 1% of observations. Omitting the 135 constant pixels, 253 pixels have a correlation greater than 0.80 with at least one other pixel. The analyst infers from this that dimension reduction beyond simple exclusion of constant pixels is likely to be effective.

The analyst chooses principal components as his dimension reduction method, reasoning that it is well understood and relatively efficient to implement. He then considers his options for generating his principal component data in R. He can use the spectral decomposition or the single value decomposition to generate the eigen-values and vectors of his data's covariance matrix. Furthermore, he can write his own function to calculate the principal components, or can use a higher-level function provided by R, e.g., `prcomp()` (R Core Team, 2014). He decides to explore multiple possible implementations and compare their computational efficiency. He ultimately chooses to use a custom function based on the SVD, but retains the other implementations and makes a note of why he chose this one (a combination of efficiency and numerical stability).

After he generates principal components for the full data (all ten digits), he decides how many components to use via scree plots and numerical summaries. He finds that he needs 148 components to account for 90% of the total variation. The first 25 contribute roughly 70%, however, and each individual component after the 23$^{\text{rd}}$ contributes less than 1% to the total. He decides to proceed with 25 components, but designs his code so that he can easily switch to using more principal components later if necessary.

The analyst then switches gears back to his two-digit task. He generates 25 separate principal components for just this subset of the data in order to give each method he will try the opportunity to perform as well as possible. He plots the components of the eigen-vectors used to generate the first nine components for his '1's and '8's data (Figure 1.3) and finds them relatively interpretable. He sees that the strongly positive and negative elements of the first vector create the outline of an '8' contrasted with the areas in the center of the image where an '8' is largely empty but a '1' is likely to have ink. The second eigen-vector emphasizes the contrast between one of the diagonals in the '8' against a largely vertical line more in-line with a '1', and so on.

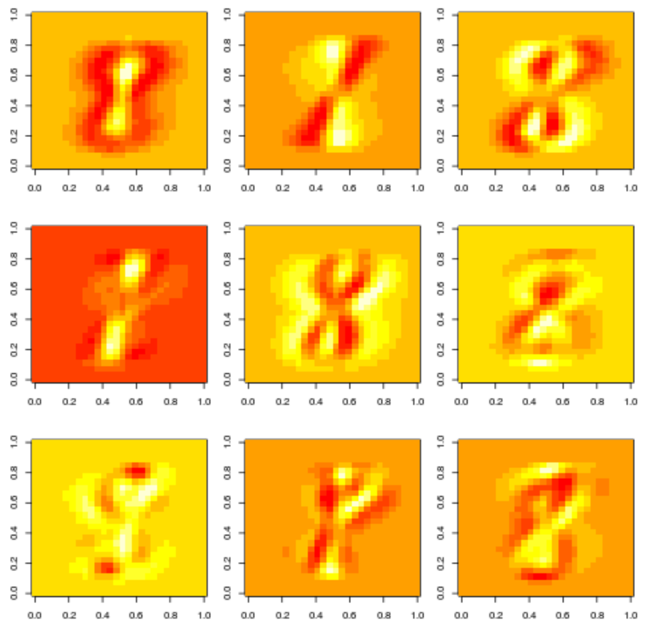


Figure 1.3: ***Interpretability of the first nine principal components for the combined '1's and '8's data.*** The eigen-vectors ("loadings") which generate the first nine principal components for the '1' and '8' data are highly interpretable. The first vector (top-left) clearly shows the outline of an '8' emphasized in one range of colors contrasted with portions of the center.

The analyst then returns to the LDA approach using the principal component data. The sample covariance matrices for these data are non-singular so the method is mathematically feasible. Before applying LDA, he investigates whether the two major assumptions of the method hold: equal covariance for the two classes, and multivariate-Normality within each population. He uses a heuristic of comparing observed Mahalanobis distances against theoretical quantiles for the Chi-squared distribution with 25 (the number of components he is using) degrees of freedom to test for gross violations of multivariate-Normality. Creating a QQ plot (Figure 1.4) he sees a highly non-linear relationship between the quantiles of the Mahalanobis distances and those of the Chi-squared distribution, leading him to conclude that the principal component data are substantially non-normal. Because each component is a linear combination of the individual variables, this implies that the raw pixel data are also non-normal. The analyst makes a note of this in case he goes back to analyzing the raw data in the future.

The analyst looks for violation of equality-of-variances by comparing the pixel-level variances. Because these form the diagonal of the covariance matrix, this provides a simple spot-check which can expose inequality without having to analyze the full matrices. A quick analysis of the ratios between the variances in the ‘1’ and ‘8’ populations of the twenty-five principal component variables show some strong discrepancies: the second component is nearly 6 times more variable in the ‘1’ population than among the ‘8’s, while the fifteenth component is about 8.4 times (nearly a full order of magnitude) higher in the ‘8’ population than the ‘1’s.

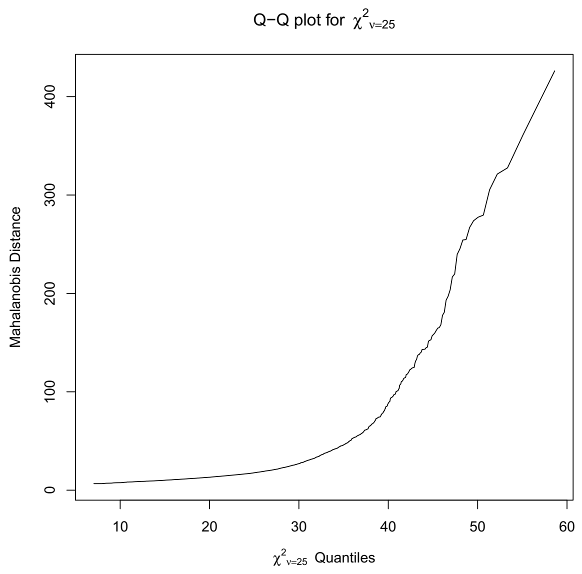


Figure 1.4: ***Testing multivariate-Normality with the Mahalanobis distance heuristic.*** The analyst tests whether the data follow a multivariate-Normal distribution by comparing the principal component data's Mahalanobis distances to their theoretical distribution under multivariate-Normality. The highly non-linear correspondence leads him to conclude that these data (and thus the original data as well) are not multivariate-Normal.

A scatterplot (Figure 1.5) shows that even among the components with less extreme differences, there is a strong trend – twenty-two of the twenty-five components – of the component variance estimates to be higher in the '8' class. The analyst concludes from this that the component variances – and thus the covariance matrices – are not equal for the two classes.

The analyst decides to go ahead with using LDA as his baseline classifier, despite evidence suggesting that neither of the two assumptions he investigated hold, because it is known to be somewhat robust to such violations. With the principal component data, the `lda()` function runs successfully and returns a classifier. Testing his LDA classifier on independent data, the analyst finds that it has an error rate of approximately 3%.

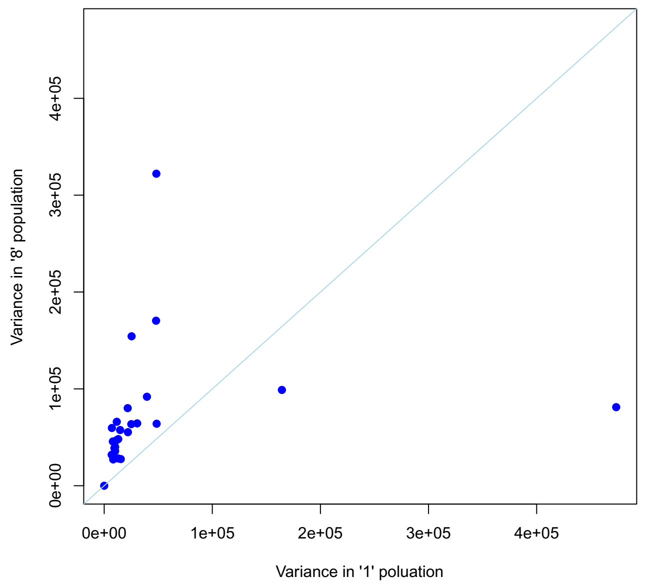


Figure 1.5: ***Comparing sample variance for the first 25 principal components between '1's and '8's.*** The analyst investigates equality of the variances for the principal component values in the '1' and '8' populations as a heuristic for equality of covariance matrices. He concludes that the covariances are not equal after seeing highly divergent values and a general trend of variability among '8's to be higher than '1's for the same component (twenty-two of twenty-five components).

The analyst suspects that quadratic discriminant analysis (QDA), which does not require the equal covariances assumption, may perform better than LDA. Using R's `qda()` function, the analyst generates a new classifier, finding that it achieves an overall error rate of approximately 1.5%, or roughly half that of LDA. QDA still assumes multivariate-Normality, however, so the analyst suspects he may do even better with non-parametric methods.

For a non-parametric classification method, the analyst turns to k-nearest neighbors (KNN). The number of neighbors considered (k) acts as a tuning parameter for this method, so he decides to select it via ten-fold cross-validation. He finds that R's built-in `knn()` function from the **class** package (R Core Team, 2014) calculates the distance matrix afresh each time it is called to create a new classifier, which is expensive when performing cross-validation. He writes a custom KNN function which, while slower to fit once the distances are calculated, allows the distance matrix to be passed in as an argument. He uses this to build a cross-validation mechanism that calculates the distance matrix for the full data once and reuses portions of it within each validation block. For his training data, 1 is selected as the optimal value of k, generating a classifier with an error rate of approximately 0.3% for independent data, performing about 5 times better than QDA and beating LDA by a full order of magnitude.

Encouraged by the strong performance by KNN, the analyst decides to explore other non-parametric methods to see if any can improve on KNN's error rate. He turns to the recursive partitioning family of methods. He tries standard recursive partition trees (Breiman et al., 1984) – using Therneau et al.'s **rpart** package (Therneau et al., 2014) – first, with a plan to apply the random forest variant (Breiman, 2001) as well. The analyst finds that the original recursive partition tree approach does not perform particularly well for his data, achieving an error rate on par with QDA (roughly 1.5%).

Finally, he finishes his exploration of two-digit classifiers by generating a random forest classifier for his data. The random forest classifier achieves an error rate of approximately 0.9%, outperforming the parametric methods (and **rpart**), but unable to surpass KNN. The analyst turns his attention to building classifiers for the full dataset, having narrowed the field of possible methods from his original five to two, KNN (the front-runner) and random forest.

Building a ten-digit classifier

The analyst's exploration of classifiers in the easier-to-distinguish two-digit case ('1's and '8's) allowed him to exclude LDA, QDA, and rpart as contenders for his final strategy. After generating the principal component data for the full ten digit training set, he compares KNN and random forests to determine which he will select as his final method.

The analyst applies KNN first. He finds that his cross-validation function is quite expensive for the full data. He notes that generating a single classifier for a given value of k using `knn()`, however, is relatively fast. He proceeds to simply try a few values for k (1, 3, 5, and 7) and compare their performance on his independent testing data directly. He finds that all four k values he tried have error rates of approximately 3%, with 5 the lowest at 3.1% and 1 and 7 roughly tied for worst with about 3.2%. The analyst notes that these error rates are much higher than KNN achieved for the two-digit case, but recalls that he had specifically chosen two digits which would be easy to distinguish. The analyst inspects the confusion matrix and confirms that the more difficult pairs ('3'-'8', '4'-'9', etc) contribute the vast majority of the errors, with misclassification rates an order of magnitude higher than other pairs in some cases. The highest error count was caused by confusing fours ('4's) as nines ('9's), with 38 of 1348 observations labeled as '4's classified as '9's (2.8%), more than twice the number of errors confusing '4's as all other digits combined.

Finally, the analyst concludes by comparing his KNN classifiers to a classifier generated via random forests. The random forest classifier achieves an error rate of about 5.5%. The ratio of the random forest and KNN error rates is closer for the full data than it was for only the 1 and 8 populations, but KNN remains the best of the investigated methods.

Ultimately, the analyst chooses KNN (with k=5) as his classifier. In a real data analysis situation, the analyst would likely consider other strategies beyond those we describe above. The analyses we have described, however, is sufficiently complex to illustrate the concepts we wish to discuss, so we stop here and move on to the analyst's process for formulating one or more write-ups of his results.

Summarizing the results

We imagine the statistician preparing several different write-ups of his analysis and results. Each might record different aspects of what he did, or contain discussion targeting different audiences (collaborators, independent researchers, reviewers, managers). We list several possible write-ups and their target audiences below, followed by a detailed discussion of the aspects of the analyst's work and code reflected by each write-up.

1. A brief *executive summary* which focuses on the results of his chosen KNN classifier and their implications within the context of the problem. This would provide high-level managers, policy makers, etc. an overview of the analyst's results without requiring them to wade through any technical details.
2. A *multi-resolution report* that summarizes the process of generating the final KNN classifier from the raw data, and that allows the reader to drill down into particular components to see the details. This would serve more technical managers or researchers in the same field by allowing them to view both the big picture result and the details (i.e., that KNN with k=5 was used on data comprising 25 principal components) of how that result was generated.
3. A *technical report* discussing the code which generates the final result from the raw data, including implementation considerations and a comparison of alternative implementations. This would serve a team implementing the analyst's approach within a production system or larger analysis.
4. An *academic article* describing the analysis. This would serve as the primary introduction to the analyst's work for peers in his field. A typical article's narrative would include a discussion of the final method and results, a subset of the exploratory results (e.g., select plots) and possibly a brief mention of the other alternatives considered and why they were

rejected.

5. An *electronic notebook* giving a full account of his research path, with his actions arranged into conceptual tasks and the reasons for each of his decisions clearly indicated. This notebook would allow the analyst's work to be audited and validated, as well as serving as a detailed foundation for future extensions of the work by himself or others.
6. An *interactive article* which allows another investigator to test out the classifier on a different set of data, and to consider different values of k or numbers of principal components. This would allow other researchers interested in applying the analyst's strategy to new data to get a feel for how the KNN classification method behaves, and how it is affected by the data and specific parameter value (k=5) used to generate his results.
7. An *interactive research notebook* that allows the viewer to select among all methods, implementations, and parameters considered by the analyst, and to show or hide supplementary detail, at viewing time. This would allow an independent researcher or referee to validate the statistical choices made by the analyst, as well as the implementation of the chosen methods.

We now consider which portions of the analyst's full process would be included in a selection of the types of write-up described above, and why. We also determine which portions of the analyst's code are necessary to replicate each write-up. By describing the write-up types this way, we will see that each type corresponds to describing a subset of the full body of research. We expand on the concept of write-ups only requiring a subset of the analyst's full body of work later in this chapter.

The executive summary (1) describes only the results of the final chosen classifier. To replicate the results presented in this type of report, we need only the code to import the data, generate the principal components, generate the final classifier, and calculate error rates and other performance summaries. The intermediate explorations performed by the analyst and the alternatives he considered but ultimately decided against are not necessary for the creation or replication of his final results. Aspects of the analyst's research which are not necessary to regenerate the results presented in such a summary include his explorations of the raw and principal component data, the investigation of two-digit classifiers, the creation of the random forest classifier for all ten digits, and the investigation of other values of k or numbers of components for the ten-digit KNN classifier.

The multi-resolution "drillable" report (2) summarizes the process of fitting the classifier while allowing the viewer to access more detail about specific steps. Because this write-up relates only to the generation of the final results, it requires the same code as the executive summary to be replicated, though more of the process can be visibly displayed and described to the viewer in this case.

The technical description (3) again discusses only the steps necessary to generate the final results from the data based on the analyst's choices. Here, however, the analyst includes discussions of other implementations of his final strategy, including different strategies for generating the principal component data and any investigations he made into alternate ways of performing the actual KNN method. To replicate this report, we need code to create the final classifier using each combination of all implementations he considered for principal component generation and KNN fitting. As before, none of the data explorations, two-digit classifier investigations, alternative ten-digit classifiers, or different parameter values for the KNN classifier are necessary for replication.

The academic article (5) would likely add a selection of the exploratory visualizations to a discussion of the final classifier and its result. It would also likely mention that other values of k and numbers of principal components were considered, and why the final ones were chosen. To replicate such an article, we require full code to generate each image in the article, as well as the code to generate our final results from the raw data. We would not need the two-digit classifier investigations, code to generate any numeric summaries or plots not used in the article, or any code involving the random forest classifier for replicability.

The comprehensive research notebook (6) involves all work done by the analyst. Thus, to replicate the results in the notebook, the full body of code generated by the analyst is required. Replicating this corroborates both the results and the research decisions made by the analyst.

Furthermore, the material necessary to replicate the research notebook also allows us to replicate *all possible write-ups of the research* because it encompasses the entirety of the analyst's work on the project. This is key to our strategy for representing computational research via dynamic documents, which we will see later in this chapter.

Looking back at the analysis process

We now briefly pause to discuss the shape and flow of the analyst's work, as this will serve as a major motivation for the design of our proposed documents in Section 1.2. We provide a simplified illustration of the structure of his research process in Figure 1.6, as well as a brief discussion below.

The analyst's work can be described as a series of high-level tasks. He first explored the data, at both the individual observation and full dataset levels. He then performed a preliminary analysis on only two of the classes ('1's and '8's), testing a number of different prospective classification methods. Finally, he used what he had learned while performing those tasks to implement and decide amongst a smaller number of methods for all ten classes.

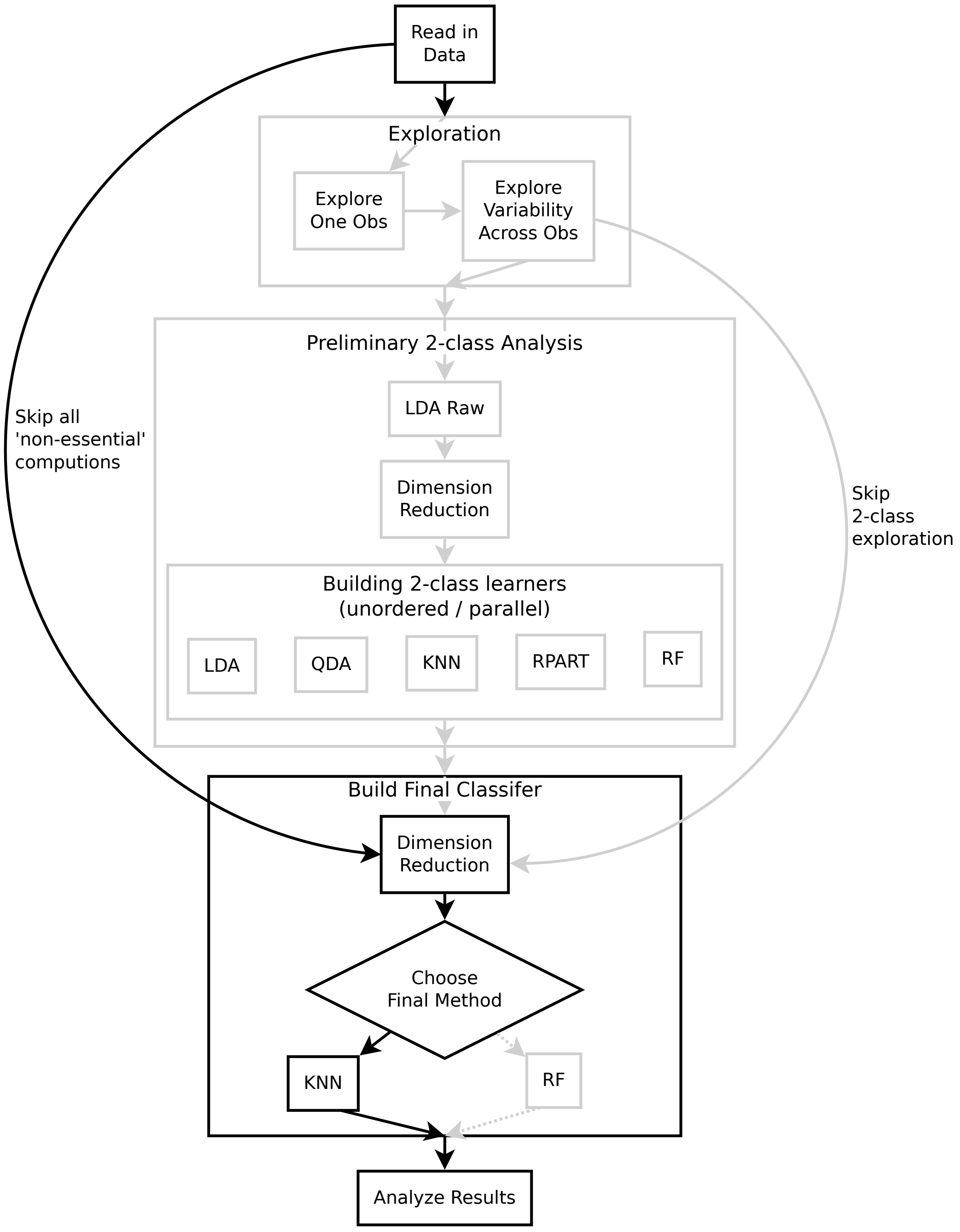


Figure 1.6: ***A simplified illustration of the structure of the digit analyst's research.*** Essential tasks and selected alternatives, i.e., those computations required to generate his final classifier and results – and the connections between them – are drawn in black. Exploratory tasks, unselected alternatives, and their connections are pictured in grey. The essential tasks make up a small fraction of the analyst's work overall, and furthermore the workflow outlined in black is very different from the research process which selected the final strategy. Note some levels of nesting are omitted for simplicity.

Each of these high-level tasks contained multiple lower-level conceptual sub-tasks. His initial explorations can be thought of as the combination of an "explore a single observation" sub-task and an "explore the regions of the grid which tend to vary across the dataset" sub-task. Likewise, the preliminary analysis is made up of a collection of sub-tasks, one for dimension reduction and one for each classification method he considered at this stage.

These tasks and sub-tasks contain the decisions made by the analyst, as we see in the ten-class analysis tasks, where he actively decides among classification methods. The dimension reduction sub-task also contains a decision, where the analyst chooses how many principal components to use. We discuss these structural concepts in more detail, as well as how they motivate a different type of dynamic document, in Section 1.2.

1.1.2 Exploring housing prices in the San Francisco Bay Area 2003-2006

In our second example, another analyst is tasked with investigating houses in the San Francisco Bay Area which were sold more than once in the mid-aughts using a census of housing sales data from April 2003 to June 2006. Each observation consists of the date and price of a sale, as well as a number of attributes of the house sold (e.g., number of bedrooms, address, city, county, geographic location in longitude and latitude, interior and lot size, and age of house).

The analyst first assesses the quality of her data by inspecting each variable for obvious problems which could impact her findings. She attempts to correct any values which are obviously erroneous, and sets incorrect values she is unable to correct to missing. The analyst starts with the variables for lot and building square-footage (`lsqft` and `bsqft`). Using a numeric summary, she identifies a number of highly suspect lot square-footages, including the minimum (19) and the maximum (418,611,600) which seem obviously erroneous.

The analyst must choose a low and high value cut-off for values she will accept as correct. Through some external research via the Web she finds that so-called “eco homes,” some of the smallest modern houses in the world, require lot spaces in the mid-double to low-triple digits (e.g., 60 sqft, 240 sqft). The smallest lot square-footage value she is able to corroborate via external listings is 355 sqft. She chooses this as her lower cutoff, making a note of the URLs for the page where she found information about eco homes and for the Internet listing where she confirmed the 355 sqft lot-size for a house in the dataset.

The analyst is more comfortable thinking about large lots in acres. She looks up the conversion between acres and square-feet (43,560 sqft = 1 acre) before considering upper-bounds for lot square-footage. The reported acreages range from less than one up to 9,610. She spot-checks a handful of houses with more than a few acres against external data. The values she checks below 200 tend to be corroborated, while none of the values above are. She notes that Web scraping or available APIs might be used to automate the checking process, but does not pursue that strategy now. She chooses 200 acres as her cutoff for assuming a listing is erroneous. She also thinks that properties with large acreage (farms, ranches, vineyards) might be different in some important ways from urban or suburban homes. She decides not to exclude properties with large (sub-200 acre) lots, but makes a note to herself to revisit this decision if warranted by downstream results.

Next, the analyst determines limits on building square-footage (`bsqft`). She sets a minimum of 400 sqft and a maximum of 13,000 sqft. As with lot square-footage, she arrives at these limits via a combination of general external research and specific spot-checking of observations with extreme values against Internet listings for the property.

The analyst notes that lot square-footage restricts building square-footage as well. Building square-footage can be larger than lot square-footage for multi-story residences, but they are unlikely to have more than three or four stories at the extreme. The analyst decides to consider any building square-footage more than four times the listed lot square-footage to be an error.

For the price variable, the analyst does not find any obviously erroneous values. The minimum ($22,000) and maximum ($20 million) appear to be corroborated by Web listings, the latter being a large winery in Napa valley. The analyst decides to leave price as is.

For city, the analyst finds that there are separate values for “Belvedere/Tiburon”, “Belvedere/tiburon”, “Belvedere”, and “Tiburon”. She learns that Belvedere, CA and Tiburon, CA are “twin cities”, historically distinct cities that have grown together. She decides to combine all four values, but notes that only correcting “Belvedere/tiburon” to “Belvedere/Tiburon” might also be a valid approach. She also fixes some obvious typos, e.g., “oakland” to “Oakland”.

For construction year, the analyst corrects non-ambiguous two-digit years into four digit values (e.g., 99 to 1999). She sets ambiguous two-digit years (0 could be 1900 or 2000) and years larger than 2006 as missing.

The analyst then briefly detours to get a better sense of how house prices are distributed in general before looking more directly at houses which were resold during the time frame. A plot of sale price’s marginal density (Figure 1.7) shows that it is extremely positively skewed. The analyst inspects the data’s empirical quantiles to investigate further, finding that 90% of the houses sold for less than about 1 million and 99.9% sold for less than about 3.5 million.

The analyst finally turns her attention to the goal at hand. Her task is exploratory: investigate the characteristics of houses which were sold more than once in the time-frame. She first identifies houses which were bought and then resold within the dataset and calculates the profit for each resale. She finds that some addresses were listed as sold numerous times in the four year period, up to 33 times or more than once every two months. She investigates and finds that this is an apartment complex where the units were all listed under the same street address. She goes back and filters for houses which were sold between two and five times in the dataset. This leaves her with approximately 26,000 observations.

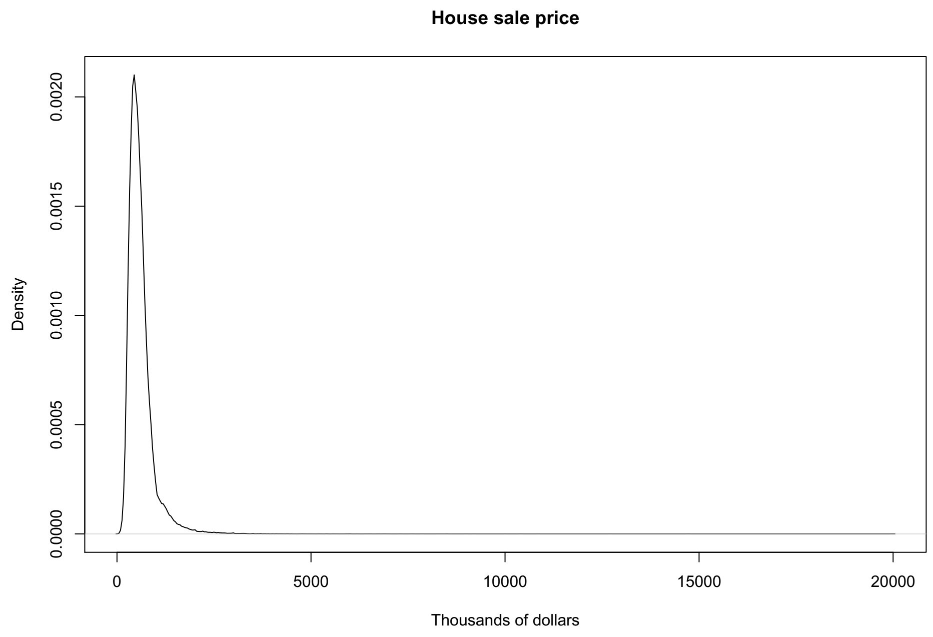


Figure 1.7: ***Density of sale price (in $1000s) for all Bay Area houses 2003-2006.*** A simple density plot of sale price across all observations in the dataset confirms the analyst's suspicion that price is highly skewed. She will keep this in mind when designing future plots involving price.

After cleaning the data, the analyst starts with an inspection of the profits from house resales. She finds that profits ranged from -3.6 million to 4.65 million dollars, and were positive on average with a mean of $102,000 and a median of $84,000. As a percentage of the previous purchase price, these sales ranged from 92% loss of value to 890% profit, with a median of 19% profit and a mean of 24%.

The analyst then looks at the amount of time each house was owned before it was resold. She finds that some houses were "resold" after zero days, and concludes that these are likely the same sale listed twice. In fact, she finds that a fair portion of the alleged resales are after suspiciously short amounts of time and for the same amount both times.

The analyst decides that she will consider 90 days to be the minimum time for her to consider two sales legitimately different. She revisits her code for identifying resales, deciding to include only sales in which the house was held for more than 90 days and had different sale prices listed in the two observations. She makes a note that she has chosen 90 days, but that this is arbitrary and other limits such as 60 or 120 days might be equally reasonable. This reduces her number of resales from 26,000 to 19,000.

With her revised dataset, the analyst recalculates the profit summary. The minimum and maximum for both absolute and relative profits remain unchanged. The averages, unsurprisingly, differ with new means of \$136,500 and 32%, and medians of \$115,000 and 27% for absolute and relative profits, respectively.

The analyst decides to investigate the relationship between location and house resales. She creates boxplots of the relative profits from each county; first she tries standard boxplots before settling on a violin-plot with the median and outliers overlayed (Figure 1.8). Some counties had larger absolute ranges than others, but this was reflected primarily in the outliers. The behavior for non-outlier data was quite similar across the counties.

Next, the analyst investigates how often houses were resold in each of the counties. She starts by looking at absolute numbers of resales, but quickly realizes that this is misleading because the counties did not have comparable numbers of total sales. She considers percent of total sales which were resales instead, again by county. She finds that the county-level resale percentages ranged from approximately 5% for San Francisco County to about 8.7% for Solano County.

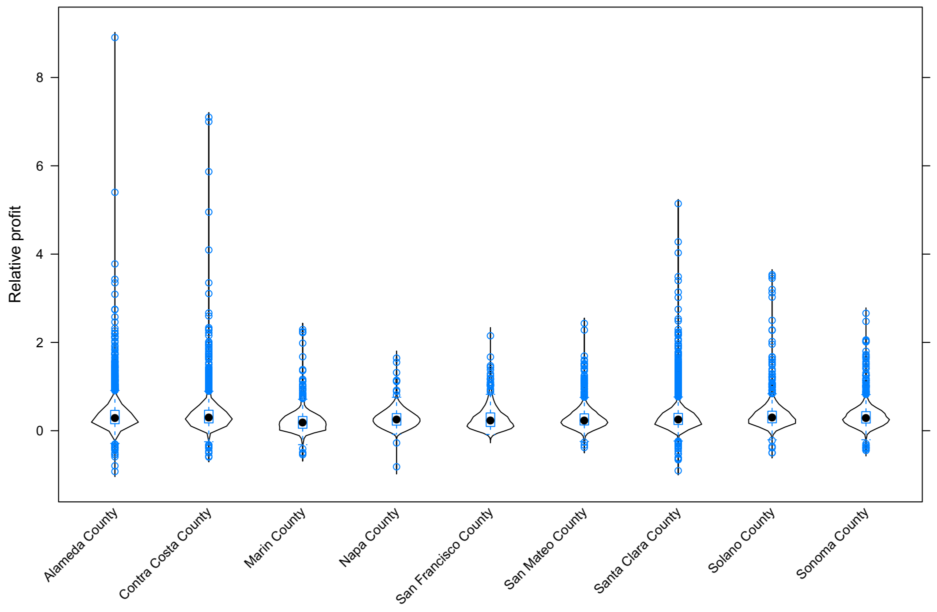


Figure 1.8: ***Distributions of relative profit from house resales by county.*** A violin plot of the relative profit from house resales by county shows that while the range and number of outliers differ dramatically between counties, the core distributions appear largely the same.

The analyst investigates resale rates by city. There are too many cities to create easily readable boxplots, so she relies on numerical summaries. The resale rates for cities with at least 100 sales range from 0% (no resales) up to a maximum of about 10%. The mean and median city resale rates were both just under 6%, with the median slightly higher (0.1%) than the mean.

After investigating the range of resale rates by city and county, the analyst decides to look for a relationship between resale rate and the price of residences in the area. She considers three general ways to assess typical price for a location: median, mean, and trimmed mean of house prices. She decides to use the median for both cities and counties. A plot of median house price against resale rate for all cities (Figure 1.9) does not show any obvious relationship at the city level. The analyst retains the plot, however, to keep a record of this negative result for both herself and anyone else who might review or study her work.

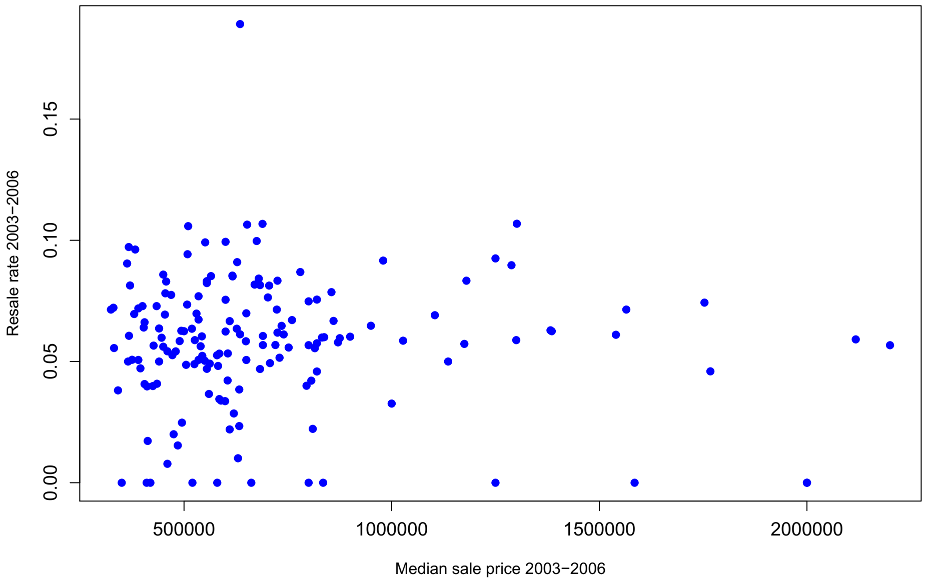


Figure 1.9: ***Resale rate vs median house price by city.*** A plot of resale rate against median house price for each city in the data. The analyst concludes that there is no obvious relationship between typical house price for a city and how often houses were resold there.

The analyst finds a strong negative relationship between county median price and resale rate, however. Plotting these values against each other (Figure 1.10), the downward trend of resale rates is obvious, although the relationship is among many fewer points. She corroborates the relationship by calculating the correlation (-.78), meaning that in general the more expensive counties had the lower resale rates and the less expensive counties had the higher rates. The analyst makes a note of this and will revisit it when she is considering and interpreting her results.

Finally, the analyst decides to investigate sales which resulted in no net profit. Subsetting her dataset of resales, she finds that 767 houses were resold without net gain; 508 of these were sold for a monetary loss. She notes that 508 sales is a tiny fraction of all sales in the full dataset (approximately 0.2%), but constitutes over 2.5% of the roughly 19,000 resales.

