Anonymous Location Based Messaging
The Yakkit Approach

by

Przemyslaw Lach
B.S.Eng., University of Victoria, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Przemyslaw Lach, 2015
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Anonymous Location Based Messaging
The Yakkit Approach

by

Przemyslaw Lach
B.S.Eng., University of Victoria, 2015

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

ABSTRACT

The proliferation of mobile devices has resulted in the creation of an unprecedented amount of context about their users. Furthermore, the era of the Internet of Things (IoT) has begun and it will bring with it even more context and the ability for users to effect their environment through digital means. Applications that exist in the IoT ecosystem must treat context as a first class citizen and use it to simplify what would otherwise be an unmanageable amount of information. This thesis proposes the use of context to build a new class of applications that are focused on enhancing normal human behaviour and moving complexity away from the user. We present Yakkit—a location based messaging application that allows users to communicate with others nearby. The use of context allows Yakkit to be used without the creation of a login or a profile and enhances the normal way one would interact in public. To make Yakkit work we explore different ways of modelling location context and application deployment through experimentation. We model location in an attempt to predict a user's final destination based on their current position and the trajectories of past users. Finally, we experiment deploying the Yakkit service on different servers to observe the effect of distance on the message transit time of Yakkit messages.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements xi

Dedication xii

1 Introduction 1

1.1 Motivation 1
1.2 Problem Statement 3
1.3 Approach 4
1.4 Contributions 5
1.5 Thesis Overview 5

2 Background 6

2.1 Sensors and Mobile Devices 6
2.2 The Internet of Things and Self Adaptive Systems 14
2.3 Cloud Intrastructure 18
2.3.1 Challenges in the Cloud 19
2.3.2 Smart Applications on Virtual Infrastructure 19
2.3.3 Measuring Latency 20
2.4 Software Complexity 21
2.5 Human Communication 22

2.6 Prediction Using Location 23
2.6.1 GeoLife 23
2.6.2 GeoLife Research 24
2.7 Summary 25

3 Location Based Social Networking **26**
3.1 A New Twist on an Old Idea 26
3.2 Yakkit 27
3.3 Yakkit 2.0 31
3.4 Yakkit Challenges and Approaches 34
3.5 Summary 35

4 Data Mining User Trajectories **36**
4.1 When Location Is Not Enough 36
4.2 Experiment Setup 37
4.2.1 Data Preparation 37
4.2.2 What Is a Trajectory? 44
4.2.3 Modelling and Classification 51
4.2.4 The Experiment 54
4.3 Experimental Results 56
4.4 Discussion 59
4.5 Threats to Validity 60
4.6 Summary 61

5 Yakkit Service Deployment and Latency **62**
5.1 As Fast as Possible 62
5.2 Experiment Setup 63
5.3 Experimental Results 65
5.4 Discussion 67
5.5 Threats to Validity 70
5.6 Summary 71

6 Conclusions **72**
6.1 Summary 72
6.2 Contributions 73
6.3 Future Work 73

6.3.1 Sentiment Analysis 73
6.3.2 Modelling Locations 74
6.3.3 Deployment 74

Bibliography **75**

A Source Code **81**
A.1 Bot Source Code 81
A.2 Time Delta Kernel Density Source 90
A.3 Trajectory Distance Kernel Density Source 91
A.4 Trajectory Generation Source 92
A.5 Experiment Source 103

List of Tables

Table 2.1 iPhone 6 Plus Sample Specification 12

Table 4.1 Sector 1,175 Model 54

Table 4.2 Delta in Prediction Success Between Symmetric Original and Shifted Experiments 59

Table 4.3 Delta in Prediction Success Between Asymmetric Original and Shifted Experiments 60

Table 5.1 Experimental Result Summary 67

List of Figures

Figure 2.1 Dr. Martin Cooper, the inventor of the cell phone, with DynaTAC prototype from 1973 in 2007. (Courtesy of Know Your Mobile) . . . 7
Figure 2.2 The MicrTAC released in 1989. Worlds first flip-phone that fit your pocket. (Courtesy of WonderHowTo) . . . 7
Figure 2.3 The IBM Simon released in 1993. The World's first smartphone. (Courtesy of WonderHowTo) . . . 8
Figure 2.4 Clockwise: Nokia 3210 (1999), GeoSentric (1999), Kyocera's Visual Phone (1999), Nokia 9000 Communicator (1997), and Motorola StarTAC (1996). (Courtesy of WonderHowTo) . . . 9
Figure 2.5 Clockwise: Motorola Razor (2004), Microsoft Pocket PC Phone Edition (2002), and Blackberry 5810 (2002). (Courtesy of WonderHowTo) . . . 10
Figure 2.6 iPhone Standard Apps . . . 13
Figure 2.7 Global Device Penetration Per Capita. (Courtesy of Business Insider) . . . 13
Figure 2.8 Internet of Things (Courtesy of Wilgengebroed on Flickr) . . . 15
Figure 2.9 Autonomic Manager (AM) [KC03] [IBM06] . . . 17
Figure 2.10 Autonomic Computing Reference Architecture (ACRA) [IBM06] 18

Figure 3.1 CB Radio Base Station . . . 27
Figure 3.2 Original Yakkit Architecture . . . 28
Figure 3.3 Autonomic Manager for Yakkit Service . . . 29
Figure 3.4 Yakkit iPhone App Interfaces . . . 30
(a) Chat View . . . 30
(b) Billboard View . . . 30
(c) Map View . . . 30
Figure 3.5 Advertising Portal . . . 31

Figure 3.6 Yakkit App 32
(a) Ad Creation 32
(b) Ad Scheduling 32
(a) Chat 32
(b) Ad Presentation 32
Figure 3.7 Yakkit Version 2.0 33

Figure 4.1 GeoLife Data Structure 38
(a) Directory Structure 38
(b) File Structure 38
(c) File Contents 38
Figure 4.2 Imported Dataset Schema 39
Figure 4.3 All GeoLife Points (Scale 1:64,000,000) 40
Figure 4.4 Downtown Beijing (Scale 1:200,095) 41
Figure 4.5 Downtown Beijing with Original Dataset (Scale 1:200,095) . . 42
Figure 4.6 Downtown Beijing with Boundary (Scale 1:200,095) 42
Figure 4.7 Downtown Beijing with Original Dataset and Boundary (Scale 1:200,095) 43
Figure 4.8 Downtown Beijing with Boundary and Filtered Dataset (Scale 1:200,095) 43
Figure 4.9 Updated Schema to Include Boundary and Filtered Points . . 44
Figure 4.10 Relative Kernel Densities of Time Deltas Between Points 99th Percentile 46
Figure 4.11 UTM Zones (Courtesy Wikimedia Commons) 47
Figure 4.12 Updated Schema to Include Trajectory and Sample 48
Figure 4.13 Downtown Beijing Trajectories 49
Figure 4.14 Kernel Density of Trajectory Distances 99th Percentile 50
Figure 4.15 Downtown Beijing Trajectories with Length Less Than 15 km (Scale 1:200,095) 50
Figure 4.16 Downtown Beijing with 6,000 m Sectors (Scale 1:200,095) . . . 51
Figure 4.17 Source Sector 1,175 With 1,000 m Sectors In Background (Scale 1:65,000) 52
Figure 4.18 Source Sector 1,175 Trajectories With 1,000 m Sectors In Background (Scale 1:65,000) 53

Figure 4.19 Source Sector 1,175 Destination Sectors With 1,000 m Sectors In Background (Scale 1:65,000) 53
Figure 4.20 Final Schema 55
Figure 4.21 Symmetric Original and Shifted Experiment Results - Side by side comparison of original and shifted experiments showing the effect of sector boundaries on classification. 56
Figure 4.22 Asymmetric Original and Shifted Experiment Results - Side by side comparison of original and shifted experiments showing the effect of sector boundaries on classification. 57
Figure 4.23 Kernel Density of Number of Sectors Error as a Fraction of Symmetric Sector Size for False Predictions (Original and Shifted) 58
Figure 4.24 Kernel Density of Number of Sectors Error as a Fraction of Asymmetric Sector Size for False Predictions (Original and Shifted) 58

Figure 5.1 Server Locations Latency Experiment 64
Figure 5.2 Message Routing Best Case 66
Figure 5.3 Message Routing Worst Case 66
Figure 5.4 Closest Proximity Experiment Results 68
(a) Oregon to Victoria Ping and Message Transit Results 68
(b) Virginia to Carleton Ping and Message Transit Results 68
Figure 5.5 Farthest Proximity Experiment Results 69
(a) Oregon to Carleton Ping and Message Transit Results 69
(b) Virginia to Victoria Ping and Message Transit Results 69

ACKNOWLEDGEMENTS

If not for the persistence of my supervisor, Dr. Hausi A. Müller, I would have never made the decision to attend grad school. Attending grad school was one of the best decisions I made and Dr. Müller's continued support allowed me to grow on an intellectual and personal level. For that I am forever grateful.