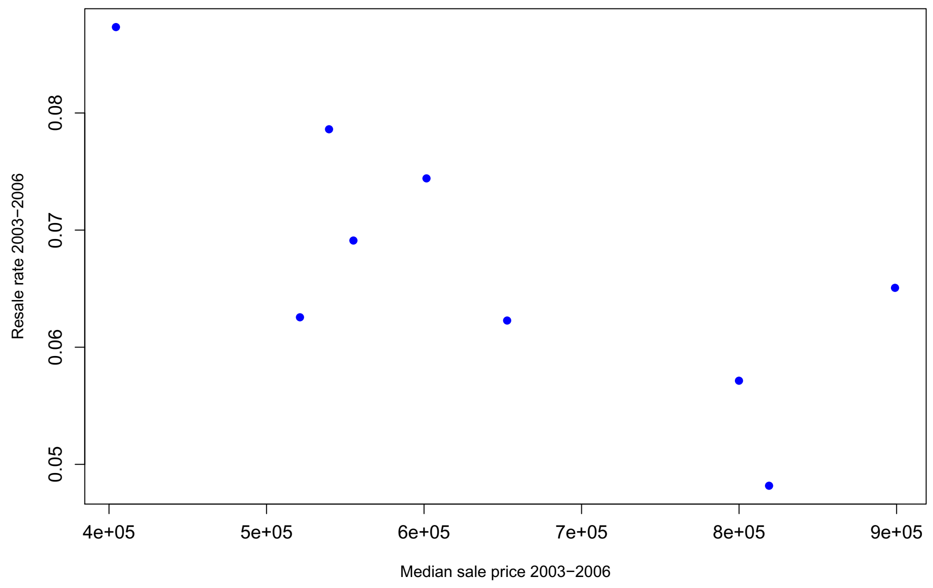


Figure 1.10: ***Resale rate vs median house price by county.*** A plot of resale rate against median house price for each county. This plot shows a clear and strong negative linear relationship. More expensive counties had fewer resales.

The analyst decides to look at rates of sale for a loss *within resales in general*. There are not enough observations for her to investigate at the city level, so she focuses on differences among the nine counties. The percent of resales resulting in a loss range from nearly 1.7% (Contra Costa) to around 5.2% (Marin). All but two (Marin and San Mateo) are between 1.6% and 3%.

The total resale rate had a strong relationship to price (by county), and so the analyst investigates whether resales-for-a-loss have a similar relationship. In fact, generating the analogous scatterplot (Figure 1.11), shows her an almost perfectly opposite relationship. The analyst again confirms by calculating the correlation between her variables, finding a slightly stronger relationship (0.84).

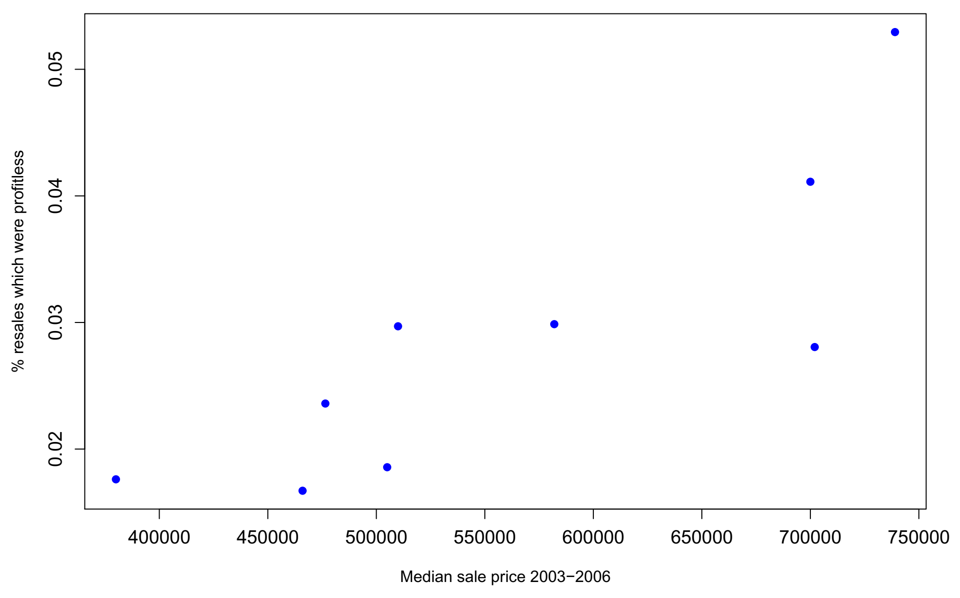


Figure 1.11: ***Percent of resales resulting in no profit vs median house price.*** The relationship between % of resales which generated no profit and median sale price is nearly the opposite of that between overall resale rate and price. More expensive counties had substantially higher percentages of their resales generate net losses.

The analyst steps back and takes stock. She has found that more expensive counties have lower rates of resale, but when houses are resold it is more often for a loss. She makes notes of possible interpretations of this for further thought and use in her write-up:

1. Buyers in extremely expensive areas may tend to buy above their means and find themselves unable to pay their mortgages more often than buyers in less expensive regions. People in this position are more likely to sell a house at a loss.
2. Prices in expensive areas were falling during this time period.
3. All other things being equal, people purchasing in expensive areas have less reason to move, and so would tend to do so only when forced to by circumstance. The less expensive an area is, the more a purchaser might want to "trade up" if things go well.

We choose to stop recounting the analyst's work here. As in our first example, more investigation would occur in a real data analysis setting, but we have enough material to motivate our next section.

1.2 Representing the Research Process in a Dynamic Document

Having seen the work of our two analysts, we turn to the task and implications of representing that work in a single document for each analysis. We focus on conceptual details here, leaving implementation to later chapters.

We have two goals throughout this thesis: investigate how best to fully and reproducibly represent the data analytic research process, and explore the types of novel capabilities these representations can grant to the authors and consumers of the research. We use our sample data analyses to motivate each concept.

We start with a description of the information we wish to capture in these documents: all work performed by the analyst(s) during a project, relationships between groups of content or actions by the analyst, and semantic information about the actions, decisions, code or text contained in the document. We then describe what can be done with such documents: selecting a narrative or subset to process from the full document, computing on the document to analyze the research or its results, and exploring the full research process interactively.

1.2.1 Comprehensively describing the research process

Early in the handwritten digit analysis we described above, the analyst performed LDA on the ‘1’ and ‘8’ classes using the raw data. A journal article describing his results, however, would almost surely make no mention of this.

The code applying LDA to the raw data didn’t even run correctly, throwing an error due to nonsingularity of the sample covariance matrices for the two populations. Nonetheless, attempting LDA was an important step in the research process. The error thrown when evaluating this code caused him to look more closely at his data, eventually finding that many of the pixels far from the center of the 28x28 grid were constant with value 0 – indicating no ink – across all the observations in the data. From the presence of these constant pixels and an exploration of colinearity among the remaining non-constant pixels, the analyst inferred that he could perform some dimension reduction with zero information loss, and could probably reduce the dimension greatly with only minimal loss.

The choice to use principal components to transform the pixel data, then, was motivated directly by the analyst's attempt to use LDA with the raw data. He gained further insight from his exploration of other methods in the '1' vs '8' classification portion of his analysis, narrowing down his list of possible classification methods for the full data from five (LDA, QDA, KNN, rpart, and random forests) to two (KNN and random forests).

Consider a dynamic document (code + text) which presents a journal article style discussion of this digit classification analysis. The discussion would treat as implicit the insights gained from the analyst's two-class exploration of possible classifiers, with little or no mention of how he arrived at them, narratively or via code. Readers would be able to confirm that the analyst's code generates the reported results, but unable to validate his choice of strategy.

To fully validate the analyst's results, we must verify both his code and his decisions which underlie that code. This requires access to more than simply a concise script which transforms the raw data into the final results.

We saw when considering write-ups of the digit analysis that only the full, auditable journal of the analyst's work was sufficient to provide reproducibility for all possible discussions of the work and its results. We refer to this feature as *comprehensiveness*. Comprehensive dynamic documents act as databases characterizing all actions performed or considered by the analyst, including dead ends, alternative methods or implementations, explorations, comparisons (implicit or explicit), diagnostics, and confirmatory calculations. Specifically, a comprehensive document for the digit analysis would contain the code and other details necessary to reproduce a write-up describing the two-class exploration and then discussing the insights gained. Thus the reader can verify the analyst's research fully, rather than only verifying that the chosen methods do produce the reported results. This brings reproduction of computational research closer to the fully independent recreation of experiments possible in laboratory-based sciences.

Furthermore, a comprehensive document provides readers with the information necessary to extend the author's research. When applying the original author's strategy to new data, researchers can see the decision-making process and assess whether their new data exhibits similar enough behavior for them to reasonably make the same strategy decisions as the original analyst. For example, the researcher's new data might be multi-variate Normal, suggesting that non-binary versions of linear and quadratic discriminant classification methods are worth re-considering. In all, this provides a much safer and more reasonable mechanism for applying existing analysis strategies to new data than simply calling the same R functions with new data.

1.2.2 Decisions, alternatives, and tasks

The digit analyst explored two classifier methods (random forest and KNN) on the full data and chose between them to select his final strategy. The set of methods he considered and why he ultimately chose KNN are crucial aspects of the research process and speak directly to the weight we might give his results.

We model an analyst's choice between two or more considered options (even if not all are fully implemented) with the related concepts of *decisions* and *alternatives*. An *alternative* is the set of content – code, text, output, and other components – for a single option considered by the analyst. A *decision* is simply a group of alternatives which encapsulates a particular, localized choice the analyst made. In the context of our digit example we have a decision representing the choice of classification method, with alternatives representing each of the two methods our analyst considered for the ten-class data (KNN and random forests).

By grouping the alternatives into decisions, we encapsulate information understood by the analyst: that the content and results corresponding to different alternatives in the same decision are related. More specifically, results generated by alternatives will generally be attempting to perform the same conceptual action, and thus can be meaningfully compared. In our classification method example, that action would be to create a classifier, and thus we can compare the results by looking at the classifiers' performance. In fact, we will do this explicitly when we discuss computing on the document in Section 1.2.5. Before moving on to that subject, however, we discuss our motivations for explicitly including decisions and alternatives in dynamic documents.

Authors and readers can use decisions to describe and differentiate between narratives. Consider again our digit example. An article presenting the analyst's results would focus heavily or exclusively on the final KNN classifier. Thus, we can think of constructing this narrative by *selecting* the alternatives representing the final strategy he chose. That is not the only meaningful narrative from the analyst's work, however. A hypothetical narrative discussing the results if the analyst had chosen to use random forests can be constructed by selecting the random forest alternative instead. This generalizes naturally to a concept of a *thread* or linear path through a document, which we define more explicitly in Section 1.2.4 and will use extensively throughout this work.

Human readers can sometimes infer decisions and alternatives given a comprehensive list of code (and text), but this can become difficult and tedious for large or complex analyses. Having the decision and alternative information explicitly available both programmatically and to human readers avoids this confusion while offering new opportunities for both displaying and computing on the document. Modeling the decision process within dynamic documents also allows authors to identify and reorganize implicit decisions. Explicitly considering the tasks, decisions, and alternatives that make up an analysis can benefit the author in two ways: it can crystallize the strategy in his or her own mind, and it ensures that (s)he can effectively communicate the strategy – and his or her justification for selecting it – to others.

Furthermore, the concepts of decisions and alternatives provide a natural way for interested researchers to extend an analysis by exploring a different method or strategy. Instead of replacing the existing code in a linear document, the new strategy can simply constitute a new alternative placed alongside the existing work from the original analysis.

A processing system that understands decisions and allows users to select specific alternatives is key for fully leveraging comprehensive documents that include decision information. Such a system can translate a simple concept such as "use the random forest alternative for the final classifier" into a concrete set of computations to use when generating an output report (e.g., a PDF or HTML file for human consumption) automatically, thus increasing the utility of comprehensive documents.

We note that decisions can occur anywhere during the research process, including within a specific alternative from a previous decision. Choosing the KNN classifier led the analyst to another choice: the value of k (1, 3, 5). If he had chosen random forests he would not have needed to choose k. We say that the choice of k is *nested* within the choice of classifier method.

Decisions can allow referees, collaborators, and even authors revisiting previous work to assess the due diligence performed by the analyst. Questions of the form "Did you try ...?" are easily answered. Motivated viewers can retrace the steps of the research process, viewing the intermediate results available to the analyst when the decision was being made.

We call groups of content which combine to represent a single higher level conceptual action, but which are not alternatives within a decision, *tasks*. Tasks often represent abstract concepts, e.g., dimension reduction, data cleaning, exploring two-class learners, generating final results. Tasks are similar to alternatives, but their uses are very different. Tasks are typically combined sequentially to define an overarching research workflow, while multiple alternatives within a single decision are not generally used simultaneously in a single narrative.

Decisions, alternatives, and tasks represent important types of semantic and structural information about the analysis described in a dynamic document. We now expand this to associating semantic information with content in a document more generally.

1.2.3 Metadata about document elements

Additional information is sometimes helpful when determining how particular actions by an analyst fit into a larger data analysis. Analysts can provide this context by annotating content in a document with semantic information. Consider our two example analyses.

The digit analyst's code which attempts to apply LDA to the raw '1' and '8' data throws an error when it is evaluated. This error was the impetus for the analyst to use dimension reduction. When processing or viewing a document, it is important to understand that the error is expected, and not an indication that the processing has failed. Annotating the relevant task in a programmatically accessible way allows the author to convey that information to both the processing system and the reader.

In the house resale analysis, the alternatives where the analyst uses the median to calculate typical house price by city and by county represent the same overarching strategy, even though the decisions choosing the methods central tendency – median, mean, or trimmed mean – for city and county prices are structurally distinct and computationally unrelated. It would be strange, for example, to use mean for cities and median for counties. This type of inter-decision relationship between specific alternatives can provide important context when selecting alternatives for constructing a thread or narrative. One way to annotate documents with this type of relationship information is via *thread-ids* that are the same when alternatives represent the same conceptual choice at different points in the analysis. In our example, we might have three thread-ids – `mean`, `median`, and `trimmedMean` – representing the three measures of central tendency the housing analyst considered.

Another possible use of metadata is to represent different *resolutions* within the document. Resolutions refers to the level of detail displayed when rendering a document or narrative. Certain computations, results, and even entire tasks can be rendered within some narratives but hidden in others. A prime example of this is the two-class exploration task in our digit analysis. The content and results of this exploration would be shown in a full recounting of the analyst's work, but would likely be hidden in a journal article describing his results. One way to represent the concept of resolutions is with different *detail levels* which can be toggled on and off during the processing or rendering stages. We discuss the concept of detail levels and their uses more in Section 1.2.4 and will use this concept throughout the remaining chapters.

Processing systems which support the concept of multi-resolution documents can construct both concise and detailed narratives from the same document by automatically showing or hiding content at different detail levels. Furthermore, interactive viewing systems, such as those we outline in Section 1.2.6, can allow users to dynamically change the resolution at which they are viewing a document.

More generally, document authors might want to include information which is important to record, but doesn't belong anywhere in a typical (even highly detailed) narrative. Examples of this might include detailed information about the data generation process, e.g., the MIAME (Brazma et al., 2001) and MIAPE (Taylor et al., 2007) specifications used in biology, technical information about the system the code was initially developed on, or original sources of the data used. These pieces of information could be included in a way intended to be programmatically accessed instead of rendered directly. To support this, we need the ability to mark these pieces of information in a way processing systems can be designed or extended to understand.

We abstract these specific annotations in the concept of *metadata*. Metadata are arbitrary key-object pairs associated with content in a document, similar to the chunk options of dynamic document systems such as **knitr** and SWeave, or node attributes within XML. A key feature of metadata is that, like the information about decisions and alternatives in the previous section, it is programmatically accessible. Thus, it can be used when processing the document into a narrative or report. In Chapter 2, we describe some particular pieces of metadata and how our **DynDocModel** package uses them when processing comprehensive dynamic documents.

1.2.4 Selecting linear narratives and other subsets from comprehensive documents

Suppose our housing analyst wanted to generate two narrative PDF or HTML reports describing her analysis using her comprehensive document. One uses the median to assess typical price, omits the univariate exploration of price, and uses the 90 day cut-off for defining resales. The second uses a trimmed mean to assess typical price, includes the exploration of price, and uses 120 days as the resale cut-off. They both use the same data-cleaning strategies.

These two narratives correspond to different paths through the comprehensive document. Conceptually, we can imagine a path as a series of choices about whether to include or exclude specific content. In practice, this typically constitutes the choice to include or exclude optional or high-detail content – e.g., the price exploration – and the choice of a single alternative for each relevant decision.

We call the subset of content corresponding to a particular linear narrative the *thread* through the document for that narrative. The content in a thread need not appear sequentially in the full document, but the code portions must be able to be evaluated in the order in which they appear in the thread. Each of the static narrative reports the analyst is generating requires the extraction and processing of a corresponding thread from the full document. This requires a processing system that allows dynamic documents to be subsetted either before or during processing (or both).

Subsetting can be done by concept (task), whether an element is visited by a particular thread, metadata, content, content type, or any combination thereof. R's extensive subsetting machinery (R Core Team, 2014) and the XPath query language for XML (Clark and DeRose, 2006) provide two very different models for subsetting. For the implementation we have chosen to pursue, the XPath-like approach is more apt. We discuss this approach in some detail in the next chapter.

Metadata will often play a role in the subsets we wish to identify. This is the case in both the *thread-id* and *detail level* cases we described in the previous section. Thread-id values can be used to identify all content associated with a particular overarching strategy or concept. We note that this may or may not define an actual thread. In our housing example, the `median` thread-id would not fully specify a thread; we would still need to select alternatives for all the decisions unrelated to assessing typical price. Detail level is more likely to be used programmatically to exclude portions of the document when rendering a document or thread at a particular resolution, but it can also be used directly by the user to identify and subset content.

We also note that the mechanism of processing a dynamic document – which we leave abstract for now and discuss in detail in Chapter 2 – necessarily requires subsetting as well. Before the code elements within a dynamic document are evaluated to generate new output, they must be identified. This constitutes subsetting. Thus all dynamic document systems are already centered around the concept of subsetting; we are simply generalizing this subsetting step to allow users to identify portions of the document to include or exclude during processing. The concept of threads – and our mechanisms for declaring and extracting them – provides a useful abstraction for this general subsetting step. We discuss this abstraction, as well as how it fits into the model we have implemented, in more detail in the next chapter.

1.2.5 Computing on the document

Beyond providing reproducibility for an analyst's research process, comprehensive documents act as data about the described analysis workflow and its results. Authors and independent researchers can use these documents to ask questions about the author's actions, decisions, and results.

We can investigate how many different final classifiers are possible via combinations of the alternatives the analyst considered and how they compare. In this case, each distinct thread (up to inclusion of, and choices within, exploratory computations and tasks) will generate a different classifier. By querying the document for decisions (or simply generating all these threads) we would see that there are two implemented dimension reduction strategies (principal components using 25 and 150 components) and ultimately four classification methods (random forest, and knn with k=1, 3, 5). Thus there are eight sets of possible final results encompassed by the analyst's work, one actual and seven hypothetical.

We can go further, evaluating these eight threads and collecting, e.g., the eight overall misclassification rates on the independent testing data. We can then use these rates in further computations (they are simple *numeric* objects in R). For example, we could create a plot comparing the eight error rates to explore how method and number of principal components used seems to affect performance for our data, allowing us to assess how sensitive the analyst's results are to the specific choices he made during his analysis. Doing this, we would arrive at Figure 1.12. We discuss the details of generating this plot, as well as computing on documents more generally, within the context of presenting our **DynDocModel** package in Chapter 2.

Now suppose we want to verify the digit analyst's work. We might first analyze his comprehensive document to determine which R packages we must have installed before initiating potentially expensive computations. We can do this by identifying all the code in the document and analyzing it, e.g., via Temple Lang's **CodeDepends** package (Temple Lang et al., 2013), to detect calls to `library()` and `require()`. This allows us to identify that we need to have the **gplots** (Warnes et al., 2014), **RColorBrewer** (Neuwirth, 2011), **rpart** (Therneau et al., 2014), **parallel** (R Core Team, 2014), and **randomForest** (Liaw and Wiener, 2002) packages available before we can proceed. We can use the same process to determine the requirements to run a particular thread, e.g., the thread that corresponds to the final article-style narrative.

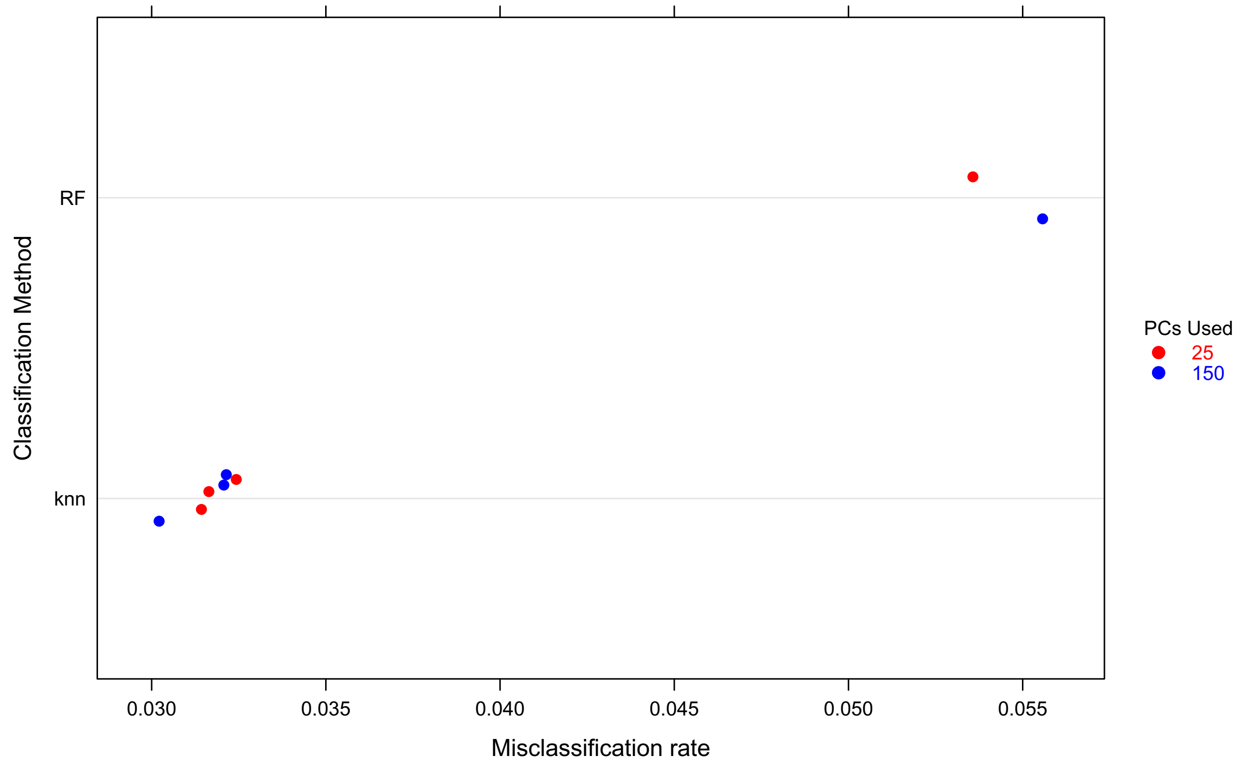


Figure 1.12: ***Misclassification rates by method and number of principal components used.*** To explore how the different final classifier candidates compare, the analyst plots their error rates together in a single plot. Note that this plot is not created by processing a single thread through the document, but rather by interactively computing on the document itself.

1.2.6 Exploring the analysis interactively

We saw in Section 1.2.4 that authors can use different threads through a dynamic document to generate different static output reports. These can be targeted at different audiences by showing different levels of detail, or used to explore hypotheticals by making different choices throughout the analysis process and showing the results.

Readers are likely to find additional detail even more valuable if they can hide and show it on command without reprocessing the entire document. For example, a student reading the digit analysis might want to view the plot of the single observation to get a more concrete understanding of the data, only to hide that content again once that understanding is gained.

Given an interactive way of rendering comprehensive documents, readers can also navigate between specific alternatives or entire threads. This allows us to step through the analysis, inspecting or even changing the analyst's choices and viewing the corresponding output throughout the rendered document.

Finally, we can leverage a mechanism for interactively displaying a comprehensive document to allow authors to associate graphical user interface (GUI) controls to code elements within the document. For example, the digit analyst might associate a slider control with the KNN classifier instead of simply having three alternatives with different values of k. This allows the reader to explore the effect of k on the result for a wider range of values, e.g., to check for robustness of KNN's results, without cluttering the document with dozens of specific alternatives.

We discuss a specific interactive viewing environment, which doubles as an authoring environment for comprehensive documents, in Chapter 3. We now conclude this chapter with an outline of the remainder of this thesis.

1.3 A Roadmap for the Remainder of this Thesis

In the remaining chapters of this thesis we describe our contributions to the field of dynamic documents. We have solidified, explored, combined, and implemented various ideas proposed in (Gentleman and Temple Lang, 2007), (Nolan and Temple Lang, 2007) and (Nolan and Temple Lang, 2014). In Chapter 2 we develop an object model for comprehensive dynamic documents as well as a computational model for computing on and processing these documents in R. We also present proof-of-concept software which implements these models. Next we present a case study for modifying an existing interactive authoring and computing platform (IPython Notebook (Pérez et al., 2013a)) to support comprehensive, non-linear documents in Chapter 3. Then, in Chapter 4, we present a caching system designed to facilitate interactive exploration of non-linear dynamic documents.

Our **DynDocModel** R package (Chapter 2) implements two computational models: an object model for representing arbitrary linear and non-linear dynamic documents within R, and a model for processing such documents which encompasses both currently standard and novel types of processing. Via these two models, our package offers full support for reading, processing, computing on, and writing non-linear documents in various file formats.

Our object model is format agnostic, allowing dynamic documents to be represented in the same data structure regardless of input and target output format. This model allows us to design and implement algorithms and methods for computing on dynamic documents once and apply them across documents stored in various formats. The object representing a dynamic document is also available to the user, allowing users to compute on the documents themselves, as we saw in our scatter plot of misclassification rates above. This allows authors and readers alike to analyze a body of work, extracting newly synthesized information from the full body of work done during a research project.

For the document rendering model, we focus heavily on customizability and extensibility. Users are able to customize behavior at each step of the standard dynamic report generation (*weaving*) process (parsing, content selection/subsetting, evaluation, and rendering). We show that current processing mechanisms, including weaving and code extraction (*tangling*) are achievable via our model. We also give example of customizing each step within our model to illustrate the flexibility our system offers. Customizable parsing allows users to easily add support for new types of dynamic documents without losing any functionality offered by the package. Content selection is crucial for non-linear documents, as it is what allows us to specify a thread to be woven into a linear report. With custom evaluation, users can modify the type of output to be generated. Finally, **DynDocModel** supports custom rendering and formatting of text, code, and generated output based on a custom dispatch mechanism operating directly on the output objects. When taken together, our model grants full control over both what appears in the output from processing a dynamic document and how that content is included or displayed.

DynDocModel also supports the concepts of multiple resolution documents and interactive code elements. We implement multiple resolutions via the *detail level* metadata field discussed in Section 1.2.3, while interactive code elements are specified via a narrow but useful widget abstraction. **DynDocModel** does not contain a way to directly display interactive code elements, but it supports outputting both full documents and woven reports for viewing in our modified IPython Notebook platform, which does support widget controlled code elements.

Our modification of the IPython Notebook platform (Chapter 3) serves both as proof-of-concept software for creating and interactively exploring non-linear documents, and as a case-study in the design challenges for building or modifying systems to support such documents. As such, we present specific decisions we made when creating the software, as well as the lessons learned about the larger issues facing developers looking to support non-linear documents. We also discuss the successes and drawbacks of designing a single system which acts as both a document authoring tool and an exploration environment.

Our non-linear IPython Notebook platform allows authors to iteratively construct and evaluate linear and non-linear dynamic documents. In fact, we used the platform to author comprehensive documents describing both of the analyses we discussed earlier in this chapter. By allowing analysts to construct non-linear documents during the analysis and authoring steps, we provide a more convenient mechanism for creating dynamic documents (when compared to combining multiple separate scripts piecemeal after the fact). The platform also allows authors and readers to perform basic interactive exploration and execution of non-linear dynamic documents. This includes navigating between threads by selecting alternatives, switching between resolutions of the document by changing the displayed detail level, (re)running the currently active thread, and using the interactive code element GUI controls to explore different parameters and configurations.

Finally, our **RCacheSuite** package (Chapter 4) implements a form of result caching which supports multiple caches for a single set of R code expressions within one or many documents. This is key when dealing with non-linear documents, as different threads will often share some code elements, but generate different values when evaluating the code expressions within these shared elements. In particular, this type of multiple caching is crucial when switching between alternatives or threads within a document in real time during interactive explorations. We achieve this by proposing a new caching mechanism which takes the current values of variables used by an expression into account when creating new caches and matching against existing ones.

Finally, we conclude the thesis with a high-level description of the work we have presented and its contributions to the field. We first summarize the discussion and identify the key points from each of the projects we presented. We then briefly discuss availability of the software described in this dissertation before concluding with a summary of how our three main avenues of research fit together to contribute to the current understanding of dynamic documents in particular and the description of scientific research and its results more generally.

The material throughout this dissertation is based upon work supported in part by the National Science Foundation under grant number 1043634.

Chapter 2

A System for R-based Nonlinear, Comprehensive Dynamic Documents

In the previous chapter we described two data analyses, one regarding the classification of handwritten digits and one exploring housing sales in the San Francisco Bay Area. We used these to motivate ways in which dynamic documents can more fully and accurately characterize data analytic research, which we briefly recount here.

We saw in both the housing and digit examples that the analysts gleaned important insights from explorations that would be omitted from a concise discussion or recreation of the respective final results. The digit analyst, for example, explored possible classification methods in the context of only two populations ('1's and '8's). By focusing on only two populations first, he was able to better understand his data and rule out methods which did not perform well before moving on to the more computationally intensive and analytically complex full problem. To understand and validate the (digit) analyst's final results it helps to understand and validate the intermediate results and reasoning on which he based his final strategy. This motivated our concept of *comprehensive* dynamic documents, which reflect the full body of work comprising an analysis rather than only the portions necessary to replicate final results from raw data.

Both analysts considered and implemented multiple strategies at points in their research, before selecting one or more methods to focus on moving forward. The housing analyst considered three measures of central tendency when assessing typical housing prices at the city and county levels: mean, median, and trimmed mean. She ultimately chose to use the median in both cases, but the fact that she considered the other two approaches – and the results generated via those methods – provides important context for both her eventual use of the median and the portions of her findings which involved typical housing price. This type of choice by the analyst motivates grouping content into *decisions* and *alternatives* within a comprehensive document. An alternative encapsulates the content (code + text) making up a single possibility considered by the analyst at that point, e.g., calculating and using the mean, while a decision is the collection of alternatives considered by the analyst during the choice. *Tasks* – as described in Chapter 1 – represent groupings of content which encapsulate a higher-level concept implemented or described by the analyst. Tasks within the analyses we presented in the previous chapter include data cleaning, fitting a particular type of classifier, and assessing a classifier's performance on independent data.

When motivating the concept of database-like comprehensive documents in Chapter 1, we noted that the analysts might prepare multiple different write-ups for different audiences. For the digit analyst these might include: an executive summary of the final classifier's performance for a high-level manager, a technical implementation-centric discussion for a team putting his classifier into production, and a full auditable record of his work. Generating or reproducing the results in each of these write-ups requires a subset of the code in the full, comprehensive dynamic document for the analysis. This motivates the concept of *threads*, which are subsets of the comprehensive document that define a linear narrative, or "path through" the research process. We can think of these specific threads and the resulting narrative reports as subsets or projections of the entire body of work. We can also consider and work with non-thread subsets of a document, e.g., by omitting certain self-contained tasks or specific alternatives. These sub-documents can be non-linear in cases when the parent document is non-linear, but they need not be so.

We also saw that computing directly on the documents can help us better understand the research in ways that viewing a narrative thread from data to results might not. In the plot of misclassification rates in Figure 1.12, we showed the overall error rates for the eight possible classifiers given the methods and choices considered by the digit analyst. The plot presents information about how the methods compare for the digit data, which would not be discernible from the result of using any single method. In Section 2.2.3 we walk through the code used to generate and extract these values from the comprehensive document.

Finally, we discussed the concept of interactive code elements. These are elements which can be rendered with graphical user interface (GUI) controls – e.g., sliders, buttons, and menus – that allow the reader to control and re-run computations at viewing time.

We present the **DynDocModel** R package, which implements a unified system for representing and processing linear and non-linear dynamic documents in R, regardless of storage format. **DynDocModel** offers a particular implementation of the ideas we recapped above. We present the features of our package within the context of the two example analyses we discussed previously. We assume for the entirety of this chapter that comprehensive dynamic documents for the two analyses already exist. We present software for constructing comprehensive, non-linear dynamic documents in Chapter 3.

The remainder of this chapter is organized as follows. We first discuss our abstraction and object model in Section 2.1. Here we describe the data structures we use to represent dynamic documents and their elements, as well as the mechanisms for creating and populating them. We also discuss a simple mechanism for defining widgets for use with interactive code elements.

Next, we turn to interacting with documents and elements programmatically in Section 2.2. We discuss visualizing non-linear dynamic documents as graphs before moving on to discuss low-level features of the API for manipulating documents within R, such as querying and modifying documents programmatically.

In Section 2.3, we discuss the components of our high-level weaving API for generating dynamic reports and other output. We illustrate the flexibility and features which our API and computation model provide by weaving output reports using in-context examples from the housing and digit analyses.

Finally, we discuss related and motivating work in Section 2.4. We first provide a brief history of linear dynamic documents for R (and S/S-plus). We then discuss existing work in the realm of non-linear dynamic documents for R including both a general history of non-linear dynamic documents and a discussion of previous investigations into non-linear dynamic documents in an R-based setting. Third, we discuss other related work, including approaches to non-linear dynamic documents within R and a brief discussion of dynamic document work in other languages. We then conclude the chapter with a brief recap of what **DynDocModel** contributes to dynamic documents.

2.1 Representing Documents as R Objects

A central tenet of **DynDocModel**'s design philosophy is that users have programmatic access to the classes and functionality used internally to represent and process dynamic documents. This access allows users to compute on the documents, moving beyond simple generation of reports. To facilitate this, users are provided a suite of both high-level and low-level tools for interacting with dynamic documents via our package. In fact, the high-level weaving API is built exclusively via the exposed low-level API, ensuring that the low-level machinery provided to developers is robust and useful. We first define our object model and then discuss the low-level interface.

2.1.1 Representing decisions, alternatives, and tasks via nesting

Gentleman and Temple Lang note that the order of code and text in a dynamic document *"need not be simply sequential but can support rich and complex document structures. For example, the ordering of the chunks may have branches and generally may form a graph with various paths through the nodes (chunks) that allow different readers to navigate the document in different ways."* (Gentleman and Temple Lang, 2007) We implement the non-linearity they describe by allowing the nesting of content within dynamic documents.

Decisions, alternatives, and tasks represent different types of content groupings within a dynamic document. We define *structural elements* to be document elements which contain nested content and then use this nesting to represent decisions, alternatives, and tasks. We draw inspiration for this model from Nolan and Temple Lang's use of specific XML nodes to signify these features in their Rdocbook (Rdb) extension (Nolan and Temple Lang, 2014) of Walsh's DocBook format (Walsh and Muellner, 1999).

A *decision element* is a structural element which contains the content representing the set of alternatives considered by the analyst. An *alternative element*, in turn, contains the content, i.e. code, text, and nested structural elements, representing a single considered alternative. Similarly, a *task element* contains the collection of content representing a conceptual task.

We call documents which contain at least two distinct threads *non-linear*. This occurs most commonly via inclusion of decision elements, but can also occur via detail levels – or other forms of content exclusion – in documents without branching. Package vignettes and similar traditional dynamic documents, then, are linear, whereas our database-like documents may or may not be (but are typically not).

2.1.2 A unified, format-agnostic representation

The **DynDocModel** package provides a single object model for dynamic documents in R, regardless of input (storage) and intended output (target) format, allowing us to handle Rnoweb, Rhtml, Rmarkdown, XML (Rdocbook) and JSON (IPython notebooks) formats within the same framework. Dynamic documents are represented via the *DynDoc* class. We discuss the creation of *DynDoc* objects – and the customization thereof – in more detail in Section 2.3.3.

The *DynDoc* class is a container for a – possibly nested – set of elements which make up a dynamic document. All classes representing elements within a dynamic document inherit from the abstract/virtual *DocElement* class. These element-representing classes indicate their basic type (text, code, or structural element) by inheriting from one of three mid-level classes: *TextElement*, *CodeElement*, or *ContainerElement*. Each of these basic classes has multiple subclasses which represent specific types of elements within a document. For example, the *RCodeElement* class represents code elements which contain R code, while the *MDTextElement* represents text elements which contain Markdown content. We list these subclasses, and what they indicate about the elements they represent, in Table 2.1. Following that, we present the formal class hierarchy (visualized via Maechler's **classGraph** package (Maechler, 2010)) representing document elements in Figure 2.1.