To all those in Rigi Group, in particular Ron Desmarais, I thank you for your friendship and for the time we spent working together. A large part of my academic success is the result of the constructive criticism and thoughtfulness that I have been shown. My work is that much better for it.

DEDICATION

I dedicate this work to my wife Cindy Matthew. Her support at a critical juncture during my undergraduate days paved the way for me to have this opportunity. You can only connect the dots looking back and looking back I am certain I would not have made it this far without her.

Chapter 1

Introduction

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."
—Mark Weiser in The Computer for the 21st Century

Within the last ten years we have seen the emergence of the social network. In the latter part of this decade this has been complemented by the proliferation of mobile devices. This new interaction paradigm affords the opportunity to connect users in ways that are socially familiar and spontaneous. Yet despite this social and technological renaissance people are not necessarily happier nor better off. Mobile devices provide a constant stream of distractions and our social networks require our constant attention. This chapter provides a brief overview of the problem, the motivation, and the solutions proposed in this thesis.

1.1 Motivation

Historically speaking we are living in exceptional times. Mobile devices connect us 24-7 and provide unprecedented computational power in a small form factor. Yet despite this technological marvel, these devices are highly under-utilized. We still predominantly build applications for mobile devices as extensions of their desktop counterparts and as such we limit ourselves to the paradigms of desktop computing.

One incarnation of the mobile device is the smartphone. A smartphone is a combination of hardware and software that serves as both a mobile computing and communication device. The idea behind the smartphone was first introduced by IBMs Simon way back in 1993, but it was not until 2007, with the release of the first iPhone,

that the post-PC era began to take shape [IBM93]. Recent studies have shown large growth in the mobile device market. Cisco predicts that by 2018 mobile device sales will hit the ten billion mark with approximately 1.4 mobile devices per capita.[1] This means that one seventh of the human population will have at least one smartphone in their pocket.

Each smartphone generation features more sensors and richer APIs that provide developers with greater tools to develop rich applications. Current modern smartphones boast hardware spaces such as: 2G, 3G, and 4G antennas, 128GB of storage space, 1920x1080 resolution displays, 8MP cameras, gyroscopes, microphones, GPS, accelerometers, compass, proximity/ambient light sensors, barometers, and 2.26GHz processors. In addition to hardware that was simply not available on a desktop, such as GPS and ambient light sensors, modern day smartphones provide computational power that was only available on a desktop PC just a few years earlier. This trend in hardware will continue as future smartphones will have even more storage, more computational power, and faster network access speeds (i.e., 5G).

In parallel to the evolution of the smartphone, and mobile devices in general, we have seen the evolution of the Internet and the services that run on it. Some services, such as Twitter,[2] owe much of their success to the proliferation of these mobile devices while others, like Facebook,[3] owe much of their ongoing success to it. In either case, these types of services have a mobile counterpart that is used to either provide its users with an optimal mobile experience or take advantage of the rich contextual information that these devices offer.

Sensors, storage capacity, and computational power have resulted in a constant stream of user information, or context, to be generated very quickly. Users' history, such as where they have been, what they have bought, and what they are currently doing is being monitored and recorded by many of these Internet services. These services mine this information for context and use it to personalize the user experience in one form or another. For example, Google reads through your e-mail in order to provide targeted advertising that is based on the keywords in your e-mails.

Although these types of personalization services are still in their infancy and are still unable to offer, what we would consider, a truly personalized experience, they have begun to shape user expectations. Users are increasingly expecting to

[1] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-forecast-qa.html

[2] http://www.twitter.com

[3] http://www.facebook.com

have their experience or content personalized. In addition, the emergence of cloud computing has made it possible for users to access their data anywhere resulting in mobile applications whose architectures span the mobile device and the cloud. Applications that span multiple devices, especially over the Internet, are susceptible to lag which may degrade the experience. As such, additional care must be taken to measure and minimize the latencies in these applications. The context provided by smartphones, the user data stored in the cloud, and the effects of latency on deployment present technical as well as social challenges.

1.2 Problem Statement

Most people who have used a smartphone would agree that they can be very disruptive and that the applications that run on those devices, particularly social applications, can be a big time sink. To some extent this problem is hard wired in us as humans. Research in psychology claims that we spend as much as 30-40% of our speech for the sole purpose of informing others of what we feel and what we have accomplished [TM12]. This innate desire for self disclosure coupled with the ease with which we are now able to do so is part of the problem.