DocElement objects have pass-by-reference semantics, causing them to behave differently than

Table 2.1: R classes for documents, instances, and document elements by type in **DynDocModel**

Class	**Purpose**	**Base class**
TextElement	contains plaintext content	*TextElement*
MDTextElement	contains text content in Markdown	*TextElement*
DbTextElement	contains text content in DocBook (XML)	*TextElement*
LatexTextElement	contains text content in LaTeX	*TextElement*
RCodeElement	contains parseable R code	*CodeElement*
PyCodeElement	contains parseable Python code	*CodeElement*
DecisionElement	represents a decision among two or more alternatives	*ContainerElement*
AltElement	represents a sing alternative within a *DecisionElement*	*ContainerElement*
AltMethodElement	a subclass of *AltElement* specific to statistical methods	*ContainerElement*
AltImplElement	a subclass of *AltElement* specific to implementation strategies	*ContainerElement*
TaskElement	represents a conceptual task	*ContainerElement*
MixedTextElement	represents raw text containing inline code expressions	*ContainerElement*
MixedMDTextElement	subclass of *MixedTextElement* for Markdown	*ContainerElement*
MixedLatexTextElement	subclass of *MixedTextElement* for LaTeX text	*ContainerElement*
MixedDbTextElement	subclass of *MixedTextElement* for DocBook text	*ContainerElement*
InlineRCodeElement	represents an inline R expression within a mixed text element	*CodeElement*
IntRCodeElement	represents R code that can be rendered with interactive GUI controls	*CodeElement*

regular R objects. This non-standard behavior allows each element to retain an accessible link to its parent (either a *ContainerElement* or the *DynDoc*). With access to the parent of each element we can traverse a document in both directions (up and down), whereas with, e.g., standard R *list* objects we can only traverse in one (downward). *DynDocModel* uses this ability extensively in the thread-related machinery we discuss in Section 2.2.3.

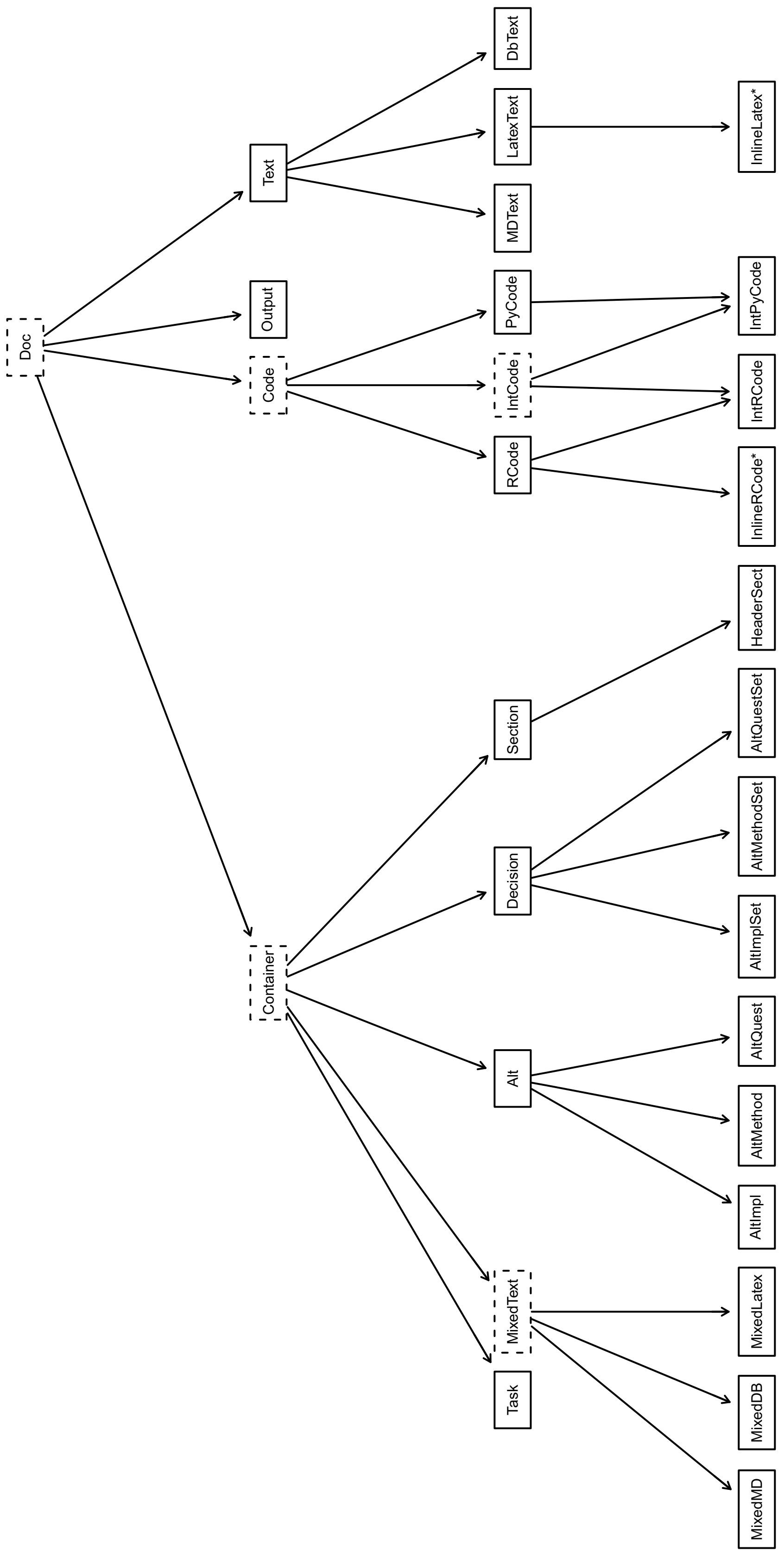


Figure 2.1: ***The hierarchy of DocElement classes.*** All classes, except those marked with *, end in 'Element', which has been omitted for brevity. Purely virtual classes are shown as nodes with dotted outlines.

We define a *thread* as a collection of elements from a dynamic document that defines an ordered linear narrative. Threads correspond most naturally to paths through the directed graph representation of a dynamic document we described in Section 2.1.1. In this formulation, a thread can be mapped directly onto a set of exactly one alternative for each relevant decision.

Threads are sets of elements which can be processed as a single sequential narrative, but the elements making up a thread need not appear that way in the parent document. Threads can represent narratives with some content removed – such as excluding the two-digit analysis in our handwritten numeral example – or even subsets of the document more generally, so long as the organization of elements in the thread is linear.

Threads will be our primary way of representing and then generating linear reports from comprehensive documents. Modeling different threads through the space of possible strategies considered by an analyst – separately or simultaneously – is a key feature of our non-linear, database-like documents. We saw in Chapter 1 that this multitude of potential threads allows us to both generate many distinct reports from the same document and to meaningfully compute on a document to explore the described research. To effectively deal with multiple threads through a document simultaneously, however, we need a way for threads to share elements while still being able to make local modifications that are not shared across all objects representing a particular element, such as associating thread-specific output to specific 'versions' of the element.

We support specific modifications while retaining the overall reference semantics of *DocElement* objects via *instances* (the *ElementInstance* class). An *ElementInstance* object is essentially a reference to a document element (a *DocElement* object) along with output and attributes associated with that element within a specific context – e.g., when the element is being processed within a thread – but which should not be associated with the element in all contexts. Many *ElementIn-*

stance objects can represent the same *DocElement*. Each *ElementInstance* includes a reference to the underlying *DocElement* in addition to instance-level information, such as attributes and associated output. This allows us to construct threads – *DocThread* objects – as collections of instances, rather than collections of elements directly, as pictured in Figure 2.2. *ElementInstance* objects have pass-by-reference semantics for the same reason that *DocElement* objects do.

The distinction between elements and element instances allows us to retain the database-like nature of comprehensive *DynDoc* objects – one central data store serving many "queries" -, while supporting the type of local modifications necessary to process threads and associate output with specific elements within those threads. Any changes to a *DocElement* is immediately propagated to all associated *ElementInstance* objects, while setting attributes on or associating output with an *ElementInstance* is not propagated to the underlying *DocElement*, and thus does not interfere with other instances of the same element. *ElementInstances* are typically created as needed – e.g., when creating a *DocThread* object for a thread -, but can be explicitly created by the user as well.

2.1.3 Modeling local element-level interactivity in dynamic documents

An obvious form of interactivity for our comprehensive documents is navigating between threads within a display of the entire document. We discuss that form of interactivity in Chapter 3. It is also useful, however, to model a more localized version of interactivity centered around the output from individual code elements. Under this model, viewers alter the parameters used in a particular expression and immediately see the results of that change reflected in displayed output. We have implemented a mechanism for supporting this type of interactivity within displays of comprehensive documents and threads within them.

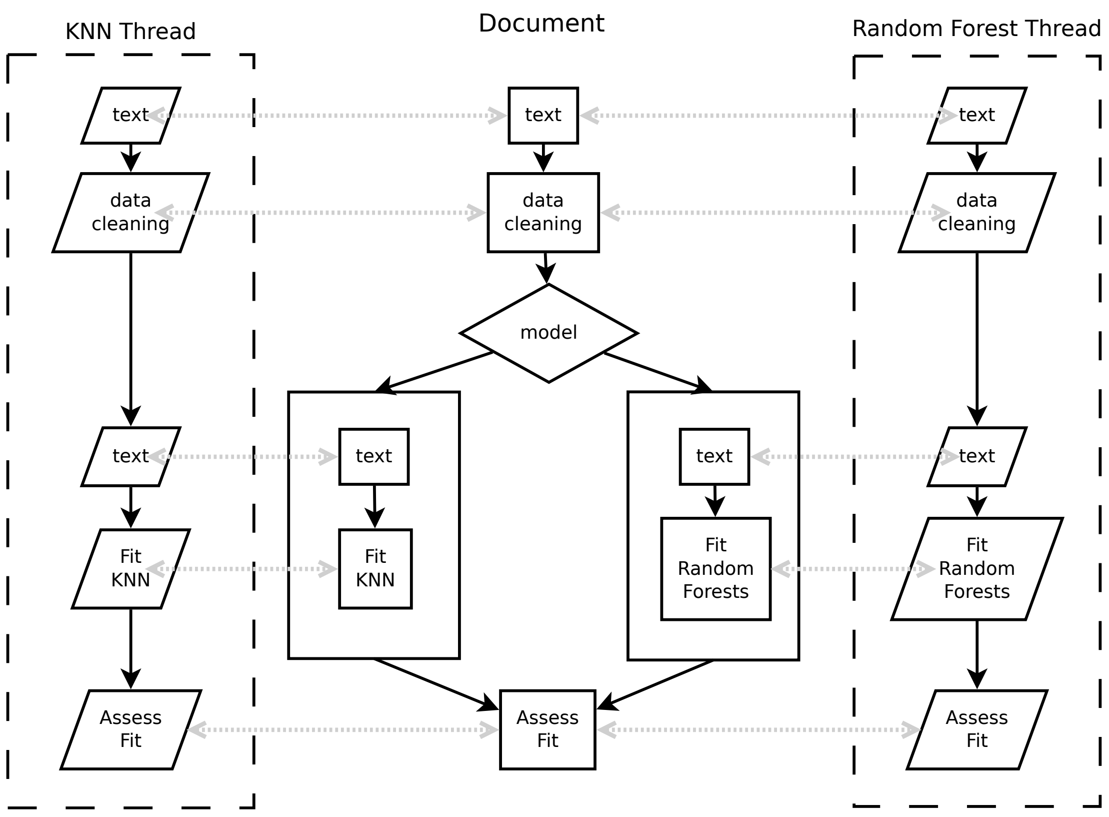
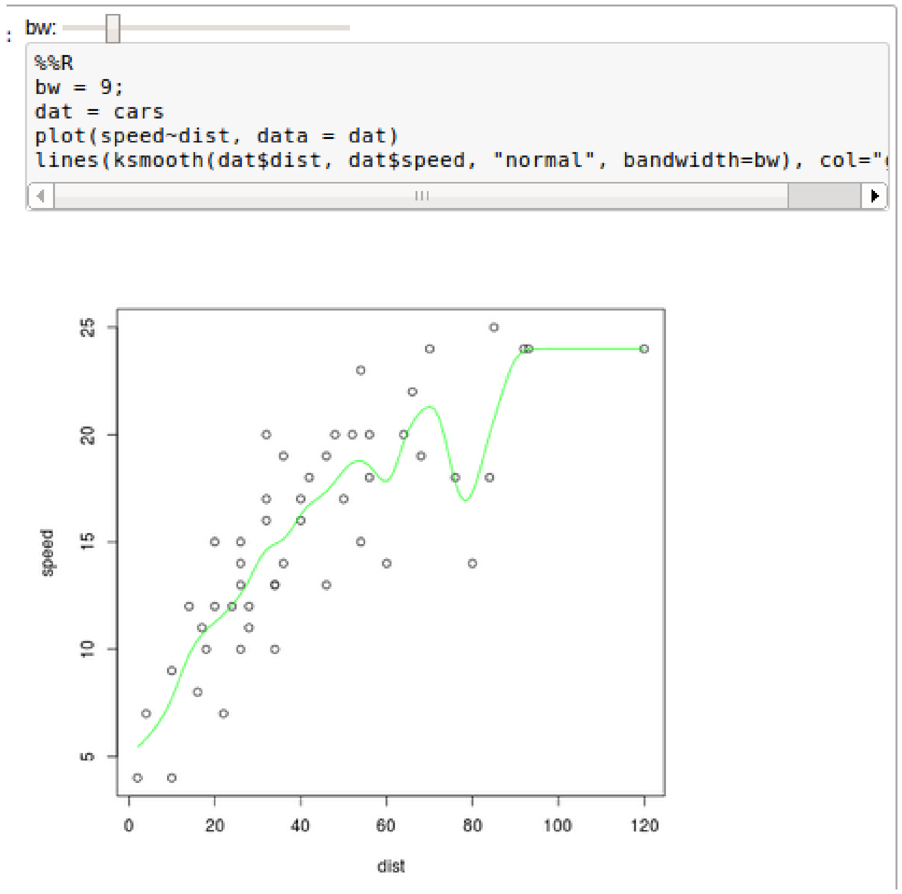


Figure 2.2: ***DocThread, DocElement and ElementInstance objects.*** Threads (*DocThread* objects) are represented as collections of *ElementInstance* objects in the same manner that documents (*DynDoc* objects) are collections of *DocElements*. Each *ElementInstance* object contains a reference to the underlying *DocElement* (pictured in the image above as a dotted grey line). This allows us to, e.g., have multiple sets of output associated with the same code element within different threads.

Consider a non-parametric regression performed via the kernel regression method (Watson, 1964). Robustness to the selection of the bandwidth tuning parameter is an important aspect of the generated result. Robustness can be investigated and summarized in a static plot or table, but it can also be enlightening to explore such robustness interactively. This would be useful for reviewers to quickly assess whether the result represents a general consensus among similar tuning parameter values. Nolan and Temple Lang have also argued that the ability for students to control parameters and see the effects of changing them would be a valuable tool when teaching statistical methods and reasoning.(Nolan and Temple Lang, 2007)

Given access to the code, the audience could alter specific code expressions and then re-run them to view new output, but this would be tedious and require some care to understand the code correctly before proceeding. A more focused and user-friendly approach is to provide graphical user interface (GUI) style controls embedded within an article-like view which hide the details of altering the code, re-generating the output, and updating the display from the casual user. Examples of this could include a mouse-controlled slider which resets the bandwidth of our kernel regression – as pictured in Figure 2.3 – or a drop-down menu which selects which kernel is used when generating the estimate.



```
%%R
bw = 9;
dat = cars
plot(speed~dist, data = dat)
lines(ksmooth(dat$dist, dat$speed, "normal", bandwidth=bw), col="(
```

Figure 2.3: ***Controlling bandwidth with a slider control.*** We control the bandwidth parameter of a kernel regression estimator relating distance and speed of traveling cars. By embedding an interactive control within a dynamic document, we allow the viewer to explore the code and the effect of the specific parameters we chose (bandwidth).

One natural way of using these controls is to embed them within dynamic documents. This extends the concept of exploring the research process further, giving audiences another tool to gain a deeper understanding of the reported results by interactively controlling and visualizing the effects of individual code elements. To support these controls we require two things. First, a document author must be able to declare these controls and associate them with specific code in his or her dynamic document. Secondly, we require a display platform capable of supporting these controls, including both rendering them and re-running analysis code in response to their use. We describe the concept of an *interactive code element* and a simple widget abstraction for use with our object model below. We leave discussion of rendering functional versions of these controls to Section 3.2.4 in the next chapter.

Interactive code elements – e.g., *IntRCodeElement* when the code is R – are an extension of standard code elements with one or more interactive controls associated with them. An *IntRCodeElement* object is simply an *RCodeElement* object with additional information about GUI control(s) that should be associated with the code and output for the element. *IntRCodeElement* objects default to standard *RCodeElement* behavior when the output format does not support interactivity.

We abstract common GUI controls (e.g., sliders, buttons, textbox inputs, etc) as *widgets* which we represent as S4 objects. The information necessary to describe these widgets is somewhat specific to the platform in which the widgets will be displayed, though much of this can be handled via different renderings of the same widget description (see Section 2.3.2). We allow arbitrary widget descriptions by providing an empty, virtual *WidgetBase* class. Any object which inherits from *WidgetBase* (or lists thereof in the form of *WidgetList* objects) can be attached to an *IntRCodeElement* in the ***widgets*** slot, specifying that these widgets should be rendered along with the code and most recent output for the element as a means for viewers to interactively modify the computations performed and update the corresponding output. The layout of these widgets during

Table 2.2: The *IWidget* class for declaring GUI controls

Slot	**Contains**
var	The name (*character*) of a variable the widget will control
linenum	The expression index (*integer*) within the code. The current expression will be replaced with an assignment of the variable to the user selected value[1]
default	A *character* value containing the default value the widget should bet set to (e.g., the value a slider is initially set to)
additional.info	A *list* containing any additional information the target platform might need

rendering is left to the display mechanism, though metadata on the widget or widget list objects can dictate layout information for display programs which support this.

Beyond the base virtual class, we provide an *IWidget* class, and subclasses, which implements a particular, narrow but useful way of describing how widgets should interact with rendered code. We have designed the *IWidget* class to contain the information necessary to construct a GUI control which modifies an assignment expression within the code and then re-invokes the code to generate new output. The *IWidget* classes have four base slots, shown in Table 2.2.

We discuss supporting the *IWidget* style of widget descriptions in a display/rendering platform in Section 3.2.4. Because we discuss the mechanism, and the benefits and downsides of this particular style of widget description there, we omit a similar discussion here.

We now turn to manipulating *DynDoc* objects and their elements once created. We first discuss low-level programmatic operations on the objects in Section 2.2. We then discuss the higher-level weaving API for generating dynamic reports and other types of output files from *DynDoc* objects in Section 2.3.

2.2 Operating on *DynDoc* Objects

We have designed **DynDocModel** so that users can operate on *DynDoc* objects beyond simply generating an output report. In this section we discuss a number of high-level and low-level operations in the context of our digit analysis. In particular, we illustrate the high-level mechanisms for visualizing the document structure as a graph, and for identifying threads. We also discuss the lower-level API for querying a document for specific elements, or sets of elements, that meet certain criteria. We leave the traditional weaving of reports to the following section.

2.2.1 Visualizing document structure

Though we represent dynamic documents as nested collections of elements, it is useful conceptually to consider the graph structure Gentleman and Temple Lang alluded to in the quote we cited at the beginning of Section 2.1.1. This gives us a natural visualization of document structure and, by proxy, the research process.

We can use a graph representation of a document to quickly get a general sense of its structure. For example, the graph allows users to see the layout of content in the document and identify the possible threads within it. This includes the ability to easily identify *terminal alternatives*, which were considered by the analyst but do not fit into the larger flow of the analysis – e.g., because their code throws an error, does not return the expected type of object, or is otherwise a dead-end in the analysis. The graph also shows the effect of declaring detail levels, which define implicit edges in the document graph which skip the affected content. Figure 2.4 represents the structure of our digit analysis document as a graph. We have excluded the individual code and text elements for clarity. The graph, drawn via our `makeStructureGraph()` function, is read top to bottom with nodes representing structural elements (task, decision, or alternative) in the document. We discuss how these element types map specifically to the graph structure and appearance below.

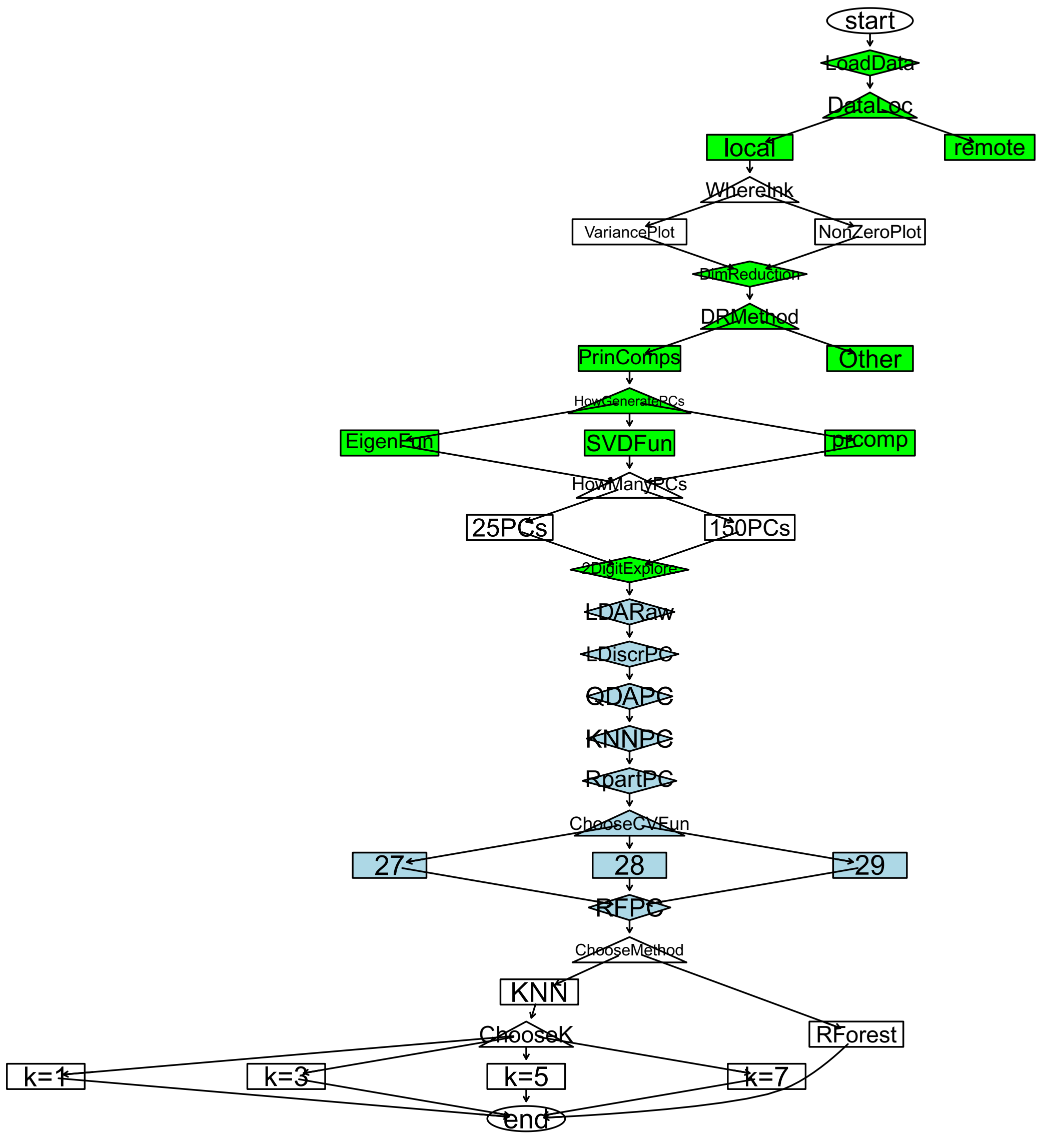


Figure 2.4: ***Visualizing the digit analysis process as a graph.*** We visualize document structure as a directed graph. Decisions, alternatives, and the starting point of tasks are pictured as triangles, squares, and diamonds, respectively. Tasks membership is indicated via color, with each colored non-task node belonging to the most recent task sharing the same color. Circles represent the start and end of the document. Nodes are labeled by their id, or by position in the list of nodes if they have no id (see nodes '27', '28', and '29' within the chooseCVFun decision).

The graph in Figure 2.4 represents decisions as triangles, and the beginnings of alternatives and tasks as squares and diamonds, respectively. For nodes which represent or are contained in a task, color indicates the nesting depth of that task. Task membership, then, is indicated by blocks of color. A non-task element will always form a continuous color block with its closest task-element ancestor (or it will be uncolored if there is no such task). Task elements – which are colored according to their own nesting depth – are contained in the task whose color forms a contiguous block up to the task. In this figure, elements with a white background are outside of the explicitly declared tasks within the document, while those with green backgrounds are contained most directly by a top level task, and those light blue ones have sub-tasks as their closest task-element ancestor. We see this in the nodes labeled `LoadData`, `DataLoc`, `local`, and `remote`. These nodes form a contiguous block of green, indicating that these are contained by the `LoadData` task. Similarly, the `DimReduction` task consists of 8 nodes representing the analysts code, text, and decision between methods during the dimension reduction phase of his analysis.

We can see that when reading the data in the `LoadData` task, the analyst made a decision between two alternatives, but only one connects to the rest of the graph. The `remote` alternative contains code to read the data from a remote server which no longer provides the files for download. This is what we call a *terminal alternative*, in that it cannot be used within a thread that continues through the rest of the document (because the code will fail to load the data used throughout the analysis). The `local` alternative, on the other hand, uses a local copy of the data. Our graph makes the presence of this terminal node very obvious without needing to inspect or try to run the code. Furthermore, we know based on the surrounding structure that were the `remote` alternative to be viable, it would replace `local` and feed directly into the same tasks and code that the `local` alternative does. The other terminal alternative in our graph (the `Other` alternative within the `DRMethod` decision), is a note the analyst made to himself that, if analyzing the principal component data does not perform well, he might investigate different dimension reduction methods.

The graph for our document also highlights nesting and the relationship among decisions within the document. Near the bottom of the graph we can see that the decision among k values for a final KNN classifier is contained in the alternative which selects KNN as the final method. If the analyst chooses random forests instead, the choice of k is never visited.

We provide two functions which create graph visualizations for dynamic documents. The first, `makeStructureGraph()`, creates a high-level visualization of the structure of a document, while the second, `makeDocumentGraph()`, displays a graph containing the code and text elements as well as the structure. `makeDocumentGraph()` can be inappropriate for visualizing full, lengthy documents – as we can see from the complexity in Figure 2.4, which omits individual code and text elements – but it can be useful to get more detail by graphing only portions of a document.

2.2.2 Subsetting and querying the document

Computing on a document often requires us to identify a specific element or set of elements within the document before processing them further in some way. For example, researchers looking to validate the digit analyst's work might analyze the document to determine what R packages they might need. This requires two steps: identifying all (R) code elements in the document, and determining the packages loaded by each of them.

Code elements can appear anywhere in the nested hierarchy of the document. We provide `[[()`, and `[()` methods for *DynDoc* and *DocElement* objects for extracting elements from documents using standard R syntax, but the nested hierarchy of document elements makes using these traditional subsetting mechanisms to collect the code elements cumbersome.

Our **rpath** package (Becker, 2013) allows users to select from R objects via a subset of the

XPath (Clark and DeRose, 2006) syntax designed for querying specific sets of nodes from within XML documents. Because XPath is specifically designed to work within nested structures, we can define a path that will match all code elements (or all elements meeting some other criteria) within the document. We provide the `dyndoc_rpath()` convenience function, which allows **rpath**'s machinery to understand how to traverse and extract attributes of *DocElement* objects.

Without getting into the details of how XPath works, the path `"//code"` would match all `<code>` elements within an XML document. To retrieve a list of all the *RCodeElement* objects within a document `doc`, then, we simply call `dyndoc_rpath()`:

```
codeElements = dyndoc_rpath(doc, path = "//rcode")
```

We can go further, however, and retrieve the code from these elements directly by accessing their `content` field (as an **rpath** attribute) via the path:

```
code = dyndoc_rpath(doc, path = "//rcode/@content")
```

With the code extracted from the document, all that remains is identifying what libraries are used. We do this via the **CodeDepends** package (Temple Lang et al., 2013):

```
library(CodeDepends)
script = readScript("", txt = unlist(code))
pkgs = unique(unlist(sapply(script, function(x)
                                getInputs(x)@libraries
)))
```

This generates a vector of all packages loaded by code in any R code element anywhere in our document, as desired. We can also use XPath-style indexing when specifying threads, as we will see in the next section.

We designed **rpath** so that nearly every part of its machinery is customizable, which allows us to use the package with a wide variety of R objects with radically different structures. These customizations include how the `rpath()` function retrieves the children of an object, matches children to entity names in the path (e.g., `"rcode"` in our example above), and retrieves the named list of attributes available for use in a path (e.g., `"@content"` above). Among other arguments,[2] `rpath()` accepts *names_fun* and *attr_fun*, which allow the caller to specify R functions used to resolve names and attributes to match against as each portion of a path is resolved, respectively. Thus one could use the classes of R objects being traversed as names in the path in one call, or use `names()` and expose the class as an XPath attribute in another, as we see in the simple example below. In the example, we create multiple *lm* objects by regressing each variable against miles per gallon (`mpg`). We then use `rpath()` to extract aspects of all the fits simultaneously by class and name.

```
vars = setdiff(names(mtcars), "mpg")
fits = lapply(vars, function(x)
                       lm(as.formula(paste("mpg~", x)),
                          data = mtcars))

modeldfs = rpath(fits, "//data.frame",
                 names_fun = getClassesVec)
resids = rpath(fits, "//residuals", names_fun = names)
```

Our first `rpath()` call in the code above searches the list of *lm* fits for elements which are *data.frame* objects. In this case, it finds the 'model' *data.frame* for each fit and returns them in a single list. Our second *rpath* call retrieves the vector of residuals for each fit – the named `"residuals"` element within each *lm* object – and returns them in a list. In our simple example

[2]We refer our readers to the **rpath** documentation for a more complete discussion of that package.

Table 2.3: Default class to rpath node name mapping for `dyndoc_rpath()`

Class	**rpath node name**
CodeElement	code
RCodeElement	rcode
TextElement	text
MDTextElement	markdown
DbTextElement	docbook
LatexTextElement	latex
DecisionElement	decision
AltElement	alt
AltImplElement	alt
AltMethodElement	alt
TaskElement	task
DocThread	thread
DynDoc	document
(other)	(full name of class)

*The **rpath** node names which match our dynamic document-related object classes. To match objects of the classes in the left column, we use (by default) the abbreviated names on the right in our XPath expression.*

we could easily achieve the same result with `lapply()` statements, but that quickly becomes cumbersome when querying more deeply nested objects – such as non-linear dynamic documents.

We will use `rpath()` throughout this chapter via the `dyndoc_rpath()` convenience function. `dyndoc_rpath()` accepts the same customization options as the base `rpath()` function, but provides alternative defaults designed for **DynDocModel** objects. Node names in **rpath** expressions – i.e., the names used to identify patterns to match – correspond to abbreviations (listed in Table 2.3) of the object's class if the parent is a **DynDocModel**-based object (*DynDoc*, *DocThread*, *DocElement*, *ThreadList*, *ElementList*, etc). Names – in the R `names()` sense – are used for named *list* and *character* objects, while numeric indices are used for unnamed objects of these types. Attributes – generated via the `dyndoc_attrs()` function – include entries for an element or instance's metadata, content, and any associated output.

The **rpath** package also implements the concept of *predicates* from the XPath specification. Predicates, which are surrounded by [] in XPath and **rpath**, add extra conditions which nodes must meet in order to match the current step in the path. For example, the path `"//decision[@id`

`=='ChooseMethod']"`, when used on the comprehensive digit document, will match only the decision element representing the choice of final classifier method between KNN and Random Forests. We use predicates again in the next section when we present code to generate and extract the error rates for all possible classifiers.

2.2.3 Specifying threads

The ability to generate static, linear reports is essential, even when dealing with comprehensive documents. We do this by processing specific threads – as defined in Section 2.1.2 – through the document. We discuss the processing itself in Section 2.3, but first we must be able to identify and extract threads.

To characterize a thread we need four pieces of information:

1. a starting point for the thread (A),
2. an ending point for the thread (B),
3. alternatives to select for each decision along the path – through the document graph – between A and B,
4. elements along the path which should be excluded from the thread.

With this information we traverse backwards from element B to element A, collecting a linear subset (thread) of the document. Traversing backwards from B instead of forwards from A drastically reduces our search space of possible paths by excluding paths which originate at A but never reach B; the number of such paths can be quite large if B is nested within a particular alternative rather than occurring at the top level of the document. With this thread in hand we can use the methods discussed in the next section to generate the standard rendered linear narratives (e.g., articles, vignettes, and dynamically generated reports more generally).

Users interacting with *DynDoc* objects directly use the `getThread()` function to create *DocThread* objects representing specific threads. Starting and ending points can be specified as numeric positions – interpreted as location in the list of the document's top-level children – or by passing the elements directly. Specifying elements directly allows users to specify threads with starting and ending points anywhere in the document. Similarly, the lists of alternatives to select and elements to exclude can be identified directly (as *list*s of elements), or as the set of elements matching an **rpath** expression (as implemented by `dyndoc_rpath()`).[3] Finally, convenience arguments *threadid* and *detail_level* allow the user to specify alternatives to select via threadid and elements to exclude from the thread by maximum detail level, respectively.

DynDocModel also provides the `getAllThreads()` function, which returns a list of all threads which match a specified criteria. We can use this, along with `dyndoc_rpath()` and the `evalDynDoc()` function, which we discuss in the next section, to collect the data for our plot of all possible misclassification rates from Section 1.2.5 (Figure 1.12):

```
allThreads = getAllThreads(doc, detail_level=1)
allThreads = lapply(allThreads, evalDynDoc)
errs = dyndoc_rpath(allThreads,
    "/thread//rcode[@id=='finalErr']/@outputs[1]/@value")
```

The above code first extracts all (valid) threads through `doc` (our digit analysis) and evaluates them (by evaluating their code and capturing the output). We specify a detail level of `1` so that the analyst's two population explorations will not be run because it has no effect on misclassification rate of the final classifier for all ten digits. We then use `dyndoc_rpath()` to extract the final error rates for each thread by identifying all (R) code elements with id `"finalErr"` (one per thread) via the first portion of the path, `/thread//rcode[@id=='finalErr']`, and retreiving the

[3] `getThread()` selects the first non-terminal alternative when none of the instructions match any of the alternatives within a given decision.

generated output value via the `/@outputs[1]/@value` portion. We discuss the details of this in Section 2.3.1.

2.3 DynDocModel's Computational Model

In the previous section we discussed **DynDocModel**'s object model for representing dynamic documents in R. Here, we describe our model for processing such documents. We define our model in general terms, with specific forms of processing, such as the common dynamic report generation (*weaving*) and code extraction (*tangling*) operations, interpreted as special cases, rather than wholly distinct mechanisms. We also show how this general abstraction allows extensions of current processing paradigms.

Our model divides the processing of dynamic documents into four steps:

1. Representing the document as a *DynDoc* object
2. Projecting the document into the subset to be processed
3. Processing the projected document and capturing generated content
4. Rendering static and dynamically-generated content

We briefly consider a concrete example to illustrate these concepts. In the context of dynamic documents, *weaving* is the dynamic creation of documents (typically HTML or PDF) containing the code and text of a dynamic document interspersed with output generated by running the code. Suppose we want to weave an article-like result presenting the housing analyst's findings when a particular strategy is used. The first step is self explanatory: we require a *DynDoc* object representing the analyst's comprehensive dynamic document.