To add to these distractions we live in a world where we have a plethora of choices. If you want to buy a bagel you have a dozen flavours to choose from. If you want to buy a pair of jeans you have to decide which type of cut you want. If you go into the average American supermarket you get to choose from 285 varieties of cookies. Just because some choice is good does not mean that lots of choice is better. Too much choice leads us to have to manage trade-offs and management of trade-offs makes us miserable [Sch09].

The intersection of mobile devices, the cloud, and the shift in user expectations present an opportunity to re-think how we build and deploy applications. In this thesis we aim to answer the following research questions as they relate to this opportunity:

1. What kind of applications can we build when we shift complexity away from the user with the intention of nurturing normal human communication?

2. How can we model anonymous user GPS location context to predict meaningful destinations without direct user intervention?

3. Does deploying location based chat services closer to a user have a positive or negative effect on message latency?

1.3 Approach

The pervasiveness of computers may lead one to think that they have become ubiquitous; however, if we reflect on the quote at the start of this chapter we realize that technology is far from invisible. Ubiquitous technologies are supposed to disappear but current technologies have the opposite effect; they distract. In his paper titled "The Computer for the 21st Century" Mark Weiser, the father of ubiquitous computing, identified three characteristics necessary for a technology to become ubiquitous: (1) inexpensive and low power, (2) ubiquitous software applications and (3) a fast network to connect them all together [Wei95].

Since the publication of Weiser's paper we have made progress in lowering the cost of computing, improving mobility and power consumption, and networking them all via the Internet and the cloud. Although this progress is still far from this ultimate vision of ubiquitous computing, it is a step in the right direction as far as hardware and networking technologies are concerned. The third piece of the puzzle, the ubiquitous software applications, is lagging behind. The intention of this thesis is not tackle challenges in the ubiquitous community *per se* but given the effect software has on people's lives it is high time we as software engineers think more deeply about how our software impacts users.

With that responsibility in mind as well as Weiser's principles for a ubiquitous computing vision we aim to tackle our research questions in the following manner:

1. Develop a context aware and feedback enabled application that uses user location to automatically create social connections with other nearby users.

2. Model the relationship between users' trajectory start and stop positions and see if the model can be used to predict a new users final destination with the goal of personalizing their experience.

3. Deploy a location based chat application on geographically separated servers and determine what effect distance has on message latency.

1.4 Contributions

The contributions of this thesis align with our three research questions and are as follows:

1. A login-less, profile-less, location based messaging framework and application that instantly connects you to those around you.
2. Experimental results that show the accuracy of our location prediction model when trying to predict a user's intended destinations.
3. Experimental results from an emulation that show the effect of distance on message latency.

1.5 Thesis Overview

Chapter 2 presents the necessary background both for motivating our work and for giving the reader a general understanding of the thesis subject domain. In Chapter 3 we present Yakkit as our first contribution and solution to our first research question. Furthermore, we use Yakkit as a platform for our next two contributions in Chapters 4 and 5 where we run experiments and discuss their results with the goal of answering our last two research questions.

Chapter 2

Background

2.1 Sensors and Mobile Devices

Over 40 years ago, in 1973, a Motorola engineer by the name of Martin Cooper made the very first mobile phone call. The number he dialed was that of Motorola's competitor, Bell Labs, and the purpose was to let them know that Motorola had managed to create a mobile phone. Ten years later in 1983 Motorola launched the first commercial version of the mobile phone, dubbed the Motorola DyanTAC 8000X, for a modest price of $3,995. Clearly, this phone was not targeted at the masses and its 30 minute battery life made the four thousand dollar price tag even harder to swallow.

Figure 2.1: Dr. Martin Cooper, the inventor of the cell phone,
with DynaTAC prototype from 1973 in 2007.
(Courtesy of Know Your Mobile)

The same decade also saw the release of two other mobile phones: the Mobira Talkman (1984) and Motorola's MicroTAC (1989). The Mobira allowed for several hours of talk time but its battery was the size of a lunchbox and required a handle for carrying it around. At the end of the decade Motorola's iteration on the DyanTAC produced the MicroTAC. This was considered the world first flip-phone as well as the world's first pocket phone.