Next is projection. When creating a report from a comprehensive document, *projection* generates the thread that represents the specific strategy we wish the report to reflect. We can think of this as analogous to projecting a high dimensional dataset – our representation of the analyst's full research process – into a lower dimensional space – the 'space' where this particular strategy was chosen and used. In particular, this typically involves generating a *DocThread* object which contains instances of each element which will be rendered into the report and excludes those which will not.

In the case of weaving, *processing* corresponds to simply evaluating the code in the projected document (a thread in this case) and capturing the output so that it can be inserted into the generated report. In our model, the captured output is associated with the individual element instances in the thread, as described in Section 2.1.2.

Finally, the collection of code, text, and generated output is *rendered* into the final woven report. Because our model is format agnostic, this step involves both transforming the objects generated by evaluating code into markup and any necessary conversion between text formats – e.g., Markdown to HTML.

We have implemented our model – pictured in Figure 2.5 – with two core design principles in mind. First, each step is fully customizable by the end user. We give examples of such customizations throughout the remainder of this section. Secondly, our implementation is highly modular, such that users can call each step separately, and access all intermediate results – e.g., the *DocThread* after processing – for their own use.

The most common use case for dynamic documents is likely to remain the generation of woven reports or other output files. **DynDocModel** implements a four-part computational model for generating output files from processed *DynDoc* or *DocThread* objects, pictured in Figure 2.6. This represents the *Formatting and Rendering* component of the overall model pictured in Figure 2.5. We discuss the details of each of the sub-steps pictured here within remainder of this chapter.

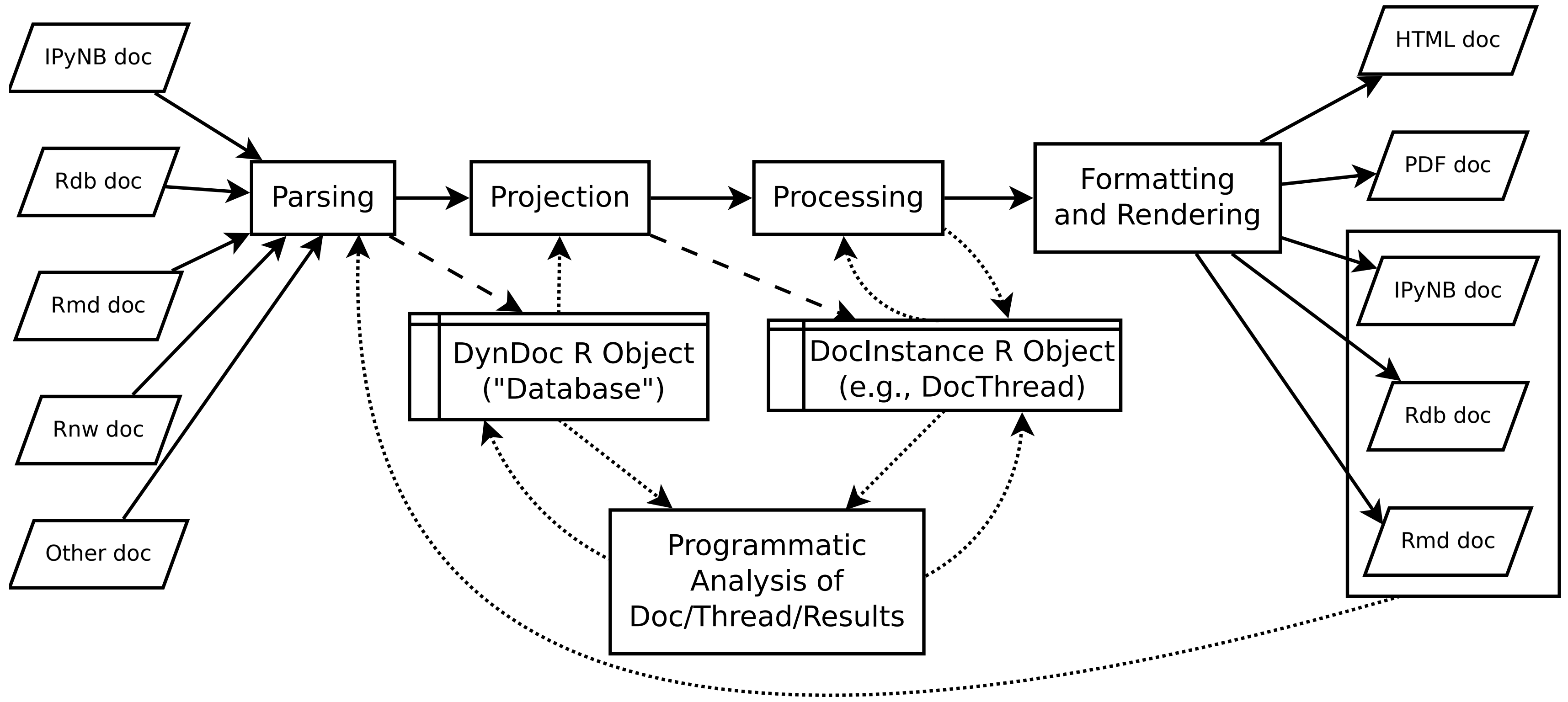


Figure 2.5: ***The DynDocModel computational model for processing dynamic documents.*** Solid lines represent the standard input-document-to-output-document paradigm under our model, which encompasses four steps: *parsing*, *projection*, *processing*, and *formatting and rendering*. Dashed lines indicate the creation of our two primary object representations, *DynDoc* objects representing entire documents, and *DocInstance* objects representing the projected (sub)documents we typically process. Dotted lines represent programmatic actions available to the user via our API when operating on these R objects directly.

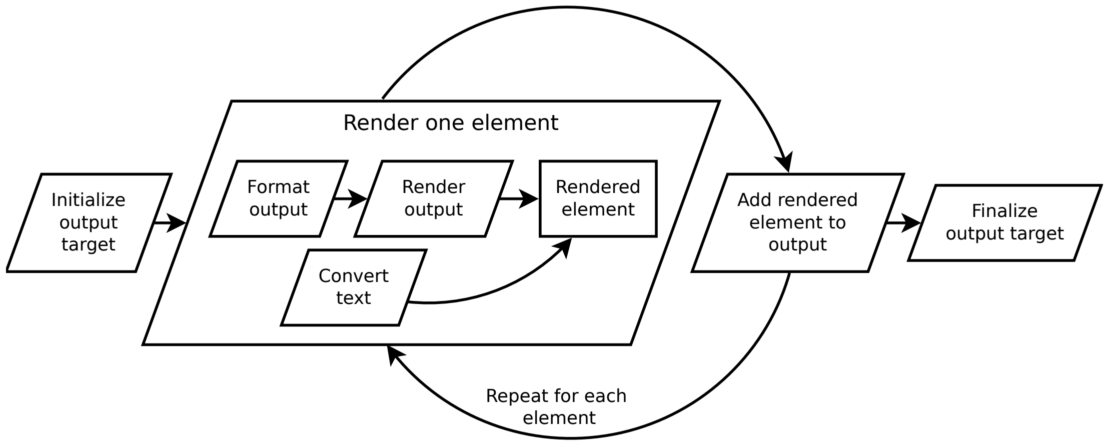


Figure 2.6: ***The DynDocModel model for generating output files.*** We operate on each element separately when constructing output markup. For each element we generate markup for any output via a two step process. We first *format* the output by transforming the output (R) object into the R object to be rendered. We then *render* the formatted object into the markup which will appear in the output document. We also convert any textual content into the target markup language. Finally, we combine the text and output markup to create the rendered output for the element as a whole, which is added to the output document.

As with other portions of our API, we focused heavily on customizability when designing our mechanism for generating rendered output. Four high-level steps make up our model for generating output files:

1. initialize an output target
2. generate markup representing a single element – including any associated output
3. add that markup to the output target
4. finalize the output target (e.g., writing to file or closing a *connection*)

By formulating our computational model with separate steps in which we initialize, add to, and finalize the output target, we have made the rendering mechanism agnostic to how content is added to the final document. As such, our model supports both streaming style targets based on R's *connection* objects and targets which collect markup in R before writing to disk (or not), such as *character* vectors or the **XML** package's *XMLInternalDocument* objects (Temple Lang, 2013).

This allows users to re-use logic for rendering markup, regardless of how the destination for that markup is represented in R.

We listed generating markup from individual *DocElement* and *ElementInstance* objects as a single top-level step above, but we actually break the markup generation into three separate sub-steps. These sub-steps – which are also illustrated in Figure 2.6 – are formatting output, converting text, and rendering. We define these sub-steps below.

In the *formatting output* step we transform any R objects representing output that are with the element instance into the form which will be rendered into markup. Examples of this formatting could include printing the object and capturing the console output, or transforming a *lm* object into a *data.frame* so that we can use existing machinery, such as that found in *ReportingTools* or **hwriter**, to represent it as a table in the generated report. We note that the *format* transformation is *not* specific to the type of report (HTML, PDF, etc) being generated. Any transformations specific to the output format – for example generating hyperlinks from text – occur in the *rendering* step which we will discuss shortly. This distinction allows the formatting transformation to be reused across different markup targets. Our two-stage transformation model for output is based heavily on the model we developed when designing version 2.0 of the API for the *ReportingTools* package (Huntley et al., 2013).

Next is the *conversion* step. This step transforms textual content into the correct markup language as necessary. By default, this conversion is performed either by MacFarlane's Pandoc (MacFarlane, 2006) or Nolan and Temple Lang's Omegahat XSL files packaged in the **XDynDoc** package (Nolan and Temple Lang, 2013), depending on source format.

Finally, the *rendering* step generates the final markup that will be added to the output target. For instances of code elements with output attached, this includes both transforming the formatted output into markup and combining it with a markup representation of the code. For text elements and instances this typically uses the converted text unchanged. Finally, for structural elements, the rendering process involves rendering all children and combining the resulting markup.

We implement **DynDocModel** such that each of these steps is fully customizable by the user. We now go through a number of simple examples designed to illustrate our model, and the customization available at each step and sub-step, in action. These examples, while based primarily on our example analyses from Chapter 1, are simplified from what one might do in practice for the purposes of conciseness and clarity.

2.3.1 Customizing the processing step

We now consider customization of the processing step in the context of weaving a dynamic report. Recall that for standard weaving, processing refers to the evaluation of code within the document and the capture of the output generated and values returned by that evaluation.

One of the central benefits of the dynamic document approach is the fact that output in woven reports is automatically generated at build time by evaluating the exact code in the document. **DynDocModel** evaluates elements (and collections of elements, e.g., threads) via `evalDynDoc()`. For code elements, the code is parsed and executed and any captured output, along with the final returned object generated by the code, are attached to the element or instance being evaluated. For threads and structural elements, `evalDynDoc()` recursively evaluates each of the object's children. In order to generate and capture the output that we need to create a standard woven report, then, we simply call `evalDynDoc()` on the thread we are weaving. Output is added to each *ElementInstance* object in the thread within the instances' ***outputs*** fields as an *OutputList* object. The code below extracts the default thread from `doc` and then evaluates it, replacing the unevaluated *DocThread* with one containing output for each *RCodeElement* the thread visits.

```
thr = getThread(doc)
thr = evalDynDoc(thr)
```

The `evalDynDoc()` function also lets us customize the evaluation step. We can use this to add post-processing to the evaluated results, use an alternative evaluation engine, e.g., Wickham's **evaluate** (Wickham, 2014) package, or capture information about the evaluation itself, such as software versioning information, timings, date executed, etc, and add it to the output. The default evaluator uses our **RCacheSuite** package to perform caching-enabled evaluation, but custom evaluators could utilize other caching mechanisms, or turn off caching altogether.

For example, suppose we want to weave an HTML report describing the digit analyst's findings which includes timings for evaluating each code element in addition to displaying any output generated by the code itself. By customizing the evaluation step we can add timings without changing the document.

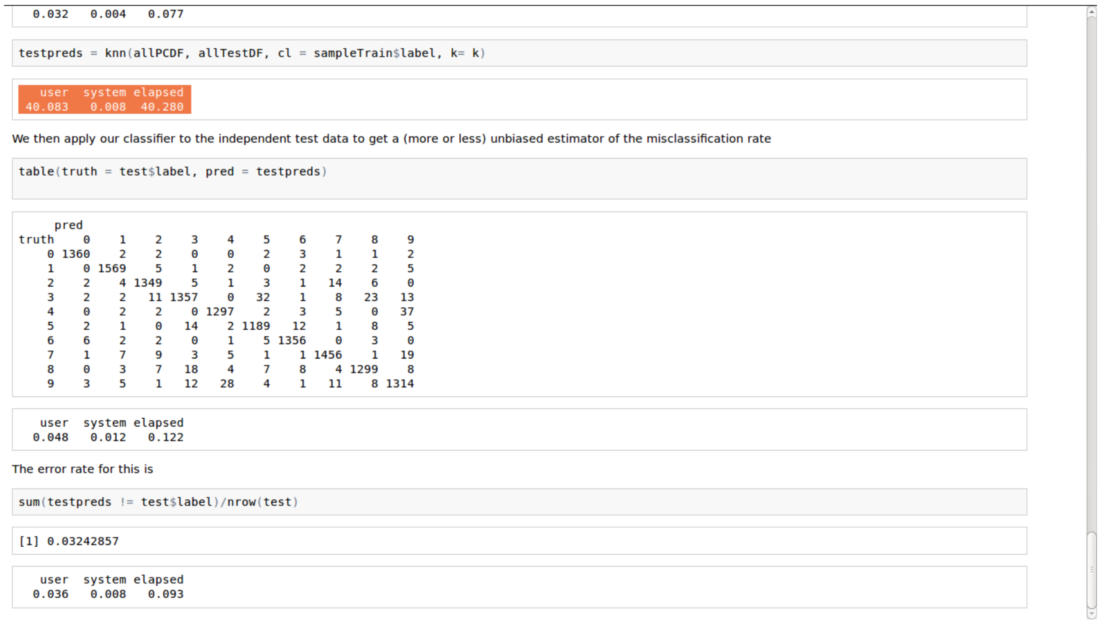
We customize evaluation via `evalDynDoc()`'s *eval_fun* parameter. The *eval_fun* parameter accepts an arbitrary R function, so long as it has named *code* and *env* arguments and accepts ... to handle any additional parameters. To generate a woven report with timings we write a custom evaluation function which calculates execution time and adds it to the generated output:

```
eval_timings = function(code, env, ...) {
  timing = system.time({
                            ret = dyndoc_evaluate(code,
                                                  envir = env)
                          })
  as(c(ret, list(timing)), "OutputList")
}
```

Our `eval_timings()` function is a lightweight wrapper around **DynDocModel**'s default evaluator, `dyndoc_evaluate()`, that adds timings to the output. The `dyndoc_evaluate()` function, in turn, is a wrapper around Wickham's `evaluate()` function – from the package of the same name (Wickham, 2014) – which captures errors, warnings, text printed to console, and the final returned value when evaluating code.

We add the timings by wrapping the `dyndoc_evaluate()` call within a call to `system.time()` and add the timing result to the end of the list of outputs. We return an *OutputList* object, which is a simple *list* re-classed via S4. This ensures that **DynDocModel** can differentiate between a single output which is a list, and a list of multiple outputs, e.g., a *recordedPlot* object containing a drawn plot and the final returned object.

Once rendered, the resulting HTML file will contain all components of the standard woven report. Following each code element (and any output for that element) there are timings for how long it took to run that code (Figure 2.7).



```
  0.032   0.004   0.077
```

```
testpreds = knn(allPCDF, allTestDF, cl = sampleTrain$label, k= k)
```

```
   user  system elapsed
 40.083   0.008  40.280
```

We then apply our classifier to the independent test data to get a (more or less) unbiased estimator of the misclassification rate

```
table(truth = test$label, pred = testpreds)
```

```
     pred
truth    0    1    2    3    4    5    6    7    8    9
    0 1360    2    2    0    0    2    3    1    1    2
    1    0 1569    5    1    2    0    2    2    2    5
    2    2    4 1349    5    1    3    1   14    6    0
    3    2    2   11 1357    0   32    1    8   23   13
    4    0    2    2    0 1297    2    3    5    0   37
    5    2    1    0   14    2 1189   12    1    8    5
    6    6    2    2    0    1    5 1356    0    3    0
    7    1    7    9    3    5    1    1 1456    1   19
    8    0    3    7   18    4    7    8    4 1299    8
    9    3    5    1   12   28    4    1   11    8 1314
```

```
   user  system elapsed
  0.048   0.012   0.122
```

The error rate for this is

```
sum(testpreds != test$label)/nrow(test)
```

```
[1] 0.03242857
```

```
   user  system elapsed
  0.036   0.008   0.093
```

Figure 2.7: ***Adding automatic timings to a woven report.*** By passing a custom function to *eval_fun* we capture timings for each code element – in addition to the evaluation output – and add them to the output associated with that element, causing the timings to be automatically added to the woven report document. The timing information for our call to `knn()` to fit the final model is highlighted in this screenshot. Furthermore, using the custom formatting and rendering machinery we talk about in the next section we could have the timings appear differently in the report than the standard output.

Thus we can use customized evaluation to expose additional or different output in woven documents. We can go farther than adding timings. For example, we can leverage Ram and Temple Lang's **rProv** package (Ram and Temple Lang, 2012) to generate provenance information and insert it directly into woven reports in an invisible but extractable form. We now turn our attention to the process of generating the actual woven report from an evaluated thread.

2.3.2 Customizing markup generation

We now consider generating an HTML report for our housing analysis by weaving an evaluated thread into Markdown (Gruber, 2004) and then using Allaire et al's **markdown** package (Allaire et al., 2014) to transform that Markdown into HTML. By default, this is done by: initializing the output document as an empty *character* vector, rendering each element and piece of associated output into Markdown, adding to the output by pasting new content into the vector, and finalizing the output document by calling `markdownToHTML()` to transform the Markdown into HTML and write it to a file. The `markdownToHTML()` function takes care of including the CSS files necessary to format code and output areas differently than text.

Suppose, however, we want to build an HTML page that compares two evaluated threads side-by-side. We can do this by calling `writeDynDoc()` on each thread using a custom finishing method so that it generates Markdown snippets that can be combined into a single page. **DynDocModel** provides this functionality via the `compareThreads()` function, but we choose to discuss a simplified version of it here to illustrate the concept of customizing file finalization behavior. Our `compare_finisher()` function, shown below, wraps each full thread in a `<div>` node whose style is compatible with side-by-side display. We can then simply combine these two `<div>`s into a single piece of Markdown[4] and call `markdownToHTML()` to generate our desired page.

[4]HTML code is valid Markdown syntax, so we will simply treat the `<div>`s as Markdown directly

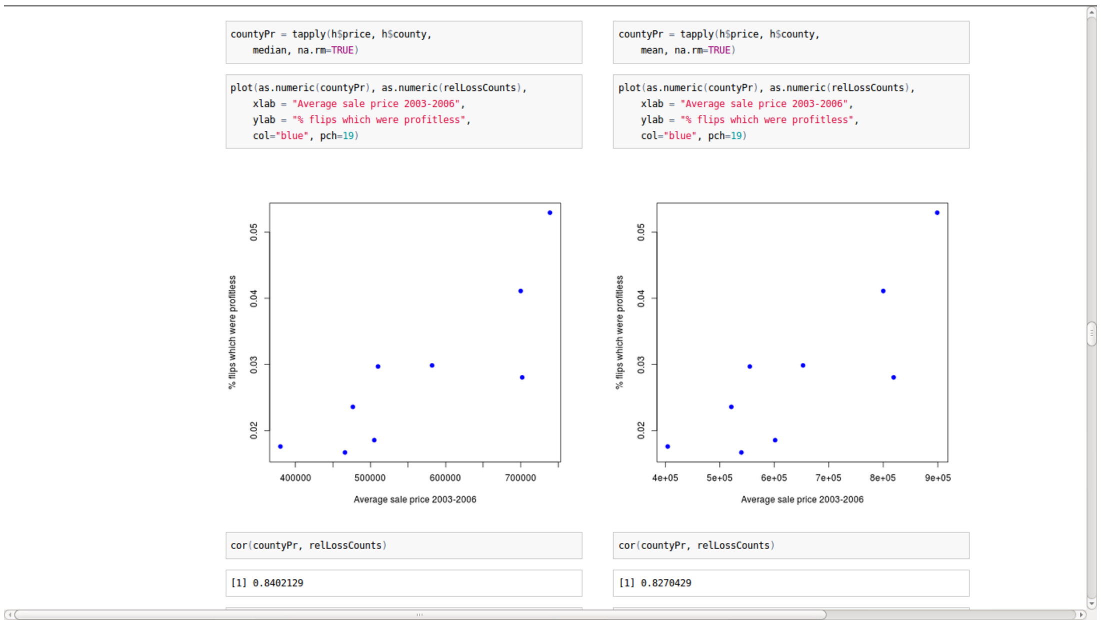
```
compare_finisher = function(out, file, doc, ...)
{
    paste0("<div style='width:50%;float:left'>\n",
        out, "\n</div>")
}
```

When we specify our `compare_finisher()` function as a custom finisher in a call to `writeDynDoc()`, it is passed the collected Markdown snippets representing all of the code, text, and formatted output that make up the contents of the woven report. The `writeDynDoc()` function returns the object returned by the finisher so, in our case, we will get a *character* vector containing that content wrapped in a `<div>` node styled so that it can be displayed side-by-side with other content.

We can now write a function which will accept two threads, generate a `<div>` for each of them, and then combine them into a single page. The `sidebyside()` function simply calls `writeDynDoc()` on each of two threads with our custom finisher, and pastes the resulting markup into two side-by-side `<div>`s which appear in a single HTML page.

```
sidebyside = function(thr1, thr2)
{
  content = paste("<div id='topcontainer'>",
                  writeDynDoc(thr1, format="md",
                    output.finisher=compare_finisher),
                  writeDynDoc(thr2, format="md",
                    output.finisher=compare_finisher),
                  "</div>", collapse="\n"
                 )
  markdownToHTML(content)
```

Figure 2.8 shows the page generated when we call this function with mean and median threads from the housing analysis.


```
countyPr = tapply(h$price, h$county,
    median, na.rm=TRUE)
```

```
plot(as.numeric(countyPr), as.numeric(relLossCounts),
    xlab = "Average sale price 2003-2006",
    ylab = "% flips which were profitless",
    col="blue", pch=19)
```

```
cor(countyPr, relLossCounts)
```

```
[1] 0.8402129
```

```
countyPr = tapply(h$price, h$county,
    mean, na.rm=TRUE)
```

```
plot(as.numeric(countyPr), as.numeric(relLossCounts),
    xlab = "Average sale price 2003-2006",
    ylab = "% flips which were profitless",
    col="blue", pch=19)
```

```
cor(countyPr, relLossCounts)
```

```
[1] 0.8270429
```

% flips which were profitless
0.02
0.03
0.04
0.05
400000
500000
600000
700000
Average sale price 2003-2006
% flips which were profitless
0.02
0.03
0.04
0.05
4e+05
5e+05
6e+05
7e+05
8e+05
9e+05
Average sale price 2003-2006

Figure 2.8: ***Comparing two woven threads side-by-side.*** We display a scatterplot of typical house price vs percent house resales which resulted in a loss between 2003 and 2006 in the San Francisco Bay area, by county. We can see that using mean vs median to calculate typical house price does not qualitatively affect the relationship between county expensiveness and the rate of resales that were for a loss.

Preparing content to be inserted into the output file happens in up to three steps. First, if necessary, the contents of the element or instance are formatted into the output format. For example, if we are generating an HTML page from an Rnw file, the text – which is in the LaTeX markup language – must be converted into either HTML or Markdown. Next the converted content of the element is wrapped in any necessary markup so that it will be displayed properly. For example, in Markdown, blocks of code are delimited by placing them between two lines containing three tick-marks (`'''`). Finally, any output objects associated with the element or instance are formatted into the target markup language and appended to the representation of the element. This combination of the converted and rendered element content and the formatted output is then passed to the custom function we specified via the *add.rendered* parameter to place it in the output document.

We can customize how elements and output appear in the woven report via the *element.renderers*, *formatters*, and *converters* arguments. *Formatters* modify output objects into the form that will be rendered into markup. This can include adding or removing rows or columns from a table, decorating columns or list elements with classes that will control how they will be rendered, etc. *Element renderer* functions perform the transformation of formatted output objects into final markup that represents the output within the resulting document. For example, we might add a "Link" S3 class to a column within an output *data.frame* during the formatting step; we would then use that column's contents to actually construct the links themselves during the rendering step. Finally, *converter* functions transform textual content from its input markup format – as indicated by the *DocElement* subclass, e.g., *LatexTextElement* – into the markup format of the output document, e.g, HTML.

Renderer, converter, and formatter functions are selected via a custom dispatch system built on top of Chambers' S4 multiple dispatch system (Chambers, 2010) in R. We illustrate this mechanism below in the context of formatters, noting that renderers and converters are dispatched in the same way.

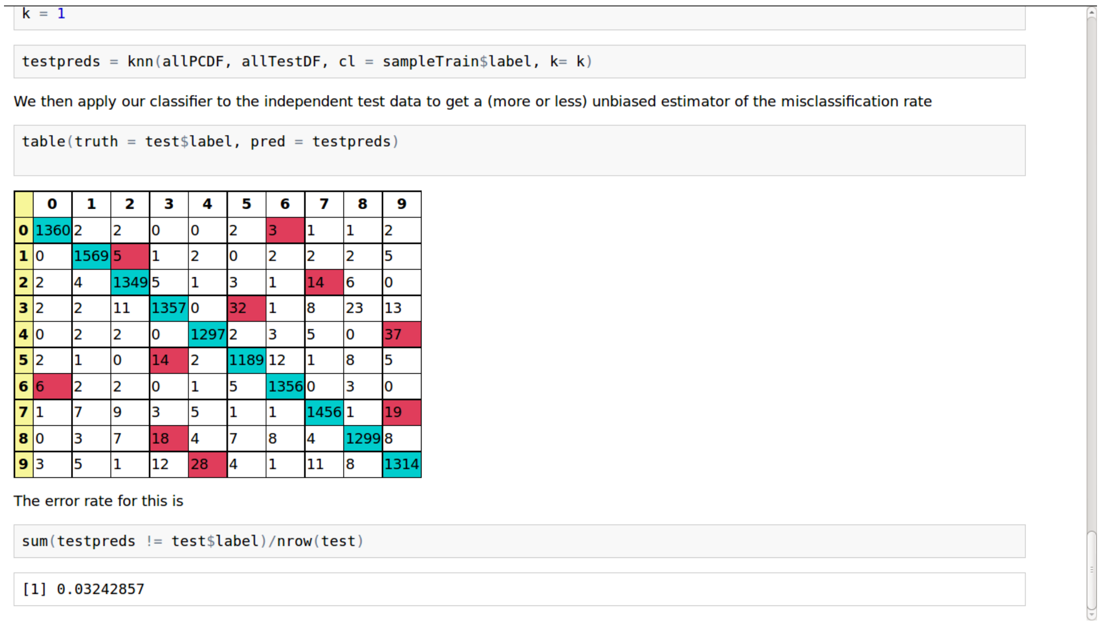
The formatting of output in **DynDocModel** is both object class- and target format-specific. We can have different logic for representing, e.g., *data.frame* objects in Markdown, *lm* objects in Markdown, and *data.frame* objects in LaTeX. By default, formatting is handled by S4 methods specific to the object class and output target provided by **DynDocModel**. We override these via the *formatters* argument to `writeDynDoc()`, or by associating formatter(s) directly with individual elements or instances thereof. In each case, we can specify a function or named list of functions. When the user specifies a single formatter function, that function is called for all relevant output objects regardless of class. When the user specifies a named list of functions, this acts as an informal methods table – with the names indicating class – against which we perform dispatch. As with S4 dispatch, if no method is found for the object's direct class, methods for its superclasses are considered in decreasing order of inheritance distance.[5] Finally, if superclass methods also don't match, the default method – i.e., the method for the *ALL* class – is called. If no default formatter is specified, the S4 generic is used, which allows users to temporarily override the formatting behavior for some classes while retaining the default behavior for others.

Suppose we want to generate an HTML report which displays the confusion matrix – a *table* object – in the digit analysis as a styled HTML table, rather than as verbatim text, with all other output displayed as usual. We do this by passing a named list of custom formatters which take precedence over the class-specific default. In our case, we can generate the report pictured in Figure 2.9 by assigning a custom *table* formatter to the element instance – identified via **rpath** – whose output contains the confusion matrix:

```
confMat = dyndoc_rpath(thread,
                        "/rcode[@id=='ConfusionMatrix']"
                       )[[1]]
confMat$formatters = list(table = table_HTML)
writeDynDoc(thread, "customTable.html",
              finish.output = table_finisher))
```

Passing a list with a named `"table"` element as the *formatters* argument specifies that whenever a *table* object is being formatted during this `writeDynDoc()` call, our custom `table_HTML()` function should be used instead of the default (which would display the output as verbatim text). The `table_HTML()` function, shown below, transforms an R *table* object into styled HTML tables and returns a *FormattedOutput* object containing the HTML markup.

[5]The first superclass method found with a particular inheritance distance will be used, regardless of how many other methods share the same distance.



```
k = 1
```

```
testpreds = knn(allPCDF, allTestDF, cl = sampleTrain$label, k= k)
```

We then apply our classifier to the independent test data to get a (more or less) unbiased estimator of the misclassification rate

```
table(truth = test$label, pred = testpreds)
```

	0	1	2	3	4	5	6	7	8	9
0	1360	2	2	0	0	2	3	1	1	2
1	0	1569	5	1	2	0	2	2	2	5
2	2	4	1349	5	1	3	1	14	6	0
3	2	2	11	1357	0	32	1	8	23	13
4	0	2	2	0	1297	2	3	5	0	37
5	2	1	0	14	2	1189	12	1	8	5
6	6	2	2	0	1	5	1356	0	3	0
7	1	7	9	3	5	1	1	1456	1	19
8	0	3	7	18	4	7	8	4	1299	8
9	3	5	1	12	28	4	1	11	8	1314

The error rate for this is

```
sum(testpreds != test$label)/nrow(test)
```

```
[1] 0.03242857
```

Figure 2.9: ***Customizing display of output in woven reports.*** We use a custom formatter for the *table* class to locally modify how tables are rendered for the element which prints the confusion matrix to display the correct classifications in cyan cells and the most common error for each digit in red. Other output tables are rendered normally.

```
library(hwriter)
table_HTML = function(object, formatters, ...)
{
  mat = unclass(object) #hwrite doesn't understand tables
  classes = matrix(NA, nrow = nrow(mat), ncol=ncol(mat))
  err_mat = mat
  diag(err_mat) = 0
  max_errs = max.col(err_mat)
  classes[cbind(1:nrow(mat), max_errs)] = "mosterrors"
  diag(classes) = "correct"
  val = hwrite(mat, class = classes)
  new("FormattedOutput", value = val, format = "html")
}
```

In our `table_HTML()` function, we use Pau's **hwriter** package (Pau, 2010), along with some custom CSS, to create and style the HTML table within our woven report (Figure 2.9). Specifically we create an HTML table with Cascading Style Sheet (CSS) classes `"correct"` and `"mosterrors"` on the cells which represent the correct prediction and highest error rate for each row (true digit class), respectively. To use these style classes, we need the CSS defining them to be included in the final HTML page, which is why we also specified a custom finishing function when generating the report. We omit the definition of `table_finisher()` here, as it simply calls `markdownToHTML()` in a way which forces our CSS to be included in the resulting page. In situations where multiple custom formatters each require specialized CSS styles or JavaScript code, they are typically added via the finalizer and the user is responsible for ensuring it is all compatible.

Formatters can be arbitrary R functions with two restrictions: they must accept parameters named *object* and *formatters*, as well as *...*, and they must return either a *FormattedOutput* or *FormattedOutputList* object. The *FormattedOutput* class has three slots: ***value***, ***format***, and ***info***. The ***value*** slot contains the R object that will be rendered into markup and inserted into the output document. The ***format*** slot describes how the data should be interpreted by the renderer; this is analogous to a MIME type, though the default renderers do not use proper MIME types currently. The ***info*** slot contains a list of extra information being passed to the renderers. Users can pass specific instructions to custom renderers via this mechanism. The *FormattedOutputList* class is simply an S4-classed list that contains only *FormattedOutput* elements.

In our `table_HTML()` function we return a *FormattedOutput* object with a ***format*** of `"html"`, indicating that it is HTML markup that should be inserted directly into the page. When the output object is a plot (*recordedPlot* object), the *FormattedOutput* object will typically have a ***format*** of `"plot"` and the ***info*** slot will contain information about which function should be called to display the plot and which graphics device should be used to save it to an image file.

Renderers implement the last step of the markup generation process: transforming formatted output into markup and combining that with markup for any relevant code and text. They return the exact markup that will be added to the output document via the function specified by *add.rendered*. Dispatch for renderers uses the same *list* overriding S4 mechanism described above for formatters, with the exception that the classes in question are for the document elements/instances rather than for output objects. The element to be rendered is passed to the renderer as the *node* argument. In addition to *node*, renderer functions are guaranteed to receive four other named arguments: *renderers*, *formatters*, *state*, and *converters*. A renderer must also accept ..., which allows specific sets of renderers to accept additional arguments. With these arguments, the renderer is expected to perform five general steps:

1. Use *converters* to convert any text content in the element to markup in the output format (as necessary)
2. Add any additional markup necessary for formatting the result from (1)
3. Call `formatObject()` on any associated output objects
4. Transform any *FormattedObject*/*FormattedObjectList* objects from (3) into raw markup in the output format
5. Return the combined markup from (2) and (4)

We illustrate the use of custom renderers by discussing a set of renderer functions which will *tangle*, or extract only the code from, a thread or linear document. Tangling is distinct from weaving in that the code is not evaluated, but we can use the same machinery (`writeDynDoc()`) to tangle a script as we do to emit a woven report by using different renderers. **DynDocModel** provides tangling functionality, but we will discuss building a set of simple tangling renderers as an illustrative example.

We need renderers for four different types of objects associated with threads: element instances; code elements; structural elements such as tasks, and alternatives; and one for all other types of elements. Our code element renderer simply emits the code content unmodified. We render container elements by rendering each of their children and combining the results. For instances, we render either a) their underlying *DocElement* if they have no children, or b) each of their children. For all other types of elements, we return an empty *character* vector, though we could add a renderer for text elements that emits comments if we desired. We include the code for these renderers in Figure 2.10. Since our tangling renderers will never need the *formatters*, *converter*, or *state* information, those arguments are subsumed in ... for brevity and readability.

```
tangle_code = function(node, ...) node$content

tangle_cont = function(node,  ...) {
  content = lapply(node$children, renderElement, ...)
  unlist(content, recursive = TRUE)
}

tangle_inst = function(node, ...) {
  if(length(node$children))
    tangle_cont(node, ...)
  else
    renderElement(node$element, ...)
}

tangle_other = function(node, ...) character()
```

Figure 2.10: ***Building a set of tangling renderers.*** We construct a tangling mechanism via four interlocking renderers. The renderer for *ElementInstance* objects (`tangle_inst()`) either renders its children, or if it has no children renders the underlying *DocElement*. R code elements (*RCodeElement* objects) are rendered by extracting their code via `tangle_code()`. All *DocElement* classes which do not inherit from *RCodeElement* are “rendered” as empty strings via `tangle_other()`.

To use our tangling renderers, we simply pass them in a list to `writeDynDoc()`'s *renderers* argument, as in the following code

```
rends = list("RCodeElement" = tangle_code,
             "ContainerElement" = tangle_cont,
             "ElementInstance" = tangle_inst,
             "default" = tangle_other)
writeDynDoc(thread, file = "tangled.R",
            renderers = rends)
```

We omit showing the output file here, as it is simply the R code contained in a given `thread`'s code elements.

The `writeDynDoc()` function performs the initial dispatch for renderers; specifically, it calls the `renderElement()` function which acts as a custom dispatcher, identifying and calling the appropriate method for an object given a list of available renderers and a default to use when there is no match. We note that renderer functions can call `renderElement()` themselves, as we see in our `tangle_cont()` and `tangle_inst()` renderers. We pass the full list of specified renderers along in each call to a specific rendering method for this reason.

Finally, renderers are expected to return any text content in the appropriate markup language. When the input and output markup languages are different, this requires that the text be transformed into the output markup language. We provide the `convertContent()` function, which custom renderers can call to perform this transformation. In addition to the content being converted, `convertContent()` accepts input (*in_format*) and target (*out_format*) formats, as well as a list of converter functions. Users customizing conversion behavior will typically modify or extend the list of default converters, which is represented as the `DefaultConverters` object within the package. Converters are organized in a two-level nested list, with output format nested within input format, with either conversion methods (or `NULL`) in the leaf elements. Both levels of nesting also support inclusion of a `default` element, which specifies a function to be called when no more specific converter for the *in_format*-*out_format* pair is present.

2.3.3 Customizing the creation of *DynDoc* objects

Typically, *DynDoc* objects are created via the `readDynDoc()` function. In the simplest case, `readDynDoc()` accepts a file describing a dynamic document – in any supported format – and returns a *DynDoc* object representing the document. By default, the function infers the format of the file from its extension, and can read .Rmd (Rmarkdown), .Rnw (Rnoweb), .Rdb (RDocbook), and .ipynb (IPython Notebook) files.

In addition to providing parsers for reading the formats listed above, `readDynDoc()` allows for customized parsing via the *parser* argument. Parsers must accept a file to parse as well as ... (which is passed directly to the parser from `readDynDoc()`).

To illustrate the concept of a custom parser, we present a parser which reads R source files that have **roxygen2** (Wickham et al., 2014) comments and populates a *DynDoc* which contains the documentation (in an *MDTextElement*) followed by source code (a *CodeElement*) for each documented object in the file. We do not intend this to be a real-world applicable parser, but rather present it solely for instructive purposes.

A parser must do two things: recognize the structure of the file passed in, and use that structure to build and populate the appropriate *DocElement* objects. We leverage existing functionality, namely various machinery in Wickham's **roxygen2** (Wickham et al., 2014) and Murdoch's `parse_Rd()` and `Rd2HTML()` functions from the **tools** (R Core Team, 2014) package. We note that this example was chosen for illustrative power rather than direct usefulness. The code for our custom parser is as follows:

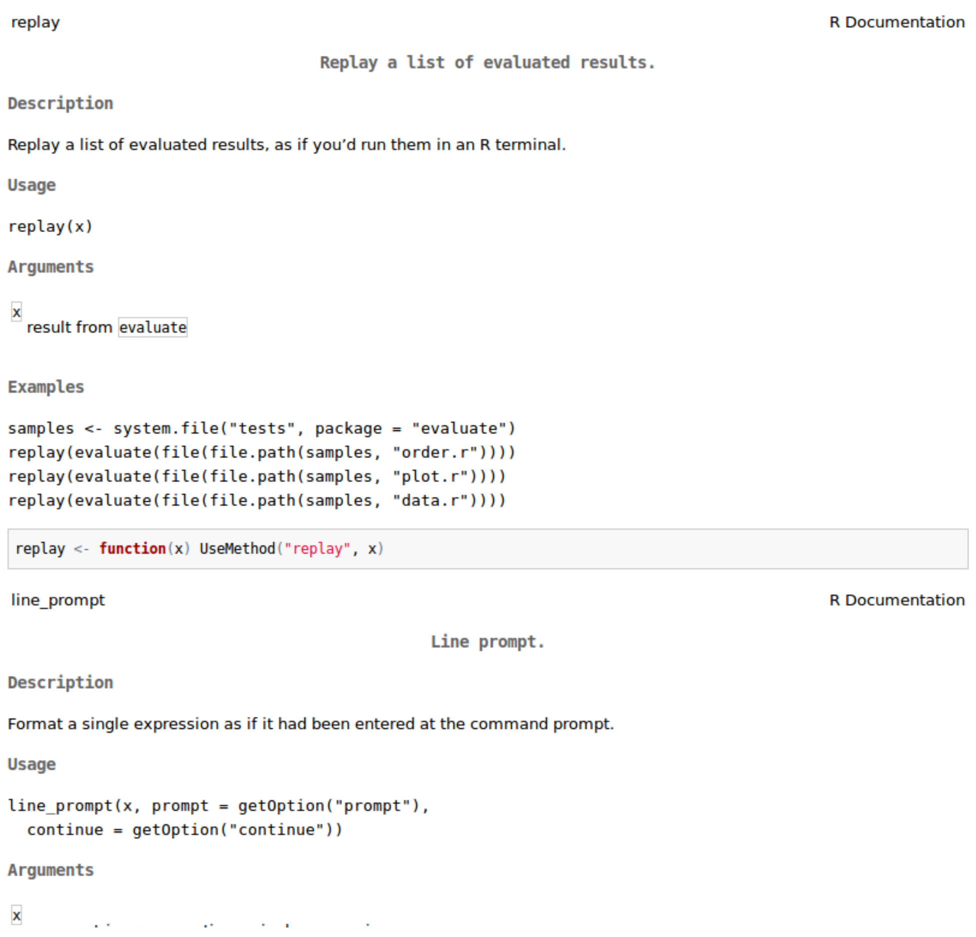
```
roxygen_parser = function(file, ...)
{
    p = parse(file, keep.source=TRUE)
    refs = getSrcref(p)
    comments = roxygen2:::comments(refs)
    keep = sapply(comments, function(x)
                         any(nchar(as.character(x))>0))
    refs = refs[keep]
    comments = comments[keep]
    els = vector("list", 2*length(refs))
    for(i in seq(along=refs))
    {
        rox = do_rd(as.character(comments[[i]]),
                    as.character(refs[[i]]))
        code = paste(as.character(refs[[i]]), collapse="\n")
        els[[2*i - 1]] = new("MDTextElement", content = rox)
        els[[2*i]] = new("RCodeElement", content = code))
    }
    new("DynDoc", children = els)
}
```

We first parse the file using R's `parse()` function, and extract the expressions for the code and comments via `getSrcref()`. Next we get the **roxygen2** style comments via that package's unexported `comments()` function. We use these comments to filter out expressions which are not documented. We loop through the remaining comment-code pairs and create an *MDTextElement* and an *RCodeElement*, adding them to a list of document elements. We generate the Markdown (actually HTML) text via a custom `do_rd()` function. We omit the definition of `do_rd()` here for brevity, but it accepts two *expression* objects – a roxygen comment and the corresponding code – and returns HTML code for the documentation for the *function* or constant defined in the code. Finally, we create a new *DynDoc* object containing our elements.

To use our roxygen source-file parser we simply pass it to `readDynDoc()`. We read in the replay.r file from Wickham's **evaluate** package (Wickham, 2014) (chosen for its short function definitions, and used with generous permission from the author):

```
replayDocs = readDynDoc("replay.r", parser = roxygen_parser)
```

Once read in, we interact with `replayDocs` the same as we would any *DynDoc* document. By calling `writeDynDoc()` (which we discuss in detail in Section 2.3), we get a page that has HTML R documentation interspersed with formatted code, shown in Figure 2.11.


replay R Documentation

Replay a list of evaluated results.

Description

Replay a list of evaluated results, as if you'd run them in an R terminal.

Usage

```
replay(x)
```

Arguments

x
result from evaluate

Examples

```
samples <- system.file("tests", package = "evaluate")
replay(evaluate(file(file.path(samples, "order.r"))))
replay(evaluate(file(file.path(samples, "plot.r"))))
replay(evaluate(file(file.path(samples, "data.r"))))
```

```
replay <- function(x) UseMethod("replay", x)
```

line_prompt R Documentation

Line prompt.

Description

Format a single expression as if it had been entered at the command prompt.

Usage

```
line_prompt(x, prompt = getOption("prompt"),
  continue = getOption("continue"))
```

Arguments

x

Figure 2.11: Using a custom parser to read **roxygen2**-style source files

The concept of custom parsers is key to the universality of **DynDocModel**. We have provided parsers to many current dynamic document storage formats. With the ability to specify a new parser without altering the rest of a workflow we can easily add support for new formats as they arrive.

We have designed the **DynDocModel** API to be both expressive and extensible. Differentiating and exposing the four stages of weaving allows us to provide a general mechanism useful for both customizing the appearance of woven reports (e.g., our styled HTML table) and computing directly on documents or threads (e.g., querying misclassification rates, visualizing the document as a graph, or building a side-by-side comparison page).

2.4 Related Work and Other Approaches

We now present related and preceding work on dynamic documents. In Section 2.4.1, we give a brief, general history of linear dynamic documents in the S (Becker and Chambers, 1984) family of statistical scripting languages (S, S-plus, R). We then discuss existing work relating to non-linear dynamic documents in Section 2.4.2.

2.4.1 Background and systems for linear documents

The first dynamic document system designed for the S family of computing languages was the REVWEB software for S-Plus (Lang and Wolf, 1996). REVWEB is a set of S-plus functions and external command-line tools which combine to allow users to save statistical analyses to file in a form where they can be loaded, partially or fully re-run, and then used to generate a new version of the report, a process the authors call *reviving* the analysis. The file format is based on Ramsey's NoWeb (Ramsey, 1994) literate programming format, which allows authors to intersperse *code chunks* – in this case sets of S expressions – within a LaTeX file.

REVWEB allows users to operate interactively at the individual chunk level, e.g., re-running a code for a particular chunk, or inserting output from the previously run code into the report. This chunk level API allows users to interactively step through their stored analyses, restart an analysis in the middle, or even insert new code into the document.

Leisch's Sweave (Leisch, 2002) was the first dynamic document system to be programmed entirely in S-plus/R. Sweave also transitioned from the interactive, chunk-level command based approach for processing documents to a whole-document approach, with users calling `Sweave()` on Rnw files to automatically generate woven PDF reports by way of LaTeX. This processing-entire-files paradigm has remained dominant in R (and S) dynamic document systems. Sweave was eventually incorporated into the core R distribution, and plays a central – and until the recent introduction of the vignette engine option, exclusive – role in R's package vignette system (R Core Team, 2014).

One exception to the processing-entire-files paradigm among REVWEB's descendants is Gentleman and Gentry's *DynDoc* (Gentleman and Gentry, 2013) package within the BioConductor project(Gentleman et al., 2004). The *DynDoc* package offers a simple object model and chunk level API for Rnw files. Zhang's *tkWidgets* (Zhang, 2013) leverages the *DynDoc* API to offer a vignette explorer GUI (`vExplorer()`) for interactively viewing and executing package Rnw files.

Dynamic documents have also entered the realm of so-called WYSIWYG (what you see is what you get) text editors, e.g., with Kuhn's **odfWeave** (Kuhn, 2014). **odfWeave** allows authors to create dynamic documents in the Open Document Format (ODF) (Brauer et al., 2005), which is used by the word processors in the OpenOffice and LibreOffice productivity suites. Temple Lang and Becker also investigated a dynamic document system for Microsoft Word documents by using styles to identify code chunks in their **RWordXML** (Temple Lang and Becker., 2013) package.

Finally, Xie introduced the Rmarkdown format, a NoWeb based format with text in Markdown rather than LaTeX with his **knitr** (Xie, 2013a) package. Other than the difference in the underlying format, **knitr** reworked the internal machinery and formatting mechanisms from Sweave with an eye to making them more flexible and easily customized. **knitr** has proven to be a popular successor to Sweave, and was recently a contributing factor to the decision by the R Core Team to implement the concept of vignette engines (R Core Team, 2014), which allow package authors to specify the software which should be used to process their vignettes.

2.4.2 XDynDocs, IDynDocs, and Vistrails

The primary existing work for non-linear dynamic documents is Nolan and Temple Lang's **XDynDocs** (Nolan and Temple Lang, 2013). **XDynDocs** uses the Rdocbook (Nolan and Temple Lang, 2014) extension of the DocBook document format to describe and store dynamic documents. The Rdocbook vocabulary includes tags for tasks, decisions, and alternatives which serve as the inspiration for our element types. **XDynDocs** includes some initial experiments in processing and rendering documents containing decisions, though its historical focus has been largely on processing linear documents.

Nolan and Temple Lang use standard XML technologies to query (XPath (Clark and DeRose, 2006)) and process (XSL (Berglund, 2006) and XSLT (Clark, 1999)) via the **XML** (Temple Lang, 2013) and **Sxslt**(Temple Lang, 2011) packages, respectively. Furthermore, XInclude (Marsh et al., 2006) allows Rdocbook files to be constructed modularly. Their use of XML technologies provides a built-in object model. The **XML** package's use of *externalptr* objects to represent XML documents and their elements, however, makes this object model difficult to extend directly via R classes.

XDynDocs uses S4 dispatch to format output objects. This use of the actual output objects – as opposed to captured textual output – served as an inspiration for our system of formatters and renderers. Our extensions to Nolan and Temple Lang's pure S4 system were largely motivated by lessons learned when designing and implementing the API for version 2.0 of Huntley et al's *ReportingTools* (Huntley et al., 2013) BioConductor package.

Nolan and Temple Lang also pioneered the concept of interactive renderings of R-based dynamic documents that allow the reader to control parameters or other aspects of the code via GUI controls with their **IDynDocs** (Nolan and Temple Lang, 2007) package. In some sense this can be seen as an extension of Zhang's vignette explorer from *tkWidgets*, but while Zhang focused exclusively on executing code chunks, perhaps repeatedly or out of order, Nolan and Temple Lang provide a mechanism to *alter and then re-run* code chunks, making the focus and scope of their work quite different.

Beyond invoking the evaluation of code chunks via GUI buttons, **IDynDocs** allows interactive controls like sliders, drop down menus, etc to control the actual computations performed. For example, an author using a kernel-based smoother might allow readers to set and change the bandwidth used via a slider, or select the type of kernel via a menu. This serves as the direct inspiration for our interactive code elements.

2.4.3 Other related work

We also build on ideas from other realms of research. The concept of recording analysis sessions in a queriable and replayable manner has received quite a bit of attention in the arena of interactive visualization environments.

Vistrails (Bavoil et al., 2005) is a provenance and scientific workflow system. While it is not a document system *per se*, Vistrails shares many conceptual features with the comprehensive documents we propose. It allows scientists to define non-linear workflows which encompass analytic choices in a way similar in function to our decision elements. Vistrails also allows results to be embedded directly – and dynamically – within $\LaTeX$ documents. Furthermore, VisMashup (Santos et al., 2009) allows VisTrails-based results to be associated with GUI controls similar to those produced by **IDynDocs** and the present work.

In the context of visualizing query results from a database, Lee and Grinstein propose storing the queries used and information about the results within the database itself(Lee and Grinstein, 1995). This allows the analyst to query and reason about both what was done and what the results were, both major focuses of our comprehensive documents.

Some visualization exploration systems also recorded certain types of decisions by the analyst, similar to our proposed documents. The GRASPARC system (Brodlie et al., 1993) defined analysis questions as computational questions answerable by a particular sequence of steps which rely on a predetermined set of modifiable parameters. Decisions in this context were then defined as the analyst modifying one or more of these parameters and restarting the analysis at a particular step. These decisions were represented as branches within a tree structure representing the full history of actions in the session. These systems do not deal in documents in the sense we use the term. Even so, they provide context and insight into the value of having more complete information about the research process available after the fact.

We have presented the **DynDocModel** R package. Our package offers an accessible, unified object model for dynamic documents, including non-linear comprehensive documents which capture a more complete picture of the data analytic research process.

DynDocModel offers the ability to perform both traditional processing (e.g., weaving dynamic reports) and less traditional computations on the document. Furthermore, each weaving subsystem (reading in document, selecting a subset of content, evaluating code in a document or thread, transforming content and generated output into the final output form, and constructing the report itself) is highly customizable, with full control given to the user.

We now turn to the task of creating non-linear documents. In our next chapter we discuss our custom fork of the popular IPython Notebook interactive computing platform (Pérez et al., 2013a), which allows authors to construct documents with tasks, decisions, and different levels of detail in a GUI driven interactive system.

Chapter 3

Authoring and Exploring Non-Linear Documents

We saw with our housing and digit classification analyses in Chapter 1 that non-linear documents can capture a richer and more accurate picture of the data analysis process. Decision elements allow analysts to preserve abandoned analysis strategies or implementations without interfering with the ability to run and view the results of the current strategy. We saw this in our digit example, where we had working code for generating random forest (Breiman, 2001) and k-nearest neighbor (Cover and Hart, 1967) classifiers from our training data, even though only one such classifier will be generated when producing the final narrative (e.g., a report describing the recommended strategy). Task elements further enhance an author's ability to capture the research process by grouping content in the document and adding semantic information.

In previous chapters, we have taken as given the functionality required to create, display, and explore non-linear documents. In this chapter we will discuss specific features we feel are important components to a system for authoring and exploration of these documents. We also present our work on modifying Granger and Perez et al's IPython Notebook (Pérez et al., 2013a) platform to support authoring and exploring documents which capture more of the research process.

The benefits of having non-linear documents are largely moot if those benefits are outweighed by the inconvenience of creating the documents in the first place. Users can currently create non-linear documents, e.g., in Nolan and Temple Lang's Rdocbook format (Nolan and Temple Lang, 2013), directly in a text editor such as Emacs (Free Software Foundation, 2013). This process provides complete control to the author, but is also somewhat cumbersome and may dissuade many analysts from exploring the possibilities of non-linear documents.

We seek to provide a convenient tool that simplifies and makes more intuitive basic activities such as constructing decision and task elements. In particular, our goal is to allow the capture of the research structure within these documents to be seamlessly integrated into the analysis process itself. We identify and discuss specific features necessary for this type of integration in Section 3.1.1.

The ability to select narrative threads interactively at viewing time by navigating between alternatives considered by the analyst enables the reader to explore the research process. To do this type of exploration, users must be able to see both the structure and content of the document simultaneously, and there must be a mechanism for selecting specific threads or alternatives. Beyond the ability to display the content, however, the exploration environment must be able to update the currently displayed output to reflect the thread selected by the user.

A straightforward way to update the displayed output when the thread changes is to run the code corresponding to the new thread and display newly generated output. This requires the system to be able to run code on command at viewing time, which the single thread PDF and static HTML renderings we saw in Chapter 2 are unable to do.[1]

With the ability to perform new computations at viewing time, we can support interactive code elements. Interactive code elements combine code expressions with GUI controls which viewers can use to set parameters or other values within the code. When readers use these controls, any

[1]Dynamic HTML renderings which can do this are possible, and we discuss briefly in Section 3.5.

existing output for the element is replaced by new output generated by (re-)evaluating the altered code. This allows readers to view new output corresponding to different parameter choices without writing any new code themselves. We discuss the concept of interactive code elements, as well as our desired features for an exploration environment more generally, in Section 3.1.2.

We present our modification of the IPython Notebook as a case study for what a non-linear document system might look like and how we altered an existing linear document system into one that supports non-linear documents. Modifying IPython Notebook has allowed us to both create an authoring and exploration environment for non-linear documents and gain insight into some general design requirements for these document systems.

The remainder of the chapter is organized as follows: Section 3.1, discusses our desired feature sets for authoring and exploration environments. We then present our IPython Notebook-based case study in Section 3.2. In particular, this includes a discussion of the core IPython Notebook program and a detailed discussion of the modifications required for each of three features we implemented: non-linear notebook support, multiple detail levels, and interactive code elements. Section 3.3 discusses the results of our case study, including both the general insights we gained into non-linear document system design and how well our modified IPython Notebook performs as an authoring and exploration system. Finally, in Section 3.4, we provide some background about related and similar work and discuss alternative approaches before concluding the chapter.

3.1 Important Features for Non-Linear Document Systems

We first describe what we consider core features for authoring and exploration environments for non-linear documents. For authoring systems, these center around creating the content and structure which make up the desired documents. In the case of exploration environments, the focus is more on the ability of the reader to understand the completed research described in a non-linear document. We discuss systems for creating these documents first, followed by environments for exploring them.

3.1.1 Features of an authoring environment for non-linear documents

We want a system which allows analysts to add structural elements – decisions, tasks and alternatives – which organize the code and text within their documents, and to add metadata to elements of all types. Ideally, the process of constructing a non-linear dynamic document can be fully integrated with the analysis process itself. R-based integrated development environments (IDEs), such as Emacs Speaks Statistics (ESS) (Rossini et al., 2001) and RStudio (RStudio Team, 2012), already offer substantial support for writing R scripts in the form of Rnoweb (.Rnw) or Rmarkdown (.Rmd) dynamic document files. We seek to extend this type of integration to non-linear documents, allowing the analyst to organize content not only into chunks of code and text, but also into a structure which captures the research process itself.

Consider what extra capabilities we need, beyond providing a sequence of text and code content, to construct the dynamic document describing our digit analysis. The analysis is discussed in detail in Chapter 1. The basic flow of this analysis is: we read in the data, explore the raw data, compare binary classifiers on data from the one and eight classes, fit classifiers for the whole dataset and choose one, and finally assess the results.

Before building the full 10-group classifier, we examined several methods for the simpler case of classifying ones and eights. Our exploration of these possible binary classifiers is a task. The analyst can create this task proactively, by starting with a task element then adding children, or he might do so retroactively, by applying the various methods under consideration and only then organizing them as a task with multiple subtasks. The analyst would likely not construct all these subtasks at the same time. Rather, he would likely pursue one strategy until it is complete, e.g., focusing on applying the linear discriminant classification method to the raw data before moving on to another method such as quadratic discriminant analysis or random forests. Thus we need the ability to create tasks – and other structural elements – and to insert new or existing text/code content into a task.

When the digit analyst attempts to apply the linear discriminant method to the raw data, he receives an error because the data are not full rank. He might add metadata to the relevant task indicating that an error is expected, and another piece of metadata to the full exploration task indicating that the task will not be required to generate his final results from the raw data, and so may not be displayed when rendering or viewing the document for certain purposes.

The analyst can construct the element representing his decision between classifiers for the full dataset in much the same way we described for his task element(s). This will constitute a single decision element containing alternative elements for each of the two strategies (KNN and random forests). He might start by pursuing the random forest and KNN strategies separately, before organizing these groups of content as alternatives within a single decision element. Alternatively, he might create his decision element first and then create the code and text for each of the alternatives directly within it. To do this he must be able to navigate within the non-linear structure of the document he is creating, so that he can select the second alternative as the place to create new content.

To summarize, then, we have identified four capabilities which are central to creating non-linear documents:

- Create content, with code and text differentiated
- Create structural elements (tasks, decisions, and alternatives), navigate amongst them, and

place content within them

- Add metadata to the content and structural elements, such as information about errors or data provenance
- Run code and capture output, including code contained within structural elements

With these abilities we can capture the research process as it happens. This allows analysts to create the type of non-linear dynamic documents we describe in Chapter 2 as they carry out their analyses, while the necessary information is still fresh in their minds.

Another approach for creating non-linear dynamic documents is to add structural elements via post-processing. We could do this by taking a flat dynamic document which includes all the relevant code (and potentially some text) and using **DynDocModel**'s facilities to programmatically modify the document. To do so, we need to know how the document should be modified. We can achieve this by possessing specific knowledge about the document/analysis, or by automatically detecting implicit structures based on analysis of code in the document. Temple Lang et al have investigated using heuristics to detect tasks within scripts based on variable usage and specifically variable redefinitions.[2](Temple Lang et al., 2013) Converting existing linear documents into non-linear documents which expose the relationships between pieces of content can be very useful, but we consider it a separate topic which we will not discuss here.

We now transition to discussing our desired features for exploration environments. Exploration requires some of the same types of activities as authoring, such as navigation through the non-linear document. Other required features are quite different, however, which we see in the next section.

[2]A variable redefinition is when an existing variable is assigned a new value which does not depend on its previous value.

3.1.2 Rendering target and exploration environment

We saw in Chapter 2 that our **DynDocModel** package can weave output reports in various formats (e.g., PDF, HTML) for each thread through a dynamic document. We also saw that non-linear documents can have many threads, each corresponding to a different potential narrative or set of strategies within the analysis.

Consider a report woven from a thread through the housing analysis example from Chapter 1 which uses the median for all "typical price" calculations and excludes the optional univariate explorations. A viewer reading that report has no way of readily seeing the results or code from when the mean was considered, or that an entire section of univariate explorations was performed but excluded from this output view. The reader may also be unable to detect the multiple options considered when the analyst cleaned the data, and made the plots during the analysis process, depending on how these steps are described in the text. To get a full picture of her analysis, then, the reader must be able to see the structural elements (tasks, decisions, alternatives), the content within those elements, and the output generated when code in the document is run. For this purpose we need the ability to create partial and full views of non-linear documents and display them for human consumption and understanding.

Output from running particular code elements in a non-linear dynamic document often depends on the results of running other, earlier code elements in the document, and these dependencies vary between different threads. For example, the data used to construct the plot of resale rates against typical house price by city depends on choices made in three decisions earlier in the analysis. The most obvious is the analyst's choice of whether to use mean or median as the "typical price" variable. The second relevant decision is how to perform quality assurance (QA) on the city variable. Specifically we must choose if and how to combine the values `"Belvedere/Tiburon"`, `"Belvedere/tiburon"`, `"Belvedere"`, and `"Tiburon"`. Belvedere and Tiburon are "twin cities", two historically distinct cities that have grown together, so multiple combination strategies appear reasonable, but each one will result in a different number of points in her resale rate versus city price plot. Finally, in the third decision, the analyst chooses the minimum time between two sales of the same property before she considers them separate transactions and thus a resale. For this reason, the analyst's single code element for generating the plot can produce any one of 16 distinct images depending on previous choices. In longer or more complex analyses this number could easily be much higher for certain code elements.

The ability to see the output for a code element that corresponds to a specific set of choices is highly useful. This allows the viewer to explore the decisions made by the analyst and the implications those decisions have on the graphical and numerical results generated. We referred to this as *exploring the analysis process* during our discussion of motivations in Chapter 1. It is not feasible in general to show all possible outputs for a code element when displaying a full-structure view of a non-linear document due to the type of combinatorial explosion described above. Thus we need the ability to specify a particular thread or set of choices for which to display output. Once this is done, we also need the ability to display the correct output for the viewer's selected thread.

Given the ability to navigate through the structure of a non-linear document and select specific alternatives, the most straightforward way of ensuring the correct output is displayed is to evaluate the correct code as the alternatives are being chosen. This differs from traditional weaving (e.g., transforming Rmd files to HTML or Rnw files to PDFs) in two ways. First, it happens interactively within the platform used to view the document, rather than within R during an earlier report generation step. Secondly, the (sub)set of code being evaluated is not necessarily contiguous within the document. This requires that the system used to display the documents be able to perform new computations on command.

With a platform that can evaluate code at viewing time, we can go further. We can also support graphical user interface (GUI) controls such as sliders which alter parameter values used in a code element, rerun the code, and update the displayed output. Nolan and Temple Lang investigated this concept with their **IDynDocs** package, particularly in the context of pedagogical documents(Nolan and Temple Lang, 2007).

To summarize, then, we want a file format and display system which allows users to:

- View the structure of the full document;
- Hide and show structure and content interactively;
- Navigate the structure of the document and select alternatives/threads;
- Run selected code/alternatives/threads and view the output within larger document structure;
- Control specified interactive code elements via GUI controls.

There are multiple ways to build a system which meets these goals. We discuss our fork of the IPython Notebook project, which provides these features, in Section 3.2, while we discuss alternative approaches in Section 3.5.

3.2 Case Study: Modifying IPython Notebook

We have chosen to modify the IPython Notebook application (Pérez et al., 2013a) to support the authoring and viewing features we described in the previous section. IPython Notebook provides us with a solidly engineered front end to the IPython scientific computing environment (Pérez and Granger, 2007), which allows us to explore our ideas.

We present this work as a case study exploring the characteristics of a system for non-linear documents, and to demonstrate the types of changes necessary to modify an existing linear document system to support the creation, display, exploration, and interactive execution of non-linear documents. As a single case study, this work seeks to provide neither a general template for, nor exhaustive insights into, modifying existing software to support the type of non-linear documents we discuss in Chapters 1 and 2. We do feel that we gained some valuable insight into the types of requirements non-linear authoring systems will have, however, which we discuss near the end of this chapter.

We make three major changes to the design/behavior of IPython Notebook to allow it to act as an authoring and exploration environment for non-linear documents. First, we extend the concepts of notebook and cell (element) to support nesting – both in general and in the form of specific task, decision, and alternative element types – when creating, viewing, and running ipynb files. Second, we leverage the concept of metadata to add support for multiple detail levels and the ability to interactively show/hide extra detail. Finally, we implement support for interactive code elements which allow the user to control aspects of the code via GUI-style controls such as sliders or drop-down menus.

We now briefly describe the unmodified IPython Notebook application, with specific focus on design elements and features relevant to our goals. Following this we discuss each of our three changes in detail.

3.2.1 The IPython Notebook

The IPython development team describes their IPython Notebook application as "a web-based interactive computational environment where you can combine code execution, text, mathematics, plots and rich media into a single document" (Pérez et al., 2013a). These documents, called notebooks, are what we refer to as linear dynamic documents in Chapter 2.

A notebook consists of a linear sequence of code and text elements.[3] The text elements can contain raw text or markdown which is rendered into HTML during viewing. The markdown renderer is extended to support $\LaTeX$ via MathJax (Cervone and Krautzberger, 2013), thus supporting the mathematics, images, and rich media referred to in the quote above.

Users create and evaluate content in the IPython Notebook application at the level of individual elements representing chunks of either code or text. When a code element is evaluated, all output – including any printed text, plots, warnings, or errors – is captured and inserted directly into the notebook, replacing any previous output for the element. Using IPython Notebook, we might create a code element and write code in it which generates a plot. We could then evaluate this element, see the plot we have generated, then create a text element below the plot where we discuss what we see. A loaded notebook, then, is essentially a woven report which is being interactively constructed, modified, and evaluated by the author.

Furthermore, users can select and evaluate any single code element within the loaded notebook at any time, regardless of its position in the notebook and what other code has been run. This allows the author to backtrack, iterate on an element or elements, and then see the results without having to re-evaluate the entire notebook. This feature is quite useful for navigating between alternatives in non-linear documents, as we will see shortly.

IPython Notebook also provides authors and viewers the ability to navigate through a notebook. This includes selecting the next and previous elements, inserting new elements at particular positions in the notebook, and even evaluating the entire notebook or the notebook up to the current element. These concepts are straightforward for linear documents and will serve as our starting point for navigation in non-linear documents.

[3]Notebook elements are referred to as *cells* in the IPython code and documentation.

Code in IPython Notebook is assumed to be Python. Other languages, including R, are supported via IPython's "magic functions". With the R magic, we are able to write R-based analyses using this application. In fact, we wrote both of the example analyses discussed in Chapter 1 with our modified version of the IPython Notebook using the R magic.

Each element in a notebook, as well as the notebook itself, can be assigned arbitrary metadata attributes. This is similar in form to the metadata we support in **DynDocModel**, though the purpose is somewhat different. The core IPython Notebook application does not use element metadata other than allowing users to view and edit it interactively. Instead, the developers intend this metadata to be used by extensions to the application in order to modify behavior without requiring modification of IPython Notebook's internal machinery. Avila, for example, utilizes element metadata in support of his Reveal.js-based live slideshow extension (Avila, 2013).

Our concept of viewing documents at different levels of detail is centered around this built-in concept of metadata in IPython Notebooks. We allow authors to identify content that represents "extra" detail via metadata, and this content can then be shown or hidden via a slider in the page displaying the document. We discuss the details of how we do this, and the implications of having content that can be hidden, in Section 3.2.3.

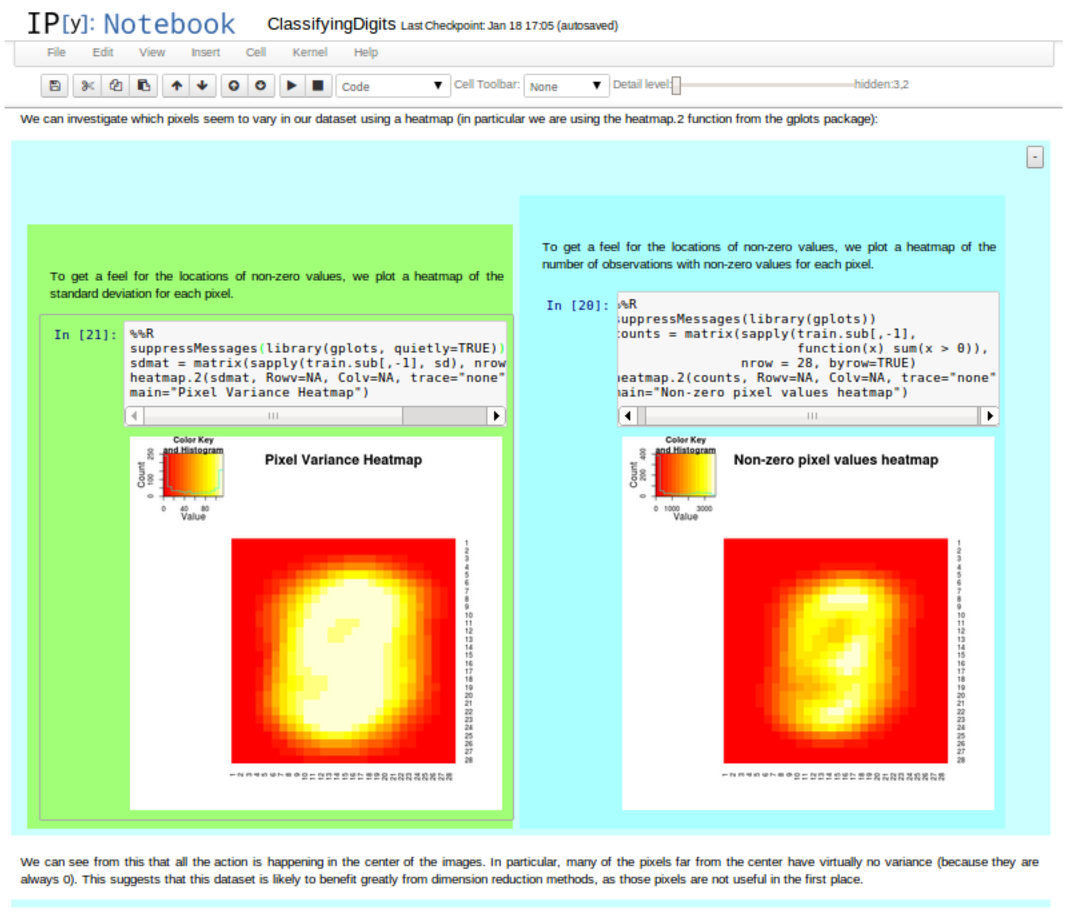
Finally, notebooks can be saved, shared, and transported via IPython's custom ipynb format. Unlike most dynamic document formats, ipynb files include the text, code, and most recently captured output for the notebook. In a sense these ipynb files can be considered a combination of a linear dynamic document file and the report generated by weaving the file.

The IPython Notebook application involves a great deal more design and engineering than we have listed here. We refer interested readers to the IPython technical documentation (Pérez et al., 2013b) for a more complete discussion of the application. We now move on to our modifications and extensions of the features outlined above.

3.2.2 Adding non-linearity

Implementing support for non-linear documents in general will require three things. First, we must create a way to represent non-linear notebooks both in the application and in an extension of the ipynb format. Secondly, we must identify and rework any design elements or internal machinery which rely on the assumption that notebooks are linear. Finally, we must implement any necessary functionality specific to non-linear documents, such as displaying and interacting with task, decision, and alternative elements. For our case study, our goal is to implement these features with as little perturbation to the existing IPython Notebook machinery as possible.

We represent the structure of non-linear notebooks via nesting, as Nolan and Temple Lang do in **XDynDocs** (Nolan and Temple Lang, 2013) and we have done in **DynDocModel**. We create three new types of elements within the IPython Notebook codebase: tasks, decisions, and alternatives. Task and alternative elements display their child elements in sequence vertically. Decision elements, on the other hand, display their children (alternatives) side-by-side, with the option to hide all but the currently selected alternative. Figure 3.1 contains a screenshot of a decision element from our digit classification analysis discussed in Chapter 1. We briefly showcase our changes to the ipynb format itself in Appendix A.