Figure 2.2: The MicrTAC released in 1989.
Worlds first flip-phone that fit your pocket.
(Courtesy of WonderHowTo)

The 1990's was the decade that saw the evolution of the mobile phone and the entrance of new players into the mobile phone market. In 1993 IBM released what would become the worlds first smartphone: the IBM Simon. Simon functioned as a pager, fax machine, and personal digital assistant (PDA). Using Simon and its interactive touchscreen you could add appointments to your calendar, search through your address book, and send e-mail. Simon was truly innovative in the sense that back in 1993 it already had several of the features that we find in smartphones today.

Figure 2.3: The IBM Simon released in 1993.
The World's first smartphone.
(Courtesy of WonderHowTo)

During the 90's the mobile phone continued to morph more into a mobile computer rather than just a better mobile phone. Motorola continued to innovate by delivering the first clam-shell phone that used the new 2G network. Nokia entered into the smartphone market as well with their Nokia 9000 Communicator which had the first QWERTY keyboard. Nokia also released their 3000 series of phones which became legendary for their indestructibility. This legendary series included the Nokia 3210 which emerged as one of the most popular phones in history. Nokia was also the first to offer access to a text based version of the Internet via the 7110 phone using the Wireless Applications Protocol (WAP). GeoSentric introduced the first phone that had built in GPS and Kyocera's Visual Phone was the first phone with a built-in camera. By the end of the 90's mobile phones were capable of more than just making phone calls.

Figure 2.4: Clockwise: Nokia 3210 (1999), GeoSentric (1999), Kyocera's Visual Phone (1999), Nokia 9000 Communicator (1997), and Motorola StarTAC (1996). (Courtesy of WonderHowTo)

In the early to mid 2000's manufacturers were still making phones, such as Sanyo's 5300 and the Motorola Razor, but the biggest innovations and technological momentum was behind smartphones. Smartphones allowed users to have access to applications that they would normally only find on a PC. Blackberry's 5810 gave professionals quick and easy access to their e-mails and schedules. Microsoft entered the smartphone arena as well with their Pocket PC Phone Edition which ran on many PDA's including the HP Jornada 928.

Figure 2.5: Clockwise: Motorola Razor (2004), Microsoft Pocket PC Phone Edition (2002), and Blackberry 5810 (2002). (Courtesy of WonderHowTo)

Although significant progress had been made since the clunky and expensive mobile phones of the 1980's, the mobile devices of the early and mid 2000's were still difficult to use. The screens were small, the input methods were limited, and the PC inspired user interfaces created a less than ideal experience. In addition, the applications that ran on the phones were mostly proprietary and not updated often or sometimes not at all. Then in 2007-2008, Apple released the iPhone and with it the App Store. However, Apple was a player in the PDA market–in particular the Newton–but when it decided to go after that market it did so by creating a revolutionary user interface as well as creating a healthy ecosystem for developers to build apps.

The success of the iPhone set a new standard for smartphones: large touchscreen, excellent battery life, high end build quality, lightweight, camera, and GPS. Each new generation of the iPhone raised the bar further both in terms of aesthetics and hardware: higher resolution screens, longer battery life, faster processor, more memory, more and higher resolution cameras, noise cancelling microphones, more accurate GPS, wider range of network antennas, and larger storage. The combination of high end hardware, multiple sensors, and an excellent development platform gave developers access to more personal information and user context.

Table 2.1 summarizes some of the characteristics of the newest iPhone, the iPhone

6 Plus. From looking at this table one would think that this is a shopping list for three separate devices but this is not the case. This specification is typical of what one would expect to find in a modern smartphone. Complimentary to the hardware is an array of built in applications that came with each smartphone. A typical assortment is shown in Figure 2.6.

Table 2.1: iPhone 6 Plus Sample Specification

Chips	A8 chip with 64-bit architecture & M8 motion coprocessor
Cellular & Wireless	UMTS/HSPA+/DC-HSDPA; GSM/EDGE; LTE 802.11a/b/g/n Wi-Fi (802.11n 2.4GHz and 5GHz) Bluetooth 4.0 wireless technology
Location	Assisted GPS and GLONASS Digital compass Wi-Fi Cellular
Touch ID	Fingerprint identity sensor built into the Home button
Display	5.5-inch (diagonal) widescreen Multi-Touch display 1920-by-1080-pixel resolution at 401 ppi
iSight Camera	8 megapixels with 1.5 pixels Hybrid IR filter Face detection Photo geotagging Slo-mo video @ 120fps
Video Recording	1080p HD video recording @ 30fps or 60fps Video geotagging
FaceTime Camera	1.2MP photos (1280 by 960) 720p HD video recording
Intelligent Assistant	Siri
Power and Battery	Talk time: Up to 10 hours on 3G Standby time: Up to 250 hours Internet use: Up to 10 hours on LTE
Sensors	Three-axis gyro Accelerometer Proximity sensor Ambient light sensor Fingerprint identity sensor Barometer