ClassifyingDigits Last Checkpoint: Jan 18 17:05 (autosaved)

We can investigate which pixels seem to vary in our dataset using a heatmap (in particular we are using the heatmap.2 function from the gplots package):

To get a feel for the locations of non-zero values, we plot a heatmap of the standard deviation for each pixel.

```
In [21]: %%R
suppressMessages(library(gplots, quietly=TRUE))
sdmat = matrix(sapply(train.sub[,-1], sd), nrow
heatmap.2(sdmat, Rowv=NA, Colv=NA, trace="none"
main="Pixel Variance Heatmap")
```

To get a feel for the locations of non-zero values, we plot a heatmap of the number of observations with non-zero values for each pixel.

```
In [20]: %R
uppressMessages(library(gplots))
ounts = matrix(sapply(train.sub[,-1],
                      function(x) sum(x > 0)),
               nrow = 28, byrow=TRUE)
eatmap.2(counts, Rowv=NA, Colv=NA, trace="none"
ain="Non-zero pixel values heatmap")
```

We can see from this that all the action is happening in the center of the images. In particular, many of the pixels far from the center have virtually no variance (because they are always 0). This suggests that this dataset is likely to benefit greatly from dimension reduction methods, as those pixels are not useful in the first place.

Figure 3.1: ***A decision element with two alternatives.*** The digit analyst we discussed in chapter 1 chooses between displaying the location of ink in the images in two slightly different ways early in his non-linear notebook. On the left, the text, code, and output correspond to a heatmap of the standard deviations of each pixel in the grid, whereas on the right they relate to a heatmap of the counts of non-zero values for each pixel across all observations in our training set.

There are two contexts in which users typically change which notebook element is currently selected in IPython Notebook: progressively evaluating the code within a notebook (e.g., via the Shift-Enter shortcut), and navigating through a document while editing its structure or existing content (e.g., via the up- and down-arrow keys). We will refer to these as the *evaluation* and *editing* navigation contexts, respectively. For linear notebooks, elements are visited in the same order regardless of which context – dictated by the author or viewer's intent and mode of operation within the IPython Notebook application – is used. With non-linear notebooks, however, this is no longer the case. For example, when editing a notebook we want to be able to enter a selected task element to visit its children, ensuring we can reach all of the notebook's content. During evaluation, however, executing a task encompasses evaluation of its children, meaning that they should not be explicitly visited individually after the evaluation of the task is complete. Similarly, evaluating a particular alternative excludes all other alternatives within the same parent decision.

We first define the *editing* navigation order. We want a *next-element* operator which allows the viewer to visit each element in the notebook once by traversing it start to finish, and which matches the naive approach when applied to linear notebooks. Because we have multiple levels of nesting within our documents, we must choose whether to visit structural elements or their children first; we elect to visit structural elements first, following a top-down traversal order.[4] This makes the concept of evaluating structural elements much more powerful, as we will see later in this section. We illustrate our traversal order in Figure 3.2, after which we define a next-element operator which achieves it.

We define our next-element operator for the editing navigation context piecewise as a series of contextual rules:

- When a code or text element which is not the last child within its parent is selected, the next element is the following sibling of the current element.
- When an element which contains children is selected, the next element is defined as the first child of that element.
- When the selected element is a code or text element which is the last child of a structural element, the next element is the *parent element's first following sibling*. If there is no following sibling for the parent, the grandparent is used, continuing until we find a trailing sibling or reach the top level.

[4]Note we are defining the visitation order based on usefulness when viewing/interacting with the documents, rather than to faithfully recreate the exact order in which actions were taken during the original analysis.

- If a code or text element is the last child of its parent and none of its ancestors have trailing siblings, there is no next element.

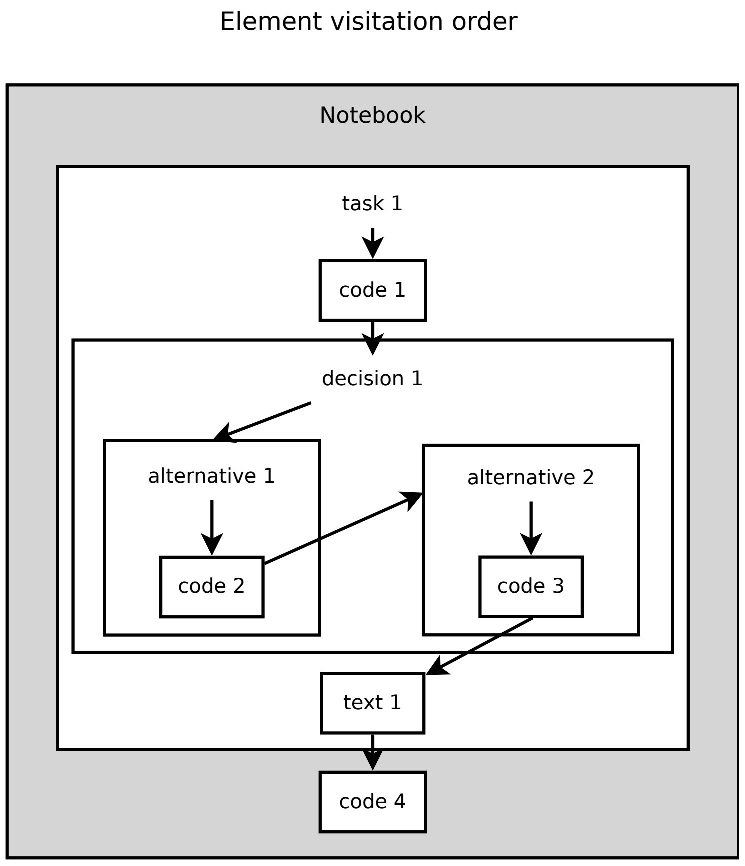


Figure 3.2: ***Desired element visitation order for a non-linear notebook.*** Structural elements such as tasks, decisions, and alternatives are visited before their children, and the children of a structural element are visited before that element's following siblings. This makes the concept of evaluating structural elements, which we describe later in this section, much more powerful.

One of our primary workflows when dealing with notebooks in the unmodified IPython Notebook is to step through the cells one at a time, evaluating any code and replacing any currently displayed output with a freshly generated version. The IPython developers have provided a shortcut (SHIFT and the Enter key) which does this. The shortcut goes further, however, by adding a new element if it detects that it is at the end of the notebook, thus making continuously authoring and then evaluating new content extremely convenient.

We now define the second of our two navigation orders, the one used in the evaluation context. As implemented in IPython Notebook, the evaluate-and-select-next operator assumes that each element is either code or text, and that each element should be visited during this process. Neither of these assumptions necessarily hold with our modified version. When evaluating code in non-linear notebooks, we typically want to run only a subset of the code contained in the document. Often, this code will be contained in a specific thread through the document, though it need not be. For example, we might run the code which performs the strategy ultimately chosen by the analyst, or code implementing a perturbation of that where specific different choices were made. During navigation, then, we would want to step from one element *to the next one in the currently active thread*. This is where navigation and stepping through the document differ for non-linear notebooks in a way that they did not for linear ones.

Essentially we want to step through a single thread embedded within the loaded document. Thus we need a way to identify which thread to step through, and a way to selectively execute the content of that thread in a convenient manner. This involves defining custom execution mechanics for each of our new structural element types. For tasks, stepping through the element involves stepping through each of its children. The task element in this case is acting as a collection of content we wish to evaluate together (sequentially). The same holds true for alternative elements. Decision elements, however, are different. When stepping through a decision we evaluate only the currently active alternative, skipping its inactive siblings. Evaluating the notebook, either all at once or via interactively stepping through, then, amounts to weaving the thread defined by the choices of alternatives most recently activated by the viewer.

Finally, we have the concept of inserting a new element when there is no next element. For linear notebooks this is unambiguous: when stepping through the last element in the notebook, create a new element, otherwise select the next element. We could leave this mechanism unchanged, but the concept of the "last" element is somewhat more complicated when nesting is present. The issue is whether the new element is added as a top-level child at the end of the notebook, or as a sibling to the current element (which may be nested within other elements).

We choose to modify the new-element-at-end mechanism slightly so that whenever a non-alternative element which is the last child of its parent is stepped through, a new element is added after it within that parent. For example, this behavior allows us to conveniently build up task elements by creating the task and then incrementally adding cells to the task via the standard – and convenient – IPython Notebook workflow. Furthermore, this behavior simplifies to that of the unmodified IPython Notebook application in the case of linear notebooks, thus constituting a strict extension of the feature to support non-linear documents. There is a mild inconvenience after constructing the last element in a task in that the step-through method cannot be used. In our experience constructing these notebooks, however, this has been outweighed by the ease of constructing multi-element tasks and alternatives.

With these concepts we are able to meaningfully allow so-called headless, or non-interactive, notebook execution. With our execution methods for the structural elements as we described above, it suffices to execute each top-level element. With our definitions of executing structural elements – execute all children for tasks and alternatives, execute the currently selected alternative for decisions – this translates into evaluating the currently selected (or default, e.g., for headless execution) thread through the document. The same goes for using the "execute full notebook" option within our modified IPython Notebook GUI. As with other features, these mechanisms simplify to the behavior of the original IPython Notebook when operating on linear documents.

3.2.3 Detail level and "hidden" elements

Next we implement support for hiding and showing content interactively without actually adding and removing it from the notebook. Recall that both our housing and digit classification analyses from Chapter 1 have content which represents a key portion of the analysis but is not included in a concise discussion of the final results. In the case of the digit classification, this includes the plotting of an observation in order to get an understanding of the data, as well as the exploration of classifiers for just two digits.

Including optional content in our documents can be valuable even in non-interactive settings. With processing systems such as our **DynDocModel** presented in Chapter 2 and Nolan and Temple Lang's **XDynDocs** (Nolan and Temple Lang, 2013) we can generate different PDF or static HTML reports aimed at different audiences by including or excluding this optional content during processing.

Optional content arguably becomes more useful, however, when the viewer can show and hide it on command while viewing the document. For example, a student or collaborator may be viewing our digit analysis and find themselves wanting more detail about the form of the data themselves. With the ability to hide and show content on the fly, they can quickly show the content where we explore and plot a single observation to gain additional insight on this subject, only to hide it again to unclutter the narrative once they are comfortable.

We implement a *detail level* system in our fork of IPython Notebook which allows authors to mark elements as being a particular level of additional detail. Notebook authors set the detail level of elements via the existing metadata mechanism provided by IPython Notebook.[5] We provide a slider which allows the viewer to choose which detail level to view for the entire document, and

[5]Specifically by setting the detail field within an associative array stored in the dyndocmodel field of the metadata.

which displays the currently available detail levels for the loaded notebook. Content with a detail level higher than the currently selected viewing level is hidden both visually and with respect to navigation through the notebook. Figure 3.3 provides a screenshot of a zoomed-out section of our digit classification document at three different detail levels, though the slider is not easily visible at this scale. This system could be extended to allow more local control over content being displayed and hidden.

In Figure 3.3 we see three levels of detail for a section of the digit classification analysis we discuss in Chapter 1. In the default narrative (detail level one) pictured on the left in the image, we assume that the description “pixel level data for images of handwritten numeric digits” is understood. This narrative transitions directly from loading the data to investigating the variances of the pixels across the dataset.

Viewers who would like a more concrete description of the data, however, can opt to increase the detail level to two. This presents them with the view pictured in the middle screenshot, where a single observation (a nine) is plotted as an image. This illustrates more clearly that the columns in our dataset represent positions in scanned images of handwritten digits, with non-zero values indicating the location of ink on the paper. Looking closely at the code for this plot, however, the viewer would see that the observation is transformed before being plotted.

Readers can view the details of that transformation and why it is necessary by increasing the detail level to three, generating the view on the right. In this version of the narrative we first see our chosen observation plotted naively, resulting in an image that is clearly either a nine or a six, but which either way is not oriented correctly. How to fix this is then briefly discussed, transitioning into the second level of detail within the content. Nolan and Temple Lang suggest that the ability to allow readers to “drill down” to receive a more detailed discussion of the content can be particularly beneficial in a pedagogical setting (Nolan and Temple Lang, 2007).

Support for levels of additional detail requires us to relax a major assumption made by IPython Notebook: namely that all content in the notebook should be displayed at all times and should be visited during navigation in the order it appears. Detail content is part of the notebook and has a fixed position within it, but when hidden it should be neither displayed nor visited during navigation or evaluation.

Recall that indexing, position, and the navigation orders within the notebook depend on the position of `<div>` HTML nodes which have the "cell" CSS class, indicating they represent the notebook elements in the page. We can easily make these `<div>`s invisible by manipulating their style information. If we stop at making the `<div>`s invisible, however, the element would still be visited when navigating or stepping through the notebook. We do not wish to remove hidden elements from the notebook entirely, however, for two reasons. First, the position must remain unchanged, even when other elements are added to or removed from the notebook. Secondly, hidden content must be included when saving the document, which is done by traversing the notebook element-wise and adding each element to the object eventually written to file.

To allow elements to remain in the notebook, but only be counted by some operations (saving, loading) and not others (navigation, evaluation) we split IPython Notebook's special CSS class for element `<div>`s into two: "cell" and "hidden_cell" class. When the viewer changes the detail level via the slider, the metadata for each element in the notebook is inspected and its CSS class is changed if necessary.

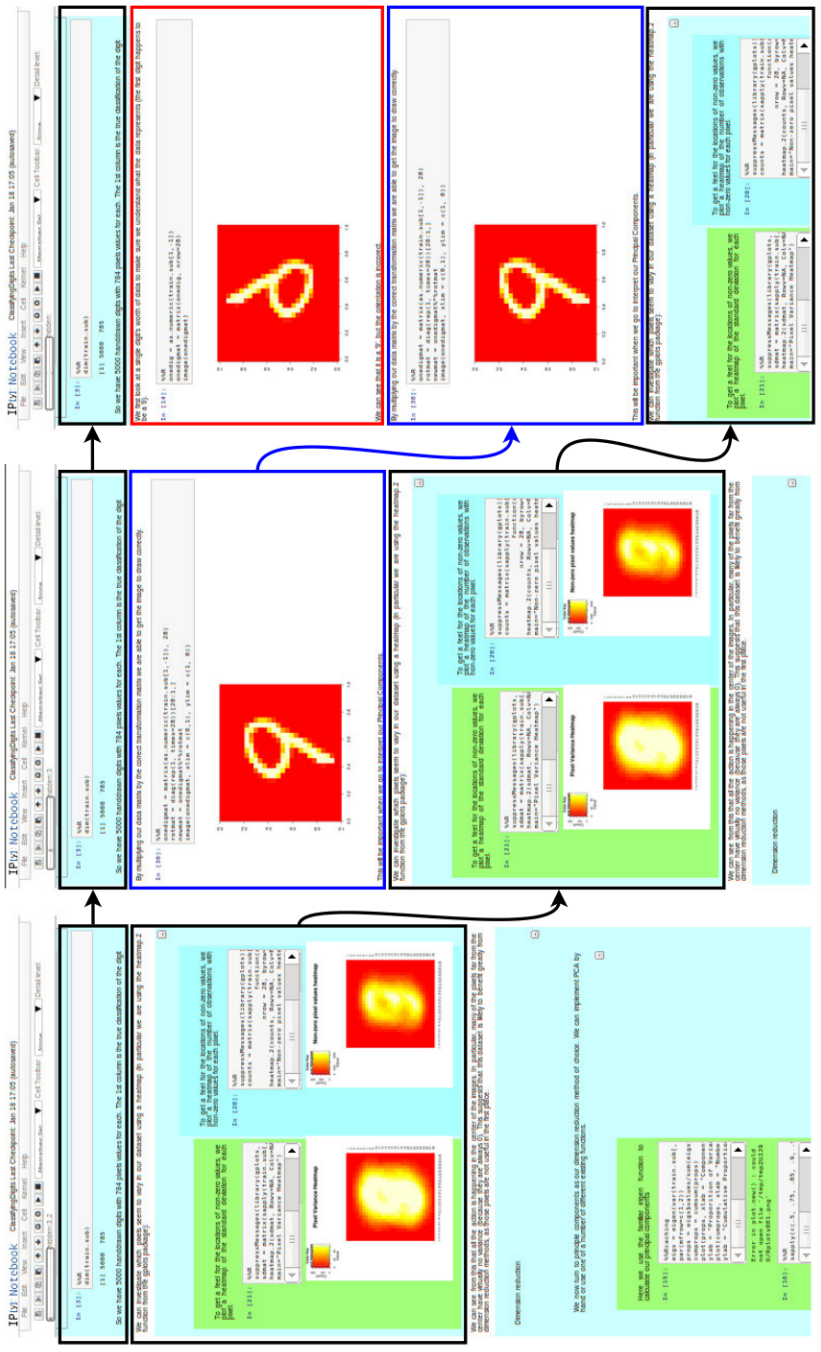


Figure 3.3: ***Detail levels in our IPython Notebook fork.*** Though the resolution of the above screenshots is not sufficient to read the text of the pages, the displayed structure – and effect of changing the displayed detail level (one, two, and three in the left, middle, and right pages, respectively) has on it – is clear. Some sections of content in the notebook are annotated with boxes colored by the content's detail level – black for level one, blue for two, and red for three – and the same content is connected by an arrow between pages where applicable. We can see detail levels are hierarchical, with higher detail levels including all content from lower levels. Thus moving to higher displayed detail level displaces content from the lower level(s) to further down on the page.

Since hidden elements should not be visited or executed, we are able to leave most machinery which looks for the “cell” class unchanged. The exception is that we implement a function which retrieves all cells and modify the notebook saving machinery to use it, which ensures that hidden content is properly saved.

Locating the next element, for both navigation and evaluation, still consists of walking through the list of `<div>`s with the “cell” CSS class via the algorithms we describe in the previous section. Inserting elements into the notebook, however, requires more care when hidden content is present.

Consider inserting an element into an existing document. Typically inserting an element after the selected element and inserting it before the element after the currently selected one are equivalent. In fact, in IPython Notebook, they are implemented via the same machinery, with insertion before an element at index n, simply inserting the element after the element at index n-1. This works well because there is only one possible position between two sequentially adjacent elements. In the presence of hidden content, however, this is not the case. The new element can be placed before or after any hidden content between the selected element and its sibling, as illustrated in Figure 3.4.

The issue with insertion in the presence of hidden content arises from the fact that the concept of “the directly following element” is no longer unambiguous. The next element in the full notebook is no longer guaranteed to be the next element which is currently displayed. Thus we decouple the insertion before and insertion after operations, and perform each with respect to the entire notebook.

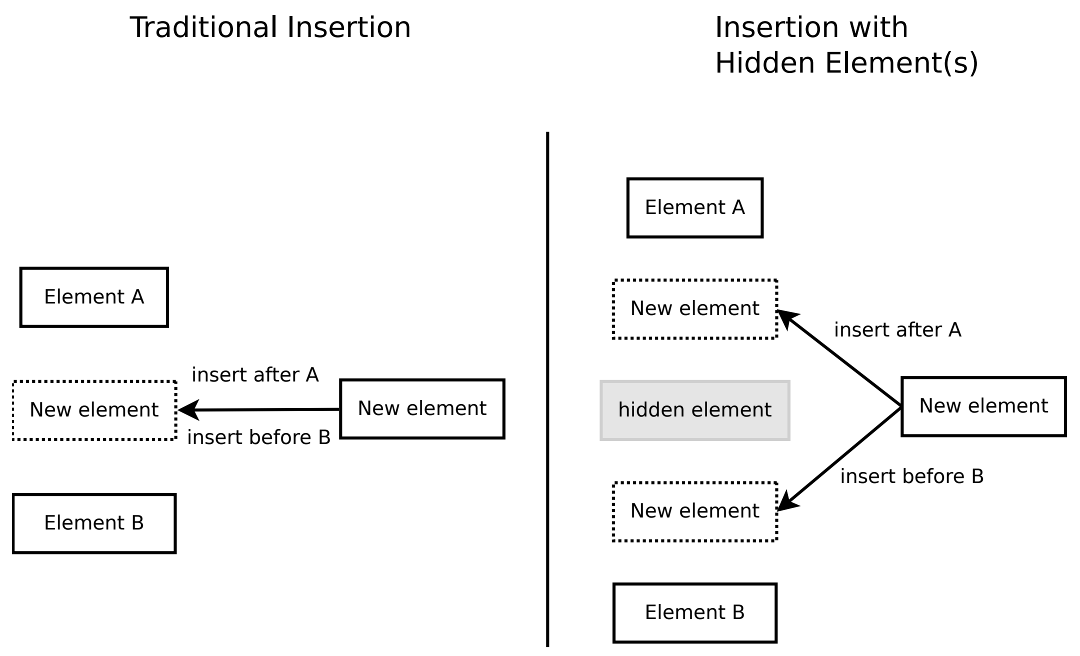


Figure 3.4: ***Inserting elements by position in the presence of hidden elements.*** Generally, insertion before one element and after its immediate predecessor are considered equivalent operations. We see here that this is not the case in the presence of hidden content.

3.2.4 Interactive code elements

Finally, our third major change to the IPython Notebook application is support for interactive code elements. As we described in Section 3.1.2, interactive code elements are code elements with associated GUI controls which can change values within the code and rerun the computations without requiring users to edit the code themselves. This allows authors to build documents where readers, or the authors themselves, can quickly explore the effect that changing a particular parameter or variable value has on the output generated by a code element. The reader can then execute the remainder of the notebook to update later results which depend on the result of evaluating the element's code.[6]

Support for these interactive code elements requires three major components: a way to declare interactivity for a code element, machinery to display the specified GUI controls, and a way of allowing those controls to alter and then rerun the code in the element on command.

We center our GUI control infrastructure for interactive code elements around modifying the notebook author's code within the elements in response to the reader's actions. For example, when the reader moves a slider to alter the bandwidth used in a kernel smoother, an expression within the author's code might be changed from `bw <- 5` to `bw <- 10`. This is suboptimal in some ways, but it allows us to work within IPython Notebook's existing framework. IPython Notebook is designed to pass the exact contents of a code element directly to the evaluation engine when the element is being executed, so modifying that code is the least disruptive way of altering the computations that will be performed.

One advantage of directly changing the code being evaluated is that it allows us to separate the logic which defines the GUI controls themselves from the code within the element. When readers view a notebook, then, they see the *exact code which generates the output that follows it*. The code within the notebook is more directly and parsimoniously associated with the analysis itself.

We limit our changes to single lines that contain simple assignments, that is expressions of the form `var = value` or `var <- value`. When we use the associated GUI control, we construct a string containing the expression for a new simple assignment using the variable associated with the control and the value selected by the user, e.g., `var = <selected value>` – with `<selected value>` replaced with the value chosen by the reader – and replace the existing expression within the code element. After this replacement, the code element is executed as normal

[6]It would be possible to have GUI controls always run all code depending on the current selection, but this goes against a major design decision of IPython Notebook, which essentially models cells as distinct and independent from a code execution perspective.

to generate the output requested by the user.

The restriction to only controlling direct assignments is quite limiting; it would be much more convenient to be able to control constant values directly within complex expressions or function calls such as `rnorm(10)`. Such expressions can easily be refactored by the author, however, to involve variables who receive their values via simple assignment and are then used within the complex expression:

```
n = 10
rnorm(n)
```

We declare interactive controls for a code element within its metadata. Instead of a single value, however, we must provide enough information to create the interactive control. We do this via a simple, expedient, and extendable widget abstraction. We characterize interactive controls for code elements via five pieces of information:

1. type of control (slider, dropdown menu, etc);
2. variable that the control will modify;
3. line number of assignment to modify within the code;
4. default or current value of the variable;
5. any extra information required by the chosen type of control (e.g., min, max, and step for a slider or the labels and values for a drop-down menu).

Widget descriptions associated with code elements are transformed into GUI controls within the page via widget constructors. We provide constructors for the common ”slider” , ”dropdown menu” , and ”textbox” widget types. Users can also create their own widget constructors which either implement new widget types or override our default constructors. The list of available constructors is declared via an associative array in the `widget_constructors` property of the

IPython.IntCodeCell object prototype, and can be added to or modified via standard IPython Notebook JavaScript extensions.

Widget constructors perform three actions. First, they create the HTML elements for the UI controls themselves. Secondly, they insert these UI elements into the HTML content representing the notebook element in the page. Finally, they attach an event handler which calls the notebook element's `doControl()` method with three pieces of extra information when the viewer uses the control to select a new value. These three pieces of information are: the name of the variable whose value the user is selecting, the line number to be modified, and an optional JavaScript function to preprocess the HTML control's raw value before constructing the new assignment expression.

The `doControl()` method on the JavaScript object representing an interactive code element performs the work of actually modifying the code element's content and executing the element. It retrieves the current value from the HTML control, and calls the specified preprocessing function, if any. With this done, `doControl()` constructs the new assignment expression and replaces the specified line in the element with it. Finally, the element is executed as normal.

Suppose we want a slider which creates assignments of *integer* values for the chosen variable, rather than generic *numeric* values. We define a new type of widget, which we will name ”intslider” . The constructor will be nearly identical to the slider constructor with two exceptions. We will throw an error if the minimum, maximum, or step values for the slider are not integers, and we specify a preprocessing function which appends an "L" to the value. Thus `doControl()` will generate, e.g., the expression `k = 5L`, specifying an integral value of `k`, rather than `k = 5`.

Notebook authors create interactive code elements by converting the type of an element from "Code" to "Interactive Code" via the conversion menu in the main IPython Notebook toolbar. This conversion displays a dialog box which prompts the user for the information necessary to define the widget, pictured in Figure 3.5. With this information, the application creates and connects the GUI control via the appropriate widget constructor.

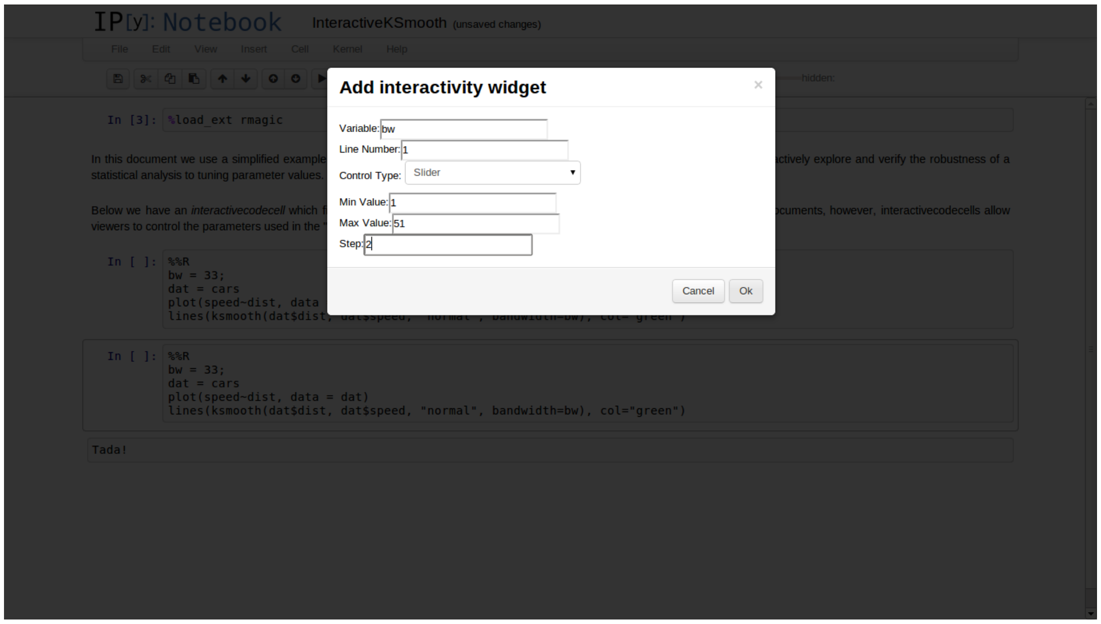
Finally, interactive code elements are intended to provide a (near) real-time experience for the user. UI elements can be highly frustrating when there is a noticeable delay between user action and the display of the new results. We provide a mechanism to cache and quickly restore the result(s) of evaluating R code during execution of code elements.

R code within code elements is executed via the %%R magic defined by IPython, which evaluates the code via Gautier's rpy2 (Gautier, 2012) interface between Python and R. We provide a new %%Rcaching magic, which performs the same function except that the code is evaluated with caching via our **RCacheSuite** package, allowing GUI controls to be used with code that is expensive to run.[7] We leave further description of **RCacheSuite**'s behavior to Chapter 4, where we discuss the package in detail.

3.3 Lessons from our Case Study

We now consider the successes of, and lessons learned from, our case study where we modified the IPython Notebook system. We do not claim that our specific solutions are optimal, or these exact changes could be directly applied to other linear document systems. The problems we addressed with our changes, however, are likely to be present whenever support for non-linear documents is being implemented in an authoring or exploration environment.

[7]The first time the code is run with each possible value for the control will still take the full time to execute.



Variable: bw
Line Number: 1
Control Type: Slider
Min Value: 1
Max Value: 51
Step: 2

```
In [3]: %load_ext rmagic
```

In this document we use a simplified example … ctively explore and verify the robustness of a statistical analysis to tuning parameter values.

Below we have an *interactivecodecell* which f… cuments, however, interactivecodecells allow viewers to control the parameters used in the "…

```
In [ ]: %%R
        bw = 33;
        dat = cars
        plot(speed~dist, data
        lines(ksmooth(dat$dist, dat$speed, "normal", bandwidth=bw), col="green")

In [ ]: %%R
        bw = 33;
        dat = cars
        plot(speed~dist, data = dat)
        lines(ksmooth(dat$dist, dat$speed, "normal", bandwidth=bw), col="green")

Tada!
```

Figure 3.5: ***Adding interactivity to a code element.*** A dialogue box collects the widget description information from the author when he or she converts a code element into an interactive code element. Our modified IPython Notebook then uses this information to identify and invoke the correct widget constructor. In this image we are creating a slider for the `bw` variable defined in line 1 of the element being converted (counting starts at 0 in JavaScript), and which ranges from `1` to `51` by steps of two.

We first discuss the changes we required to IPython's computational model in the context of the more general issues they address. We then discuss the drawbacks and limitations of our resulting modified IPython Notebook application. Finally, we draw on our experience designing and using an environment which combines authoring and exploration to discuss the pros and cons of such two-in-one applications versus designing two separate specialized tools.

3.3.1 Changes to the computational model

We found three major assumptions made by IPython Notebook's computational model to be incompatible with non-linear documents, which we list below. Developers seeking to implement systems for non-linear dynamic documents will need to avoid these assumptions, or modify machinery to remove them when adapting existing systems which make the assumptions.

1. The definition (and display) of a notebook as a flat ordered list of atomic code and text[8] elements with no additional structure.
2. The assumption that a single concept of "next element" is appropriate for both navigation and execution.
3. The assumption that the set of all elements in the notebook and the set of elements currently visible to the user are always identical.

The issue with the assumption that notebooks are flat, sequential lists of code and text elements with no additional structure is clear. We will not belabor it here. There are many ways to model non-linear documents. We chose to model non-linear structures via nesting. We saw in Section 3.2.2 that this strategy required only relatively narrow changes to IPython Notebook's machinery, though we make no comparisons to other strategies which we did not attempt.

The assumption that execution and navigation will have the same concept of "next element" arises from the fact that linear notebooks have only one full thread. When executing elements from a notebook in sequence, we are traversing a thread through the document. When navigating between elements without executing, we are traversing the notebook with the expectation that we can visit every element in the notebook by navigating in only one "direction". With linear documents, traversing the thread through the notebook will visit each element. This allows systems for linear documents to have a single concept of next and previous.

With documents containing decisions, however, executing a thread will not involve all elements within the document. Thus we need two separate concepts of next and previous: one which travels through the entire notebook, and one which travels along a single currently active thread through the document.

[8]IPython Notebook also has title/header elements equivalent to the `<h1>`-`<h5>` HTML tags, but these are simply specialized text elements with specific formatting.

To support element-level navigation in non-linear documents developers must do one of two things. They can define two different traversal methods as we did in Section 3.2.2. Alternatively, they can abandon the assumption that starting at the first element in the notebook and navigating continually in one direction will eventually reach each currently displayed element.

Finally, supporting the concept of multiple levels of detail requires that we have the concept of hidden or deactivated elements which cannot be seen or directly navigated to by the user. The ability to hide elements without removing them allows us to place detail content in the right position within the document, and simply leave it dormant until it becomes visible.

The presence of hidden elements requires us to have two separate concepts of the notebook, however. One, the true or full notebook, knows about all elements, even hidden ones. This full notebook concept is used for loading, saving, and insertion of elements into the document, as we discussed in Section 3.2.3. The other concept of the notebook involves only the elements which are currently displayed, and is used when navigating through and executing elements within the document. These two concepts will not be the same if any elements in the true notebook are hidden, and thus developers wishing to support detail level or similar mechanisms must abandon the assumption that all elements will be active and displayed at all times.

Some additional work is also required to support interactive code elements. This, in general, will consist of three components: a system to describe the interactivity desired, a way to associate interactivity with a specific element, and a way to alter the computations performed based on user feedback. We have chosen to have our controls alter the code within the element itself, preserving the concept that the code contained by an element when it is executed is identical to the code which generated the output inserted into the page.

We will now address the limitations of our modified IPython Notebook platform. Following this we discuss the pros and cons of combining authoring and exploration environments we have experienced when writing and using our system.

3.3.2 Drawbacks of our modified IPython Notebook

We have used our modified IPython Notebook system to successfully author and explore multiple non-linear dynamic documents, including notebooks representing both example analyses shown in Chapter 1. While we found the process convenient in some ways compared to the currently available alternatives, there were non-trivial inconveniences as well. We give a brief, frank discussion of these shortcomings in the remainder of this section.

The largest weakness of our modified IPython Notebook system is that cut, paste, and conversion operations happen at the element level. This means that when an analyst looks back and realizes that the code they have been writing should be organized into a task, moving that existing content into the task is less convenient than we would like.

Secondly, our system is limited in the types of more complex navigation it allows users to perform. Individual alternatives can be activated or deactivated, but selecting threads at the document level, e.g., via a visualization of the structure of the notebook, is not currently supported. A custom front end which could be developed which offers this behavior, but the core IPython Notebook front-end application focuses on an individual cell-centric computational model and as such does not lend itself to extensions in this direction.

Support for more advanced transformations, such as rendering multiple threads side-by-side, is also lacking. This can be achieved by saving the notebook and processing it via **DynDocModel**, but GUI support for at least common use-cases such as direct thread comparison is desirable and not present.

In fact, we have no direct interface between IPython Notebook and **DynDocModel** at all, beyond saving dynamic document files and passing them between the systems. This can cause cumbersome workflows if an analyst is doing more complex manipulations or meta-visualizations in **DynDocModel** during the authoring process.

Editing metadata on elements in the notebook is also cumbersome. With the exception of adding an interactive widget to a code element, notebook authors must use an element-level toolbar provided by the core IPython Notebook application to manually enter or modify the metadata in raw JavaScript Object Storage Notation (JSON). This is inconvenient, and could be streamlined greatly with a better metadata authoring tool for common fields.

Finally, our layout for displaying alternatives side-by-side within a decision element is problematic in the face of many levels of nesting or large numbers of alternatives in a single decision. In these cases, the elements do not have enough room within the page, and thus are not displayed correctly. We have provided the ability to hide all but the currently selected alternative. Showing only one alternative fixes the problem of the content not displaying correctly, but also limits the usefulness of the exploration aspect of our system somewhat.

3.3.3 Separate versus 'both-in-one' authoring and interactive exploration environments

IPython Notebook – both the original and consequently modified version – offers a single system for creating and exploring non-linear notebooks, rather than two separate, more specialized systems. We found both advantages and significant downsides to our implementation of this single application approach, which we discuss in the remainder of this section.

When using our system to build documents, we noticed that some exploration actions are central to our writing process. When adding detail elements, for example, we found ourselves repeatedly changing the displayed detail level to show and hide the elements. This allowed us to ensure that the elements were in the right position in the entire document. Repeatedly changing the detail level also allowed us to confirm that the notebook flowed properly, both programmatically and narratively, at all different detail levels.

Furthermore, when authoring code in dynamic documents in general, it is very convenient to be able to run the code to test it within the context of the document. When writing code within non-linear notebooks, however, this testing requires us to run it in the context of all possible threads which include that code. To this end, we found the ability to select alternatives and step through threads executing the code without leaving the authoring environment highly efficient and effective.

Finally, we found the ability to have functional interactive code elements within our authoring environment to be beneficial as well. The primary purpose of interactive code elements is generally to provide interactivity for the reader, but our experience suggests that having interactive code elements function during the analysis as we create the document served two valuable purposes. First, it allowed us, as authors, to see the interactivity as it would be presented to the reader, ensuring that it worked correctly and was effective in its message. Secondly, it allowed us as analysts to explore certain aspects of the analysis more conveniently as we performed our research than if we had needed to manually change the code expressions ourselves. We were able to do the same type of interactive parameter space exploration available to the reader and to base actual analytic decisions or strategies on any insights we gained. This extends a major focus of the original IPython Notebook, which was to allow interactive and iterative computational research.

We found our modified IPython Notebook system to be less useful for pure exploration. Specifically, we found two relatively major deficiencies. One weakness of our system as an exploration platform is that it does not support advanced transformations or views of the notebook, such as displaying two threads and their results side-by-side. Furthermore, there is currently no mechanism in our modified IPython Notebook to do computations on the document and its structure, e.g., our plot of the different achieved misclassification rates for all the different threads through our digit classification example from Chapter 1. We can perform these types of computations in R with **DynDocModel**, but it would be convenient to be able to perform common tasks within the GUI itself.

Finally, IPython Notebook, both the original and our modified version, displays code in a way designed for programming. The code and text content in the notebook are always editable. This blurs the creation and reading use-cases. The ability to edit content as one views it is useful when carrying out an analysis. However, it can distract from the experience of reading the notebook, even when the reader is able to alter the computations via interactive code elements.

Overall, we consider our modified IPython Notebook a success as an authoring platform for non-linear documents. We were able to create multiple complex notebooks via the system and found it more convenient to do so than constructing ipynb or XML (.Rdb) files via raw text editing. Furthermore, we identified some general issues with supporting non-linear dynamic documents which we think are likely to generalize to other projects.

As an exploration and content delivery platform, however, our system is less successful. Interactive code elements and basic navigation through the document are functional, but the lack of more advanced navigation and viewing mechanisms is a serious limitation. Furthermore, the platform doesn't act as a reading platform, because it allows viewers to edit code and text. This issue is inherited from the core IPython Notebook application which is primarily focused on interactive computational work. Ultimately, we feel the field would benefit from either separate authoring and viewing platforms, or substantially more exploration features being added to our modified IPython Notebook system.

3.4 Background and Related Work

The concept of authoring tools and exploration environments for R-based linear dynamic documents is relatively well established. Emacs Speaks Statistics (Rossini et al., 2001), a mode for the open source Emacs editor, and the RStudio IDE (RStudio Team, 2012) both offer the ability to run code within Rnoweb or Rmarkdown documents as the analyst is creating them. Furthermore, RStudio provides a convenient way to preview a rendered, or *woven*, HTML version of the narrative without leaving the IDE. This preview functionality can be seen as the combination of exploration of the woven report with RStudio's authoring capabilities. RStudio is also investigating a type of interactive notebook powered by their Shiny technology, though currently this is in relatively early stages (Xie, 2013b).

Zhang's *tkWidgets* R package (Zhang, 2013) in the Bioconductor project provides a *vignette explorer* GUI. This interface uses Gentry and Gentleman's *DynDoc* package (Gentleman and Gentry, 2013) to allow users to explore the vignettes of R packages installed on the local system. This exploration includes viewing the structure of the document (a list of code and text chunks in the .Rnw file) as well as evaluating code chunks individually on command.

Nolan and Temple Lang's **XDynDocs** R package offers some support for non-linear documents (Nolan and Temple Lang, 2013). **XDynDoc**'s XML-based format, RDocBook, supports storing non-linear documents via the same type of nesting paradigm we use in **DynDocModel** and our extension of IPython's ipynb format. Furthermore, Nolan and Temple Lang offer some limited capabilities for rendering full documents in the form of XSL files which render decisions by displaying all of their alternatives within a tabbed HTML control.

Nolan and Temple Lang have also investigated the concept of interactive code elements with their **IDynDocs** package (Nolan and Temple Lang, 2007). In particular, they argue for the use of interactive code elements and similar technologies for teaching. As with **XDynDocs**, **IDynDocs** provides a great deal of the inspiration for our work.

3.5 Possible Alternative Approaches for Exploration Environments

Here we briefly discuss possible alternative approaches to creating authoring and exploration environments. Because we have not implemented these, we will not attempt to discuss any of them in detail. Instead, we will describe how they might work, and what aspects of the authoring or exploration processes they might facilitate.

Our **RBrowserPlugin** software embeds a local instance of the R interpreter within many modern Web browsers, allowing Web pages to invoke new R-based computations at viewing time(Becker and Temple Lang, 2013). **RBrowserPlugin** could be used in conjunction with a dynamic document processing system such as our **DynDocModel** or Nolan and Temple Lang's **XDynDocs** to generate HTML pages which display the full structure and rendered content of a non-linear document. These pages would contain JavaScript instructions associated with switching between threads/alternatives which would automatically perform the necessary computations via **RBrowserPlugin**.

HTML pages which show the full document structure could also be generated which invoke R computations when navigating between alternatives or threads using server-based technologies, such as RStudio's Shiny (RStudio Inc., 2014), Urbanek's **fastRWeb** (Urbanek and Horner, 2012) or one of many others. The differences between these approaches and **RBrowserPlugin** are out of scope here. It suffices to say that, again, the pages would allow the user to navigate through the page, selecting alternatives or threads and having the output displayed throughout the page reflect those changes.

Finally, and most ambitiously, a GUI front end could be built for our **DynDocModel** or Nolan and Temple Lang's **XDynDocs** (Nolan and Temple Lang, 2013). These packages have programmatic support for querying and computing on the documents to varying degrees, and so supporting a subset of these more advanced features in the GUI would be relatively straightforward.

3.6 Conclusion

Authoring and exploration tools are needed before non-linear documents are likely to be widely adopted. We presented a modified version of the IPython Notebook (Pérez et al., 2013a) which facilitates the authoring and interactive exploration of these documents. As an authoring tool, the system allows analysts to capture the research process more completely and to capture it as it is being performed. As an exploration tool, it allows readers to navigate though a document, selecting alternatives and re-running computations to reflect the currently chosen thread. This allows viewers to get a more complete concept of the research that was done and the statistical decisions made by the analyst.

In addition to providing usable software, however, we have presented our work as a case study for modifying linear document systems more generally. Our experience has granted us some insight into the types of challenges unique to non-linear documents. Specifically, we saw that issues of location and navigation order in non-linear document platforms are much more complex than their linear document counterparts. While in general this is fairly obvious, we have provided specific discussion of both the types of machinery and abstractions which break down when moving from linear to non-linear documents, and a set of specific changes which achieved non-linear document support in our case.

Chapter 4

Caching for Non-Linear Dynamic Documents

The concept behind *caching* is simple: we can store and reuse the results of expensive computations instead of repeatedly performing those computations in contexts where we can assume the results will be identical. The success of a caching mechanism, then, hinges on its ability to predict when results will be identical and thus can be loaded from a cache.

Determining whether to use a cached result is particularly challenging in the case of non-linear dynamic documents. Evaluating different threads through the same document commonly involves evaluating the same code expressions with different sets of input values. For example, we would expect the output when evaluating code which utilizes variables defined within a decision – i.e. within the code contained in each alternative of the decision – to differ depending on which alternative was previously evaluated. Each thread selects only one branch within the decision. Thus when processing two or more threads we cannot safely assume that the outcome from evaluating a code expression will be identical in each thread, even if the expression itself is identical within all threads.

We saw an example of different threads generating different results for the same code element in Section 1.2.5 of Chapter 1, where we plotted the overall misclassification rate achieved by all five strategies encompassed by the non-linear dynamic document we used in that example. The code element which calculated the misclassification rate was identical across all five threads, but evaluating each thread assigned a different value to the variable representing the classifier itself.

The values input to a code element can also vary during interactive evaluation, even in cases where the document has not changed. In this situation, evaluation is often being performed only on a subset of the document, or even an individual element, rather than the document itself. Typically, a code element's inputs will be created by evaluating other code elements within the document, but this need not be the case. For example, an author might quickly create a mock object[1] in order to test his or her code element without running a full thread up to that element. Furthermore, a researcher might create an external extension to a dynamic document, which takes a subset of the existing document and appends new content to it.

Existing R-based caching mechanisms are not designed to support this type of multiplicity of input state. Their machinery is targeted at weaving linear dynamic documents where each code element in the document can only ever use the input values generated by running the code elements in the document up to that point. We propose a caching mechanism based on the values of the variables the code uses at evaluation time. We have implemented this concept in the **RCacheSuite** package which provides a flexible, input-value aware caching mechanism for evaluating R code.

4.1 Caching

At their core, caching systems consist of two components: a *storage method* capable of saving and later retrieving the result of a set of computations, and a *decision function* which determines whether the result of evaluating a set of expressions is guaranteed to be identical to any available

[1]An object representative of the input values the code expects, but smaller, simpler, and easier to use.

cached values. Matching cached values are loaded directly into the environment, by-passing evaluation of the code. The decision function can use arbitrary logic to make this determination, so long as it does not misidentify cached results as matches which would be different from the result of evaluating the code in the current context.

The crux of a caching mechanism, then, is in the information accepted and logic applied by its decision function. We propose a decision function which takes into account the values of the variables required to evaluate the code in question.

4.1.1 An input-value based caching mechanism

The purpose of the decision function is to determine whether we can safely use cached results in lieu of evaluating a given set of code expressions. We can do so when the cached value is identical to the result we would receive via evaluation.

An *input variable* is a variable whose existence prior to evaluation is assumed by a code. In other words, input variables which are used within an expression without first being defined within that expression. We provide some example expressions and identify the corresponding input variables in Table 4.1.

In most cases, repeated evaluations of a code expression with the same values of its input variables will always generate the same result. Specifically, identical results can be perfectly predicted by looking at input variable values for R code expressions that meet four criteria:

1. Any external data used by the code is unchanged between evaluations,
2. Behavior of R and any package-provided functions remains unchanged,
3. Options and other state not captured by explicit variables do not affect the result,
4. No (pseudo)random numbers are used.[2]

[2]This requirement can be relaxed by considering the seed state to be an implicit input variable as pursued by Xie.(Xie, 2013a)

Table 4.1: R code expressions and their input variables

Expression	**Input variables**
`x = rnorm(n, sd = sd)`	`n`, `sd`
`mylist[["x"]] = 1:10`	`mylist`
`xyplot(y ~ x, data = mydata)`	`mydata`

Some R code expressions and which of the variables they use constitute input variables. Note that in the call to `xyplot()` `x` *and* `y` *are assumed to be columns in the data frame* `mydata`.

We propose a simple decision mechanism in which both the expression and the values of all input variables must match between a cache and the current evaluation context for a cached value to be used. For our discussion here we assume the input variables of each code expression are known. In practice we use Temple Lang et. al's **CodeDepends** (Temple Lang et al., 2013) package to detect these automatically.[3]

We compare the expressions and input values for the prospective evaluation and a given existing cache via two *keys*. *Keys* are strings of characters that are assumed to be the same only if the content used to generate them is identical. One key represents the text of the code expression, while the other is a hash of the collected values of all input variables for the expression. We discuss the efficient generation of these keys from potentially large R objects – the input values – in Section 4.1.5.

By comparing the input values specifically, our decision mechanism obviates the need for any complex dependency tracking between code expressions. Under our assumptions, the only way earlier code can affect a code expression's result is by causing changes to one or more of its input values. These changes will be automatically detected and correctly handled via the value inspection. Furthermore, changes to code will only trigger re-evaluation of later code when the inputs of the later code have *actually* changed. This is in contrast to existing systems, which

[3]**CodeDepends** extends Tierney's similar **codetools** (Tierney, 2011) package in ways that make it more immediately useful for our current task.

implement multi-level dependency tracking and must re-evaluate a piece of code whenever an earlier piece of code which *could* affect its inputs changes.

Though we are assuming that the result of evaluating a piece of code is determined entirely by its input values, code text, and metadata values, the mapping of input values to results need not be one-to-one. A concrete example of this is that the mean, median, and standard deviation of a data vector (and many other meaningful statistics) are independent of data order.[4] In addition any calculation involving, e.g., the absolute value, sine/cosine/etc., floor, positive even exponent, or binning of an input variable can generate the same output from different values of that input.

We know of no way to detect this type of many-to-one mapping via general static code analysis, so both our system and existing ones must re-evaluate such code when its inputs have and could have changed, respectively. Moving forward from that point, however, our system will be aware of whether the results have actually changed, whereas the pure static code analysis approach will not. This means that we can prevent unnecessary propagation of re-evaluation, while existing general systems cannot.

This prevention of cascading re-evaluation allows our system to safely load cached results in a strictly greater number of situations than code text dependency-based systems. We discuss a concrete example of this in some detail in Section 4.1.3.

Our system's direct reliance on the input values themselves has strong advantages in contexts other than traditional weaving. Input values are necessarily available in all situations where code is to be evaluated. This allows our system to be applied directly in situations such as finer-grained evaluation of code elements within a document and interactive exploration of a document. We discuss this situation in more detail in Section 4.1.4.

[4]Barring finite precision arithmetic issues in the cases of mean and standard deviation

4.1.2 Caching non-assignment effects

Tracking and reproducing the return value and any values assigned to variables during evaluation is not sufficient to fully replicate the behavior of some code expressions. Code can have a number of *non-assignment effects* (NAEs) when evaluated, including but not limited to: printing output to the console, opening and drawing to graphics devices, throwing warnings or errors, loading libraries, and returning a value not assigned to a variable by the code.[5]

Our model allows for the storage and reproduction of NAEs by caching a secondary representation of the evaluation result – the *result object* – rather than simply storing the assigned and returned value(s).[6] These result objects are generated by an *evaluator* function, and processed – whether they were generated fresh or loaded from cache – by a *return-handler* function. During this processing, the return-handler recreates any supported NAEs. We illustrate the full caching mechanism in Figure 4.1, and discuss the details of these functions below. Furthermore, both the evaluator and return-handler functions are customizable by the user, allowing developers to experiment with new NAE capturing strategies without being required to build an entirely new caching mechanism or implement substantial changes to an existing one.

By default result objects contain information about any assignments performed during the evaluation, the final return value it generated, and any plots the code generated (or modified), but there are no explicit limits on the form, class, or contents of a result object, other than that it can be processed by a particular return-handler. Because different evaluator functions can return different result objects with all other things being equal, the evaluator function is treated as an implicit input variable for the purposes of locating matching caches (the decision algorithm we described in Section 4.1.1).

[5]We mean assignments within the code, not assignments of the return value of evaluating the code.

[6]In essence, this is what Xie's **knitr** does via its use of Wickham's **evaluate** package, but the user has little control over what is stored or how it is processed when being loaded from cache.

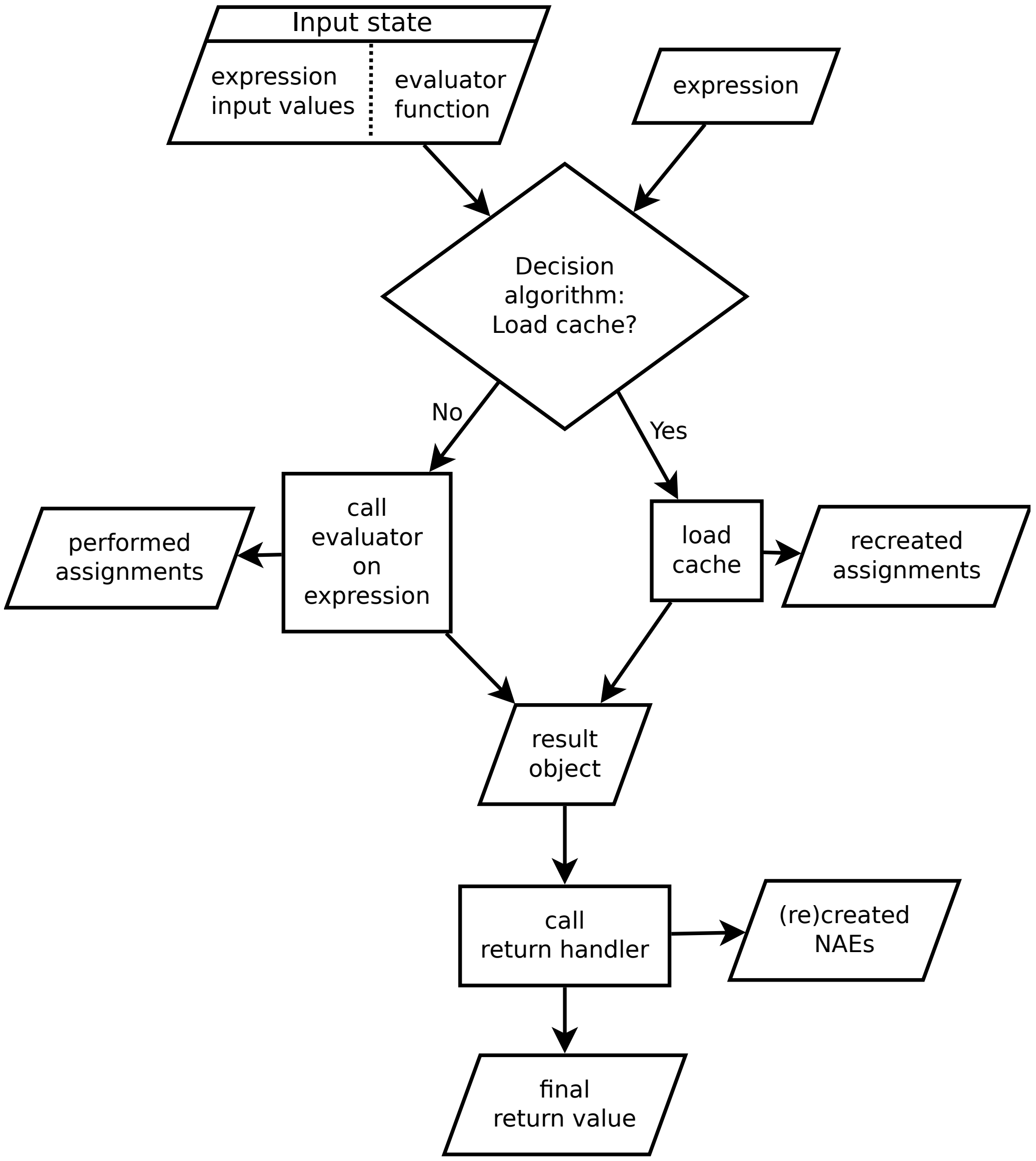


Figure 4.1: ***The computational model of our caching system.*** Our computational model is designed around a secondary representation of the results of evaluation – the *result object* – which is generated by an *evaluator* and processed by a *return-handler*, both of which are customizable by the user. These functions evaluate code expressions in a way that captures information about NAEs, and use that information to recreate the NAEs, respectively. The captured NAE information is stored within the cache alongside the final return value, allowing us to regenerate NAEs even when loading cached results.

The return-handler uses the result object to recreate all supported NAEs before processing and returning the final output value. This occurs regardless of whether the result object was obtained by calling an evaluator or loading an existing cache. We present a simple example of a custom evaluator and return-handler pair below.

Suppose we want to cache global or graphical options in R, as set by `options()` and `par()`, respectively. We note that while we use this example to illustrate the flexibility of our system, care should be taken when setting options without explicit instructions from the user.

In order to add caching support for these types of options, we can simply create a new pair of evaluator and return-handler functions, shown in Figure 4.2. These functions add a layer of options handling around the default evaluation (Wickham's `evaluate()`) and NAE recreation mechanisms (our `evaluate_handler()`). We do this by inspecting the current options before and after evaluating `code` and including the new values of any changed options in the result object within the evaluator. The return-handler then sets options based on this information. Because the return-handler is called after either generating or loading the result object, the new values of the options will be set even when loading from cache.

```
eval_opts = function(code, env, ...) {
    pre_ops = options()
    def_res = evaluate(code, env, ...)
    post_ops = options()
    list(evaluate_ret = def_res,
         opts = option_changes(pre_ops, post_ops) )
}

handler_opts = function(res_obj, ...) {
    do.call(options, res_obj$opts)
    evaluate_handler(res_obj$evaluate_ret)
}
```

Figure 4.2: ***Custom evaluator and return-handler functions.*** These functions allow caching of global option changes. The `eval_opts()` compares the current options before and after code evaluation and includes any differences in the result object, while `handler_opts()` sets any options altered by the code to their new values before continuing with default processing.

When we designed our system there was no existing way to evaluate code and have access to both the raw return value generated during evaluation and information about any plotting, console output, errors, and warnings which occurred. Since then, however, changes have been made to Wickham's **evaluate** package which allow the return of raw final values instead of emulated printed output. With this change, we are able to use `evaluate()` as our core evaluation mechanism, which allows caching of plots, console output, errors, and warnings. This rendered the customizability of our system less necessary, though we feel that there is still value in the flexibility offered by our approach. As such we have left the mechanism unchanged, other than leveraging **evaluate** by default, as it is described above.

We now present a detailed comparison of our approach – as implemented in our **RCacheSuite** package – with two existing systems. We do this in the context of a concrete example which illustrates a case where our system is able to (correctly) load a result from cache where existing systems do not.

4.1.3 A comparison of caching mechanisms

Existing dynamic document caching systems such as *weaver* (Falcon, 2013) and **knitr**(Xie, 2013a) implement decision mechanisms based on a proposal by Leisch to track changes in dynamic document files as a proxy for possible changes in results.

Specifically, Leisch notes that there is an inherent dependency graph among the code elements in a dynamic document based on their order and the variables they use, define, and modify. This dependency graph allows him to categorize the situations where a code element needs to be re-run during the weaving process as follows: "*[i]n general, each node has to be re-evaluated only if any ancestral node in the graph [of dependencies among code elements] or the node itself changes. Otherwise cached results from previous computations can be re-used.*"[7](Leisch, 2003)

[7]We emphasize that he is referring to the graph of dependencies among code elements within a flat dynamic document, rather than the implicit document graph we defined in Chapter 2.

Table 4.2: Criteria for loading from an existing cache

weaver	**knitr**	**RCacheSuite**
1. Code expression matches 2. Code expressions which generated input variables match 3. No expressions from (2) were rerun	1. Full code element matches: (a) code text (b) computed chunk option values 2. No cached dependencies have changed	1. Code expression matches 2. Current values of input variables match those previously used

The criteria for loading an existing cache differs substantially between our ***RCacheSuite*** *package and the existing weaver and* ***knitr*** *packages.*

Both our system and these existing systems track and identify stored results via keys. *Keys* are unique values, typically strings, which encode information about the computations which generated the result such that computations assigned the same key are assumed to give the same result. These keys are then used during the process of deciding whether to re-run a piece of code. The exact manner of the decision is slightly different in each case, as are the keys used. These differences embody the differences between the systems themselves, as illustrated by the examples in this section.

Falcon's *weaver* and our **RCacheSuite** both operate at the individual expression level, while the caching mechanism in Xie's **knitr** operates on entire code elements (which often encompass many individual expressions). The criteria for using a cached result in these three systems are listed in Table 4.2.

Consider the snippet of an Rnoweb document pictured below. It contains three expressions involving three variables (`foo`, `bar`, and `done`).

```
<<b1 cache=TRUE>>=
bar <- 10
foo <- bar^2 * 3
done <- foo + 5
@
```

When any caching system is applied in the absence of pre-existing caches, it evaluates all pieces of code and creates the relevant caches. For the sake of brevity, we assume this has already occurred for our code element via both *weaver* and **RCacheSuite**. We now discuss the behavior of these three systems when the first expression in the element is changed to `bar <- -10` to illustrate the differences between Falcon's decision mechanism and our own. Figure 4.3 illustrates the behavior of the two systems in this situation.

We now discuss the two algorithms in detail within the context of the example described above. Checking the expression which generated each input variable only handles direct dependencies. Dependency under the Leisch/Falcon model is transitive. That is, a piece of code depends on all of the dependencies of each of its dependencies, and on each of their dependencies, and so on. We see this in our example code element in the form of the implicit dependency of the `done <- foo + 5` expression on the expression `bar <- 10` by way of `foo <- bar^2 * 3`. In order to handle these implicit dependencies without actually constructing Leisch's dependency graph, Falcon adds a check for whether any of the direct dependencies needed to be re-evaluated. If so, it is assumed that there is a mismatch somewhere farther up the stream and the cache is declared stale (not usable).

As we saw in Figure 4.3, each of our three expressions fails to meet a different one of *weaver*'s three criteria for loading an existing cache, and thus all three expressions are re-evaluated. The first expression has actually changed, and thus no cache exists for it at all. The second expression has a dependency mismatch, in that previously the expression which generated `bar` was `bar <- 10` whereas now it is `bar <- -10`. Finally the third expression has a dependency (foo being generated by `foo <- bar^2 * 3`) which matches but was re-evaluated due to an earlier change.

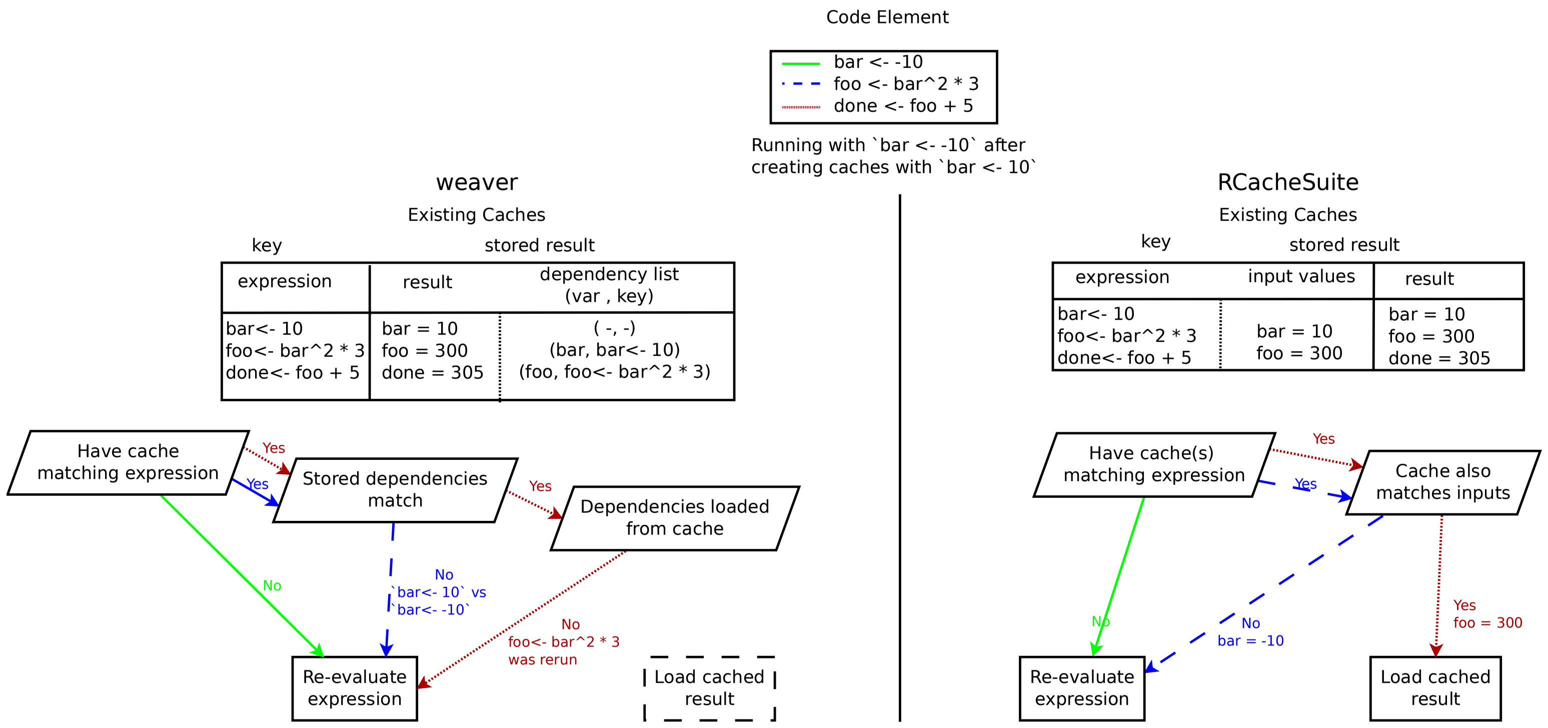


Figure 4.3: ***A side-by-side comparison of the weaver and RCacheSuite decision algorithms.*** Given the set of three code expressions (top) and existing **weaver**- and **RCacheSuite**-style caches for them (middle-left and middle-right, respectively), we compare the behavior of our caching mechanism (lower-right) to that of Falcon's **weaver** (lower-left). After altering the first expression in a way that does not affect the result of the second (from `bar <- 10` to `bar <- -10`), **RCacheSuite** loads the cached result for the third expression, while **weaver** must re-evaluate all three. Paths through the algorithms for each expression coded by color and line type.

Our system, on the other hand, is able to load the existing cached result for the third expression. The reason for this is that the change in the first expression does not cause a change in the value of the variable `foo` present when the third expression is considered.

As above, **RCacheSuite** re-evaluates the first expression because none of the existing caches match the new expression. This results in a value of `-10` for the variable `bar` when considering the second expression. The second expression remains unchanged, but the value of the `bar` input variable does not match. Thus the cache does not match and the second expression is re-evaluated, as it was by *weaver*.

The re-evaluation of the second expression, however, generates a value of `300` for the variable `foo` – the same value generated when `bar` was `10` – due to the fact that `bar` appears only as a squared term in the expression. Thus, when **RCacheSuite** considers the third expression, both the expression and its input value match, allowing our system to load the cached result and avoid the redundant evaluation performed by *weaver*.

As stated above, Xie's **knitr** operates at the level of entire code elements. This means that our change to one of the expressions within the element causes the single cache for the element to not match. Thus it will re-evaluate all three expressions during the course of re-evaluating the code element itself. If the three expressions are split up into three separate code elements, the end result is the same as in *weaver*: all three expressions are re-evaluated.

In our example, only one subsequent expression had `foo` as an input variable and thus the effective difference between the two systems was not particularly large. In general, however, all expressions which use `foo` or any variables whose value depends on `foo` will be loaded from cache by our system but recomputed by *weaver*. In lengthy analyses, this can amount to a substantial difference in total computations performed by the two systems.

4.1.4 Caching in non-linear and interactive contexts

Our primary purpose for pursuing an input-based decision mechanism is to extend caching for dynamic documents into interactive contexts and non-linear documents. Both of these situations require the ability to retain more than one cache per expression, because code run before a particular element, and thus the set of input values present when evaluating, is not unique. Our system inherently supports this due to the fact that cache storage and matching depends on both the expression and the current values of all input variables.

When weaving reports from non-linear dynamic documents, the evaluation history, and thus dependencies of the current expression, at any point in the process depends on which thread through the document is being woven. This means that a dependency mismatch between an existing cache and the current state for a particular expression[8] is not evidence that a change to the underlying document has been made which renders the cache permanently invalid, as is the case with flat documents.

Assumptions about element dependencies break down even further when operating on dynamic documents interactively. In this situation, the unit of evaluation is typically individual code elements, which can often be run by the user repeatedly and in any order. This may be via a graphical user interface (GUI) as in Gentleman and Gentry's *DynDoc* and the IPython Notebook platform – both Granger and Perez et al's original and our non-linear extension discussed in Chapter 3, but can also be done directly via interactive scripting as in Temple Lang et. al's **CodeDepends**[9] and our **DynDocModel**.

Furthermore, in non-GUI interactive work, the values of input variables for an expression can

[8]Or the entire code element.

[9]The CodeDepends package does not define a dynamic document system per se, but it provides mechanisms for reading scripts in various formats into a dynamic document-like structure containing a series of script (code) elements and provides methods for evaluating the structure both as a whole and in parts.

be modified directly using code which does not appear in the document at all. For example, an analyst might interactively modify an intermediate value to debug later code that uses it – or more generally to explore the later code's behavior given corner-case inputs – without needing to modify the full document to create a thread which generates that specific value.

Because the decision algorithm we propose in Section 4.1.1 allows us to avoid explicit dependency calculations, we are able to completely bypass the complications of determining dependencies without a fixed, or even finite selection of possible evaluation histories. Furthermore, because input values are sufficient to detect cache mismatches based on changes to the dependency chain, we do so without introducing any chance of false positives.

Existing systems which use only calculated dependencies and code text, on the other hand, would require substantial changes to their underlying assumptions about dependency structure in order to be useful in non-linear and interactive contexts. Unlike existing systems, however, we must fully process all input values. Some of these values may be many times larger and more complex than representations of the code, making the generation of keys potentially much more expensive than in existing systems. We present our approach for efficiently generating keys from large input values in the next section.

4.1.5 Efficient comparison of large R objects via hashing

One of the goals of our caching system is to facilitate interactive code elements. For interactive code elements to be fully effective the GUI controls must be responsive to actions by the user. The entire process of finding a matching cache and loading the data, then, must occur as quickly as possibly, with a strong preference for near real-time performance. We explore this issue with an illustrative, though admittedly contrived example. We note that this example is not intended to be realistic, but rather is a minimal reproduction of the issue in order to showcase our solution.

Suppose we have a large numeric vector containing 125 million elements – about 1 gigabyte of data – in R and we wish to investigate the distribution of its elements. We want to create an interactive code element which plots a histogram of the values, with the number of bins in the histogram controlled by a slider GUI element.

To achieve a satisfactory user experience we must generate a key for the data vector, compare it to existing caches until we find a match or exhaust the set of existing caches, and – if a matching cache is found – load the result with minimal delays. We will focus here on the generation of the key, which is a substantial bottleneck when taking the naive approach of serializing the object into a character vector then calling a hash function on the vector.

With data this size, constructing the input-value portion of the key to compare with existing caches can itself be expensive. We need a way of generating these keys which is as close to instantaneous as possible while still maintaining the requirement that the same key will be generated for different data with extremely low probability. For reference, Eddelbuettel et al.'s **digest** function (Eddelbuettel et al., 2013) takes on the order of 7 seconds to generate a key from an object this size on our laptop [10], primarily due to the fact that it copies the data. Furthermore, calling `identical()` on two R vectors of this size with the same data, but which are stored in different objects internally, takes approximately 2.7 seconds on the same machine.

Our **fastdigest** package takes 0.19 seconds to generate a key for our approximately 1 gigabyte numeric vector. This is nearly 40 times faster than `digest()` for one such vector and about 7 times faster than `identical()` for two such vectors when the data are the same but are stored in separate objects internally.[11]

fastdigest provides a streaming-hash option for R's own internal serialization framework. The

[10]machine and software specs are provided in Appendix B.

[11]`identical()` searches until it finds a difference, so if a difference happens early in the vector it would be faster than **fastdigest** which always processes the entire object.

bulk of the serialization logic in R is implemented in the R_Serialize() C function, which accepts an R object and an output target. We call this function with a custom streaming-hash target so that instead of writing the serialized data to a file, string, or raw vector, the data is fed directly into the SpookyHash algorithm. No duplication of the data is ever done, as the hashing algorithm is able to process the same copy of the data in memory that constitutes the contents of the actual R object.

By using R's own mechanism, we are guaranteed that the data being passed to the hashing algorithm uniquely describes the R object. We are also able to provide direct access to the hook functionality R's `serialize()` function provides for handling reference style objects such as *externalptr* objects and *connections*. We simply accept a hook function of the same specification expected by `serialize()` and pass it directly to the R_Serialize() C function, where it performs its typical role.

We use Jenkins' SpookyHash hashing function(Jenkins, 2012), though the same approach could be used for any hashing function which can accept streaming data. SpookyHash generates 128-bit hash values which can be compared to infer whether the two data sources used to generate the hashes were the same. The algorithm is deterministic, meaning identical data will always generate the same key. Thus two pieces of data which have different hashes are guaranteed to be different. Two pieces of data which generated the same hash can be distinct, but this occurs with a very small probability. Specifically, because the hash values are 128-bit, the expected number of such hash value collisions is approximately 1 collision per 2^{128} unique, randomly distributed pieces of data.[12](Jenkins, 2012)

Jenkins' algorithm is also extremely fast, achieving rates of 3 bytes of data processed *per processor cycle* (Jenkins, 2012), the maximum read rate for non-cached memory on the machine, or the speed at which the data is being passed to it, whichever is smaller. For perspective, a mod-

[12]Due to hardware constraints, the algorithm has only been empirically tested to 2^{73} unique pieces of data.

ern processor core performs billions of cycles per second (1 gigahertz is approximately 1 billion cycles/second).

In practice, the speed of `fastdigest()` is limited by the throughput of data from the R object being passed to Jenkins' algorithm. For atomic vectors, we are able to pass the entire vector as a single block, thus approaching the speeds Jenkins cites. For lists and other non-atomic structures, however, each element is processed separately by R_Serialize() and thus a list with a few large vector elements will be processed much more efficiently than a list with the same total amount of data split into many small elements. Even in this case `fastdigest()` outperforms `digest()` by about a factor of two – 0.5 seconds vs 1.12 seconds for a list with 5 million scalar elements.

Our `fastdigest()` function is able to generate keys much faster than the operation of reading a cached result of equal size from disk. Thus we consider our bottleneck resolved. We now discuss some drawbacks of our general caching approach and how we seek to mitigate them.

4.2 Known Limitations and Drawbacks

The most significant weakness of our system is that it requires the ability to compare input values for all input variables across different evaluation environments or even across R sessions. We do this via hashing based on R's internal serialization machinery. That machinery, however, cannot handle certain types of R objects – e.g. *connections* and objects based around external pointers – without specific instructions from the user. Absent such instructions, R will generate non-unique serializations for these objects, e.g. by recording the address of all *externalptr* objects as `0`. This causes distinct evaluation contexts to generate the same key at a much higher rate than we assume based on SpookyHash's collision rate when the input values include these types of objects.

We mitigate this issue in the same manner that R does: by allowing users to specify custom handlers for serializing, e.g., *connections* and *externalptr* objects. In fact, as with standard serialization, the exact same machinery is used when calling these custom handlers within `fastdigest()` as when users call the `serialize()` function directly. We admit that this is somewhat cumbersome, but feel that matching R's own approach to serializing these objects is preferable to implementing a different workaround.

Another downside to our input value based decision mechanism is that it provides no obvious mechanism for detecting when a cache is no longer needed. In the traditional caching systems offered by *weaver* and **knitr**, any time a cache with a matching key is not loaded due to dependency mismatches, it is assumed that the document being woven has permanently changed. As such, the cache is considered permanently invalidated and is typically overwritten with a new cache after the next re-evaluation of the corresponding code. While this causes problems in the non-linear and interactive contexts we discussed in Section 4.1.4, it also provides significant, though not perfect, garbage collection for unnecessary caches in code-text-based systems.

We currently have no automatic system for detecting and removing unneeded caches. In fact, in the context of arbitrary interactive evaluation, caches can never be guaranteed not to match any future evaluation context. This does not mean, of course, that caches should never be deleted, but heuristics and active decision making by the user are required in our system where other systems are able to automate the process.

To facilitate the identification of caches which may not be worth keeping, we track date and usage-frequency information for each cache. This allows users to set garbage collection mechanisms based on date created, most recent use, or frequency of use which match the specific needs of their application. We also allow the restriction of how many caches can be maintained for a single expression. Restricting the number of caches per expression to one effectively simplifies our system to those offered by existing packages, including their implicit cleanup mechanism.

Finally, while not specific to our system, caching at the expression level instead of the larger code elements involves a potentially substantial trade-off. Expression level caching is much more precise in determining which computations must be performed than its less granular counterpart, but it can, in certain situations, result in many times more total space being required to cache the result of evaluating a given piece of code. Consider the simple pseudo-code in Figure 4.4 describing three data-cleaning steps we might apply to a dataset.

```
<<datacleaning>>=
replace non-positive values of variable 1 with NAs
replace values greater than 10 of variable 2 with NAs
detect and fix typos in categorical variable
@
```

Figure 4.4: A simple (pseudo-code) data-cleaning code element

Given a code element with multiple expressions modifying the same object – e.g. a *data.frame* representing the data being analyzed – an element-level caching mechanism such as the one used in **knitr** will create only one cache storing the final version of the object after all three cleaning steps. With expression level caching, however, three separate caches are made, each of which contains a full copy of the data after the respective cleaning step is performed. This causes the total cache size for our code element to be three times as large when cached by expression than by code element. This can have significant and even prohibitive consequences when caching is applied to code which makes repeated or incremental changes to large objects or datasets.

We address this issue by allowing users to specify that a code element should be cached as a single expression.[13] We provide the user three mechanisms for doing this: via specific metadata on the code element, via an explicit argument in the call to `evalWithCache()`, or by providing an R function which accepts the full contents of the code element and returns a logical indicating

[13]In practice this is done by wrapping the full code block in curly braces.

whether it should be treated as a single entity. We also provide a default heuristic which causes a code element to be cached as a single unit if all expressions within it set or update the same variable.

We have presented a proposal to use direct input value inspection during caching in order to protect against changes to previously evaluated code in a way that extends naturally to contexts where this evaluation history is not unique. This allowed us to seamlessly support weaving multiple threads through the same non-linear document, as well as the interactive exploration and piecewise evaluation of documents. We also briefly discussed an implementation of this proposal in our **RCacheSuite** package, and how its behavior compares to existing caching mechanisms for dynamic document systems in R. Finally, we discussed the drawbacks of our system and our efforts to mitigate them.

Chapter 5

Summary

Our work has focused on implementing, exploring and extending ideas proposed by Gentleman and Temple Lang (Gentleman et al., 2004) and Nolan and Temple Lang (Nolan and Temple Lang, 2013), which we have incorporated into our concept of *comprehensive dynamic documents*. Our contributions roughly cluster into three arenas, each associated with a software tool we have developed:

- We have designed and implemented computational models for representing and processing linear and non-linear dynamic documents in R (**DynDocModel**) and a general mechanism for querying the contents of arbitrarily complex R objects (**rpath**).
- We have created an interactive authoring and exploration environment which supports non-linear dynamic documents and discussed the design lessons learned during its creation.
- We have developed and implemented a caching strategy for use when multiple threads are being evaluated through a comprehensive document.

We now summarize our contributions in each of these arenas, as presented in the preceding chapters of this dissertation.

5.1 DynDocModel: A Flexible, Unified System for Dynamic Documents in R

In Chapter 2 we presented our **DynDocModel** R package. **DynDocModel** contributes two primary advances to dynamic documents in R. First, we implement a format agnostic object model for linear and non-linear dynamic documents. Secondly, we utilize a highly modularized and customizable computational model for processing dynamic documents.

All but the most straightforward data analyses will involve decisions by the analyst amongst multiple alternatives. No existing dynamic document system for R, however, provides standardized, out-of-the-box processing for non-linear dynamic documents[1]. **DynDocModel** provides high- and low-level APIs for handling non-linear dynamic documents. This ensures the author and audience can generate reports and compute directly on comprehensive research documents.

Linear dynamic documents are currently used primarily to generate woven reports. We meet this need by ensuring that linear narrative reports can be generated from comprehensive and non-linear documents. Computing on the documents directly, on the other hand, opens up new use-cases to both analysts and readers which can incorporate structure, content, and results of the documents to synthesize new information about the research process or results. We saw this in the plot of misclassification rates for all the possible final classifiers in our digit analysis (Section 1.2.5), which displays information not contained by any single thread through the document[2].

We have defined a four step computational model for processing dynamic documents: constructing or population the document object, subsetting the document, evaluating code, and rendering content (including formatting output). Furthermore, we have designed **DynDocModel** so that each of these four steps is fully customizable by the user. Customizability at each step of

[1]Though Nolan and Temple Lang's .Rdb format does provide a mechanism for storing non-linear documents, and XPath can be used to manually construct the set of content to process.

[2]Unless one considers a linearization of the full body of code a single thread.

our computational model grants us specific capabilities relevant to dynamic documents; we briefly recount our motivating examples and the concepts they showcase below.

By enabling customization of the parsing mechanism, we allow **DynDocModel** to – in principal – support arbitrary, future dynamic document file formats. This serves to future-proof any advances we or **DynDocModel** users make in the arena of processing or computing on dynamic documents.

User control of the subsetting mechanism allows them to specify which thread or non-thread set of content is being processed. This concept is central to our comprehensive documents, as it allows the user to declare which narrative he or she would like to process.

Customizing the evaluation of code when processing a document – or thread therein – allows users to affect the output included within a woven report without changing the code in the document itself. We saw an example of this where we used custom evaluation to automatically add timing information for each code element in the processed thread. More generally, we can add provenance or other information for the results presented in a report. This information could be a function of the *processing mechanism*, the code being evaluated, or a combination of the two.

Furthermore, the benefits of a separate, customizable rendering step of content (rendering) and results (formatting) is two-fold. First, the fact that it is a separate step from evaluation allows us to render a single evaluated thread into multiple different output files of different formats without any re-evaluation of the content. Secondly, by customizing the insertion of output – R objects – into a report, users can create rich, context-specific formatting for their reports. We saw this in our rendering example where we displayed the confusion matrix within our woven HTML report as a styled `<table>` with the counts for the most common error for each type of digit highlighted.

We now move on to summarize our work modifying the IPython Notebook (Pérez et al., 2013a) computing platform. This work constitutes our second area of contributions: software to create and interactively display non-linear dynamic documents.

5.2 The Modified IPython Notebook: A Non-Linear Authoring and Computing Platform

Our modified IPython Notebook platform – presented in Chapter 3 – provides document authors with the ability to create non-linear and comprehensive dynamic documents. Our contributions stemming from this software are twofold. The software itself provides a proof of concept tool capable of generating the type of non-linear document **DynDocModel** is designed to process. Secondly, the process of creating the software acted as a case-study for the types of challenges and design considerations inherent in supporting non-linear documents.

Our modified IPython Notebook platform allows users to add non-linearity in the form of tasks, decisions, and multiple resolutions (detail levels) to IPython notebooks. This allows analysts to create comprehensive documents as they are proceeding through their research. The platform also provides a way to display and explore non-linear documents, both those generated within it, and those constructed elsewhere and converted to non-linear ipynb files via our **DynDocModel** package.

The process of modifying the original IPython Notebook application allowed us to explore some of the implications of allowing users to create and explore non-linear notebooks from a design perspective. We found that a number of concepts which are straightforward for linear documents require more care when handling non-linear ones. When faced with these differences, we both addressed them for our specific software and attempted to generalize where appropriate into insights applicable to the development of other non-linear document creation and exploration tools.

A major difference between linear and non-linear documents is that the order of navigation through the document and the order of code execution when running the document are identical for linear documents but differ for non-linear documents. The reason for this is that when referring to 'running' a non-linear document, we almost always actually mean *running a single thread within the document*. Thus, the execution order describes a linear traversal of a subset of the document, while the navigation order would describe a non-linear traversal of the entire document. These concepts of ordering were further complicated when we introduced the multiple resolution/detail level concept, which calls for elements to be contained in the document but not displayed or visited during navigation. We found that other, less central details of the software that involved order or position were also affected by our changes, such as insertion of elements into the document when undisplayed content is present.

Next we summarize our third avenue of research, which involved efficiency when processing dynamic documents. More specifically, we designed and implemented a caching mechanism which supports the same code – from the same code element – generating different results. We see this behavior both when evaluating multiple threads through the same document and in displaying interactive code elements.

5.3 RCacheSuite and FastDigest: a Caching Solution for Non-Linear Dynamic Documents

Finally, we presented our work towards facilitating interactive use of non-linear dynamic documents via caching in Chapter 4. We proposed and implemented a new caching mechanism based on comparing the values of input variables before code evaluation to those present when a cached result was generated.

By distinguishing caches based on the input values used to generate their results, rather than solely on code text, we are able to support caching in the context of evaluating multiple threads through a single non-linear document. Furthermore, when these threads contain the same code element, that code can be associated with multiple caches, one for each distinct set of inputs.

Caching multiple results for the same code is crucial for interactively navigating between threads. Requiring full re-evaluation of a thread each time we wish to display or load its associated results is infeasible when the code in question is computationally expensive. Furthermore, inspecting input values allows our system to safely load cached results in situations where existing systems – which look only at the code in a document – cannot.

A major obstacle to employing our strategy was the cost of comparing current input values with those associated with existing caches (which we do via hashing). For large values this can be expensive enough to render interactive use infeasible, even if the comparison remains orders of magnitude faster than re-performing the computations. To mitigate this we developed the **fastdigest** package, which leverages Jenkins' *SpookyHash* to implement a streaming hash target for R's built-in serialization machinery.

Implementing a streaming hash target for R's serialization machinery allows us to avoid storing a full, serialized copy of the object being hashed. This lack of copying – as well as using Jenkins' impressively fast 128-bit algorithm – lead to a 40x speed-up compared to existing approaches when the value consists of relatively few large vectors. We were also able to leverage R's own handling of external pointers and environments during serialization. Though potentially still more costly than a code-text-only comparison scheme, **fastdigest** makes hashing large R objects – and thus generating hash keys based on input values – an order of magnitude faster than loading a cached object of equivalent size from an .rda file.

5.4 Availability

The software tools I implemented during this work are publicly available on github at `https://www.github.com/gmbecker`. In addition, I intend to publish the three R packages discussed herein (**DynDocModel**, **RCacheSuite**, and **fastdigest**) on the CRAN package repository system. My work modifying IPython Notebook, however, is unlikely to find its way into the core IPython software. I will continue to make it publicly available via github until such a time as it warrants a more formal release.

5.5 Concluding Remarks

We have implemented software tools which address three parts of a proposed comprehensive non-linear document workflow. Our modified version of the IPython Notebook application allows analysts to interactively create, view and execute non-linear notebooks. The **DynDocModel** package provides a flexible, customizable, and format-agnostic object model for representing dynamic documents in R, including non-linear notebooks created in the modified IPython Notebook platform. Finally, we implemented a caching mechanism which supports the type of multi-path evaluation of dynamic documents necessary to navigate and explore a comprehensive document in real time, e.g., within our IPython Notebook platform.

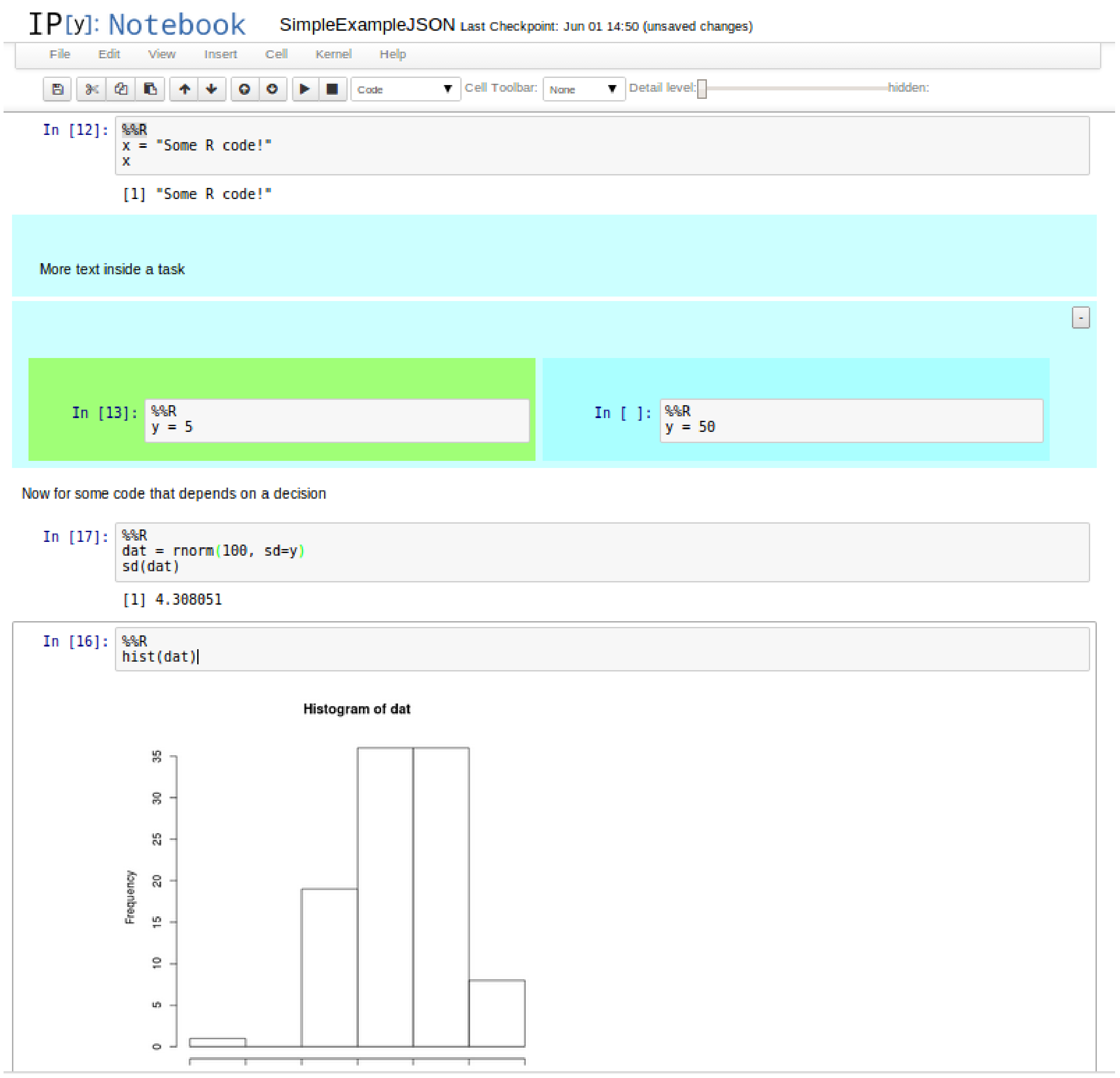
Throughout this process, we explored both the specific steps necessary to implement our systems, and the more general design problems which face developers building tools which deal with non-linear documents. We feel our work both stands as a proof of concept for implementing such tools, and provides a set of guidelines and lessons learned for future developers in the non-linear document space. Finally, our examples throughout this document illustrate how these concepts can inform the ways we record data analyses, and transform those records into articles, reports, and interactive representations of the analyst’s research process.

Appendix A

An excerpt of a non-linear IPython Notebook .ipynb file

Here we present the JSON source for a very simple and slightly abridged non-linear ipynb file to illustrate our extension of the linear ipynb format. Image data has been removed for clarity and brevity, but would appear in this JSON object as a quoted string. We present a screenshot of the notebook in question for context, followed by the JSON string.

We have extended the JSON ipynb format to allow cells to be nested (i.e., certain cell types can contain arrays of child cells), but have left the format specification otherwise unchanged. We also provide structural element types – tasks (*task* cells), decisions (*altset* cells) and alternatives (*alt* cells) – which use nesting to implement the types of non-linear structure we describe in Chapter 2. Expressions within the JSON displayed below which represent cells of these types are shown in a bolded, blue font for identification and emphasis.


IP[y]: Notebook
SimpleExampleJSON Last Checkpoint: Jun 01 14:50 (unsaved changes)
File Edit View Insert Cell Kernel Help
Code
Cell Toolbar: None
Detail level:
hidden:
Histogram of dat
Frequency
0
5
10
15
20
25
30
35

```
In [12]: %%R
         x = "Some R code!"
         x

         [1] "Some R code!"
```

More text inside a task

```
In [13]: %%R
         y = 5

In [ ]: %%R
        y = 50
```

Now for some code that depends on a decision

```
In [17]: %%R
         dat = rnorm(100, sd=y)
         sd(dat)

         [1] 4.308051

In [16]: %%R
         hist(dat)
```

Figure A.1: A very simple non-linear IPython notebook

Expressions which utilize our extension of the ipynb JSON format

```
{
 "metadata": {
  "name": ""
 },
 "nbformat": 3,
 "nbformat_minor": 0,
 "worksheets": [
{
 "cells": [
  {
   "cell_type": "code",
   "collapsed": false,
   "input": [
    "%load_ext rmagic"
   ],
   "language": "python",
   "metadata": {},
   "outputs": []
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "Here is some Markdown text"
   ]
  },
  {
   "cell_type": "code",
   "collapsed": false,
   "input": [
    "%%R\n",
    "x = \"Some R code!\"\n",
    "x"
   ],
   "language": "python",
   "metadata": {},
   "outputs": []
  },
  {
   "cell_type": "task",
   "cells": [
    {
     "cell_type": "markdown",
     "metadata": {},
     "source": "More text inside a task"
    }
   ],
   "metadata": {}
  },
  {
   "cell_type": "altset",
   "cells": [
   {
    "cell_type": "alt",
    "cells": [
     {
      "cell_type": "code",
      "collapsed": false,
      "input": "%%R\ny = 5",
      "language": "python",
      "metadata": {},
      "outputs": [],
      "prompt_number": 4
     }
    ],
    "metadata": {},
    "most_recent": true
   },
   {
    "cell_type": "alt",
    "cells": [
     {
      "cell_type": "code",
      "collapsed": false,
      "input": "%%R\ny = 50",
      "language": "python",
      "metadata": {},
      "outputs": []
     }
    ],
    "metadata": {},
    "most_recent": false
   }
  ],
  "metadata": {}
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "Now for some code that depends on a decision"
  ]
 },
 {
  "cell_type": "code",
  "collapsed": false,
  "input": [
   "%%R\n",
   "dat = rnorm(100, sd=y)\n"
 ],
 "language": "python",
 "metadata": {},
 "outputs": [],
 "prompt_number": 7
},
{
 "cell_type": "code",
 "collapsed": true,
 "input": [
  "%%R\n",
  "sd(dat)"
 ],
 "language": "python",
 "metadata": {},
 "outputs": [
  {
   "metadata": {},
   "output_type": "display_data",
   "text": [
    "[1] 3.675473\n"
   ]
  }
 ],
 "prompt_number": 9
},
{
 "cell_type": "code",
 "collapsed": false,
 "input": [
  "%%R\n",
  "hist(dat)"
 ],
 "language": "python",
 "metadata": {},
 "outputs": [
  {
   "metadata": {},
   "output_type": "display_data",
   "png": *base 64 encoded PNG image*
  }
 ],
 "prompt_number": 10
},
{
 "cell_type": "code",
     "collapsed": false,
     "input": [],
     "language": "python",
     "metadata": {},
     "outputs": []
    }
   ],
   "metadata": {}
  }
 ]
}
```

Appendix B

Machine information and benchmarking code for digest comparison

Benchmark comparisons between `fastdigest()` and `digest()` performed on an Intel-based laptop with 4 gigabytes of RAM and the Core i3 370 M quad-core processor clocked at 2.4GHZ.

The following code was used to perform the benchmarks:

```
library(fastdigest)
library(microbenchmark)
library(digest)

x = rnorm(125000000)
fdvec = microbenchmark(fastdigest(x))
dvec = microbenchmark(digest(x), times=10)library(fastdigest)

rm(x)
gc()
y = as.list(rnorm(5000000))
fdlist = microbenchmark(fastdigest(y), times=10)
dlist = microbenchmark(digest(y), times=10)
```

We used versions 1.3-0, 0.6-4 and 0.5-0 of the **microbenchmark**, **digest**, and **fastdigest** packages, respectively. Full session info was as follows:

```r
sessionInfo()
```

```
R version 3.1.0 (2014-04-10)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8
 [2] LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8
 [6] LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8
 [8] LC_NAME=C
 [9] LC_ADDRESS=C
[10] LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8
[12] LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils
[5] datasets  methods   base

other attached packages:
[1] microbenchmark_1.3-0 digest_0.6.4
[3] fastdigest_0.5-0

loaded via a namespace (and not attached):
[1] compiler_3.1.0 tools_3.1.0
```

Bibliography

ActivePapers Development Team (2013). ActivePapers: python edition. `https://github.com/activepapers/activepapers-python`.

Allaire, J., Horner, J., Marti, V., and Porte, N. (2014). *markdown: Markdown rendering for R*. R package version 0.7, `http://cran.r-project.org/package=markdown`.

Avila, D. (2013). live_reveal: Project to implement a "LIVE" reveal version for the IPython notebook. `https://github.com/ipython-contrib/live_reveal`.

Baggerly, K. A. and Coombes, K. R. (2009). Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. *The Annals of Applied Statistics*, 3(4):1309–1334.

Bavoil, L., Callahan, S. P., Crossno, P. J., Freire, J., Scheidegger, C. E., Silva, C. T., and Vo, H. T. (2005). Vistrails: Enabling interactive multiple-view visualizations. In *Proceedings of IEEE Visualization*.

Becker, G. (2013). *rpath: XPath-based querying for R objects*. R package version 0.1.0, `https://github.com/gmbecker/rpath`.

Becker, G. and Temple Lang, D. (2013). *RBrowserPlugin: R bindings to NPAPI Browser Plugin*. R package version 0.1-5, `https://github.com/gmbecker/RBrowserPlugin`.

Becker, R. and Chambers, J. M. (1984). *S: An Interactive Environment for Data Analysis and Graphics*. Chapman and Hall/CRC.

Berglund, A. (2006). Extensible Stylesheet Language (XSL) Version 1.1. W3C recommendation, World Wide Web Consortium (W3C). `http://www.w3.org/TR/xsl11`.

Brauer, M., Durusau, P., Edwards, G., Faure, D., Magliery, T., Radius, B., and Vogelheim, D. (2005). Open Document Format for Office Applications (OpenDocument) v1. 0. Technology standard, Organization for the Advancement of Structured Information Standards (OASIS). `https://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf`.

Brazma, A., Hingamp, P., Quackenbush, J., Sherlock, G., Spellman, P., Stoeckert, C., Aach, J., Ansorge, W., Ball, C. A., Causton, H. C., Gaasterland, T., Glenisson, P., Holstege, F. C., Kim, I. F., Markowitz, V., Matese, J. C., Parkinson, H., Robinson, A., Sarkans, U., Schulze-Kremer, S., Stewart, J., Taylor, R., Vilo, J., and Vingron, M. (2001). Minimum information about a microarray experiment (MIAME)-toward standards for microarray data. *Nature genetics*, 29(4):365–71.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Breiman, L., Friedman, J., Stone, C., and Olshen, R. (1984). *Classification and Regression Trees*. Chapman and Hall, New York, NY.

Brodlie, K., Poon, A., Wright, H., Brankin, L., Banecki, G., and Gay, A. (1993). GRASPARC-A problem solving environment integrating computation and visualization. In *Proceedings Visualization '93*, pages 102–109. IEEE Comput. Soc. Press.

Buckheit, J. B. and Donoho, D. L. (1995). WaveLab and Reproducible Research. In Antoniadis,

A. and Oppenheim, G., editors, *Wavelets in Statistics*, Lecture Notes in Statistics, pages 55–82. Springer-Verlag.

Cervone, D. and Krautzberger, P. (2013). Mathjax: Beautiful math in all browsers. `http://www.mathjax.org`.

Chambers, J. (2010). *Software for Data Analysis: Programming with R*. Statistics and Computing. Springer, New York, NY.

Claerbout, J. and Karrenfach, M. (1992). Electronic documents give reproducible research a new meaning. *1992 SEG Annual Meeting*.

Clark, J. (1999). XSL Transformations (XSLT) Version 1.0. W3C recommendation, World Wide Web Consortium (W3C). `http://www.w3.org/TR/xslt`.

Clark, J. and DeRose, S. (2006). XML path language (XPath) Version 1.0. W3C recommendation, World Wide Web Consortium (W3C). `http://www.w3c.org/TR/xpath`.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27.

Eddelbuettel, D., Lucas, A., Tuszynski, J., Bengtsson, H., Urbanek, S., Frasca, M., Lewis, B., Stokely, M., Muehleisen, H., and Murdoch., D. (2013). *digest: Create cryptographic hash digests of R objects*. R package version 0.6.4, `http://cran.r-project.org/package=digest`.

Falcon, S. (2013). *weaver: Tools and extensions for processing Sweave documents*. R package version 1.30.0, `http://www.bioconductor.org/packages/release/bioc/html/weaver.html`.

Free Software Foundation (2013). GNU Emacs. `https://www.gnu.org/software/emacs/`.

Gautier, L. (2012). rpy2: A simple and efficient access to R from Python. Software version 2.30, `http://rpy.sourceforge.net/`.

Gentleman, R. and Gentry, J. (2013). *DynDoc: Dynamic document tools*. R package version 1.42.0, `http://www.bioconductor.org/packages/release/bioc/html/DynDoc.html`.

Gentleman, R. and Temple Lang, D. (2007). Statistical Analyses and Reproducible Research. *Journal of Computational and Graphical Statistics*, 16(1):1–23.

Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y. H., and Zhang, J. (2004). Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10):R80.

Gruber, J. (2004). Markdown. `http://www.daringfireball.net/markdown`.

Herndon, T., Ash, M., and Pollin, R. (2013). Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. *Cambridge Journal of Economics*.

Huntley, M. A., Larson, J. L., Chaivorapol, C., Becker, G., Lawrence, M., Hackney, J. A., and Kaminker, J. S. (2013). ReportingTools: an automated result processing and presentation toolkit for high-throughput genomic analyses. *Bioinformatics (Oxford, England)*, 29(24):3220–1.

Jenkins, B. (2012). SpookyHash: a 128-bit noncryptographic hash. `http://burtleburtle.net/bob/hash/spooky.html`.

Kuhn, M. (2014). *odfWeave: Sweave processing of Open Document Format (ODF) files*. R package version 0.8.4, `http://cran.r-project.org/package=odfWeave`.

Lang, L. and Wolf, H. P. (1996). The REVWEB manual for SPLUS with WINDOWS. `http://www.wiwi.uni-bielefeld.de/fileadmin/emeriti/naeve/manuale.ps` (Defunct), Accessed Nov, 2013.

Lee, J. and Grinstein, G. (1995). An architecture for retaining and analyzing visual explorations of databases. In *Visualization '95, Proceedings of the IEEE Conference on Visualization*, pages 101–108. IEEE Computing Society Press.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In Härdle, W. and Rönz, B., editors, *Compstat 2002 - Proceedings in Computational Statistics*, pages 575–580, Heidelberg. Physica Verlag.

Leisch, F. (2003). Sweave and Beyond: Computations on Text Documents. In Hornik, K., Leisch, F., and Zeileis, A., editors, *Proceedings of the International Conference on Distributed Statistical Computing (DSC-2003)*.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomForest. *R News*, 2(3):18–22. Available at `http://cran.r-project.org/package=randomForest`.

MacFarlane, J. (2006). Pandoc. `http://johnmacfarlane.net/pandoc/`.

Maechler, M. (2010). *classGraph: Construct Graphs of S4 Class Hierarchies*. R package version 0.7-4, `http://CRAN.R-project.org/package=classGraph`.

Marsh, J., Orchard, D., and Veillard, D. (2006). XML Inclusions (XInclude) Version 1.0 (Second Edition). W3C recommendation, World Wide Web Consortium (W3C). `http://www.w3.org/TR/xinclude`.

Neuwirth, E. (2011). *RColorBrewer: ColorBrewer palettes.* R package version 1.0-5, `http://cran.r-project.org/package=RColorBrewer`.

Nolan, D. and Temple Lang, D. (2007). Dynamic, Interactive Documents for Teaching Statistical Practice. *International Statistical Review*, 75(3):295–321.

Nolan, D. and Temple Lang, D. (2013). *XDynDocs: Dynamic Documents with XML and XSL.* R package version 0.3-1, `http://www.omegahat.org/XDynDocs/`.

Nolan, D. and Temple Lang, D. (2014). *XML and Web Technologies for Data Sciences with R*. Use R! Springer, New York, NY.

Pau, G. (2010). *hwriter: HTML Writer - Outputs R objects in HTML format.* R package version 1.3, `http://cran.r-project.org/package=hwriter`.

Pérez, F. and Granger, B. E. (2007). {IP}ython: a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29.

Pérez, F., Granger, B. E., and IPython Core Development Team (2013a). {IP}ython Notebook. `http://ipython.org/notebook.html`.

Pérez, F., Granger, B. E., and IPython Core Development Team (2013b). {IP}ython Notebook Messaging Specification. `http://ipython.org/ipython-doc/stable/development/messaging.html#messaging`.

Python Development Team (2013). Python. `http://www.python.org`.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ram, K. and Temple Lang, D. (2012). *rProv: Provenance tracking in R*. R package version 0.2-0, `https://github.com/karthik/rProvenance`.

Ramsey, N. (1994). Literate programming simplified. *IEEE software*, 11(5):97–105.

Rossini, A., Maechler, M., Hornik, K., Heiberger, R., and Sparapani, R. (2001). Emacs Speaks Statistics: A Universal Interface for Statistical Analysis. *UW Biostatistics Working Paper Series*, Nr. 173.

Rossini, A. J. (2001). Literate Statistical Practice. In Hornik, K. and Leisch, F., editors, *Proceedings of the International Conference on Distributed Statistical Computing (DSC-2001)*.

RStudio Inc. (2014). *shiny: Web Application Framework for R*. R package version 0.9.1, `http://cran.r-project.org/package=shiny`.

RStudio Team (2012). *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA. Available at `http://rstudio.com`.

Santos, E., Lins, L., Ahrens, J., Freire, J., and Silva, C. (2009). Vismashup: Streamlining the creation of custom visualization applications. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1539–46.

Taylor, C. F., Paton, N. W., Lilley, K. S., Binz, P.-A., Julian, R. K., Jones, A. R., Zhu, W., Apweiler, R., Aebersold, R., Deutsch, E. W., Dunn, M. J., Heck, A. J. R., Leitner, A., Macht, M., Mann, M., Martens, L., Neubert, T. A., Patterson, S. D., Ping, P., Seymour, S. L., Souda, P., Tsugita, A., Vandekerckhove, J., Vondriska, T. M., Whitelegge, J. P., Wilkins, M. R., Xenarios, I., Yates, J. R., and Hermjakob, H. (2007). The minimum information about a proteomics experiment (MIAPE). *Nature biotechnology*, 25(8):887–93.

Temple Lang, D. (2011). *Sxslt: R extension for libxslt*. R package version 0.91-1, `http://www.omegahat.org/Sxslt`.

Temple Lang, D. (2013). *XML: Tools for parsing and generating XML within R and S-Plus*. R package version 3.98-1.1, `http://www.omegahat.org/XML`.

Temple Lang, D. and Becker., G. (2013). *RWordXML: Simple tools for Open Office Word Processing XML documents*. R package version 0.1-0, `http://www.omegahat.org/RWordXML`.

Temple Lang, D., Peng, R., and Nolan, D. (2013). *CodeDepends: Analysis of R code for reproducible research and code comprehension*. R package version 0.3-5, `http://www.omegahat.org/CodeDepends`.

Therneau, T., Atkinson, B., and Ripley, B. (2014). *rpart: Recursive Partitioning and Regression Trees*. R package version 4.1-8, `http://cran.r-project.org/package=rpart`.

Tierney, L. (2011). *codetools: Code Analysis Tools for R*. R package version 0.2-8, `http://cran.r-project.org/package=codetools`.

Urbanek, S. and Horner, J. (2012). *FastRWeb: Fast Interactive Framework for Web Scripting Using R*. R package version 1.1-0, `http://cran.r-project.org/package=FastRWeb`.

Walsh, N. and Muellner, L. (1999). *DocBook: The Definitive Guide*. O'Reilly Media.

Warnes, G. R., Bolker, B., Bonebakker, L., Gentleman, R., Liaw, W. H. A., Lumley, T., Maechler, M., Magnusson, A., Moeller, S., Schwartz, M., and Venables, B. (2014). *gplots: Various R programming tools for plotting data*. R package version 2.13.0, `http://cran.r-project.org/package=gplots`.

Watson, G. S. (1964). Smooth Regression Analysis. *Sankhya*, 26(4):359–372.

Wickham, H. (2014). *evaluate: Parsing and evaluation tools that provide more details than the default.* R package version 0.5.5, `http://cran.r-project.org/package=evaluate`.

Wickham, H., Danenberg, P., and Eugster, M. (2014). *roxygen2: In-source documentation for R*. R package version 4.0.1, `http://cran.r-project.org/package=roxygen2`.

Xie, Y. (2013a). knitr: A comprehensive tool for reproducible research in R. In Stodden, V., Leisch, F., and Peng, R. D., editors, *Implementing Reproducible Computational Research*. Chapman and Hall/CRC.

Xie, Y. (2013b). An R notebook in Shiny. `http://glimmer.rstudio.com/ropensci/knitr/`.

Zhang, J. (2013). *tkWidgets: R based tk widgets*. R package version 1.42.0, `http://www.bioconductor.org/packages/release/bioc/html/tkWidgets.html`.