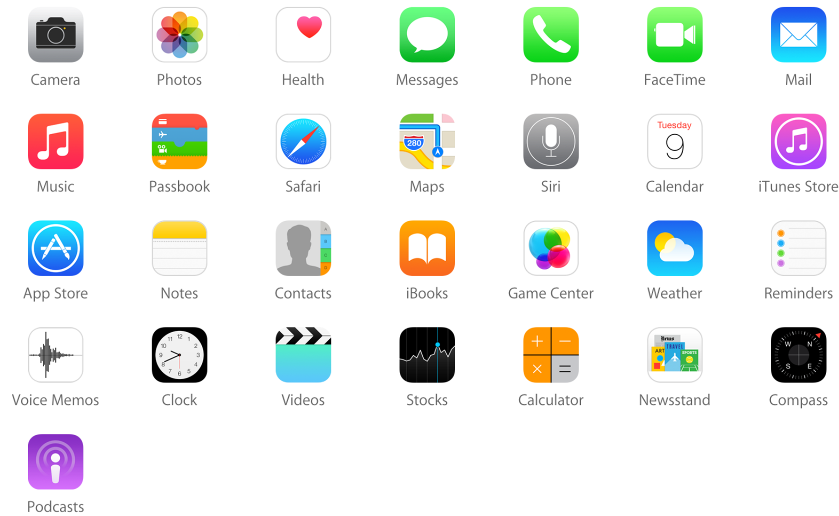


Figure 2.6: iPhone Standard Apps

Unsurprisingly, as the price of smartphones decreased and the number of features and quality increased these devices have proliferated. Figure 2.7 shows the per capita penetration of PCs, Smartphones and Tablets. In 2013 there were more smartphones in people's hands than PCs. Given the PCs successful history this is a remarkable feat. What this means is that in the near future people will be able to partake in more personalized experiences as their smartphones allow them to interact with their instrumented world.

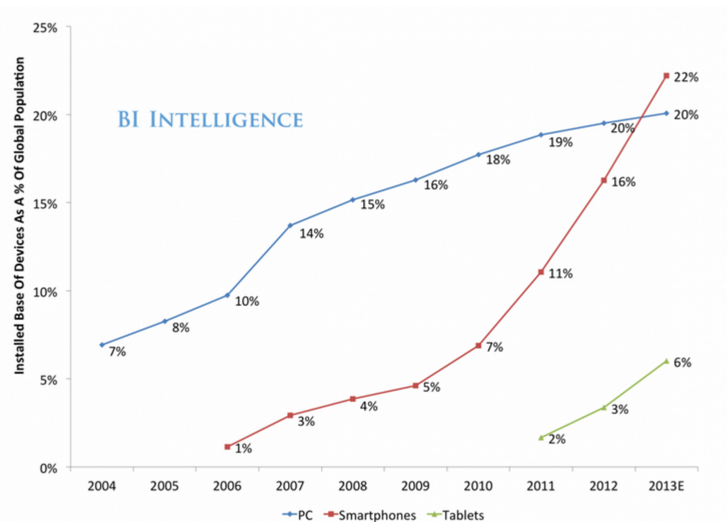


Figure 2.7: Global Device Penetration Per Capita.
(Courtesy of Business Insider)

2.2 The Internet of Things and Self Adaptive Systems

The pervasiveness of smartphones, and mobile devices in general, is a consequence of the lower cost and the miniaturization of all the components from which smartphones are built: CPUs, memory, antenna's, and battery. These economics have also played a role in ushering a new era in computing where every aspect of our physical world is instrumented and interconnected. The backbone for all these instrumented and interconnect things is the Internet and this new computing era is known as the Internet of Things (IoT).

Although the possibilities of what can be done when the environment is fully instrumented and interconnected are endless the current state of the art is driven by four main industries: 1) energy, 2) healthcare, 3) manufacturing, 4) transportation, and 5) the public sector.[1] These are the industries as identified by the Industrial Internet Consortium whose members including IBM, Intel, General Electric and 145 other members.[2] The term Industrial Internet was first coined by General Electric [EA12] and the work being conducted in this area falls under the general umbrella term of IoT.

Energy systems are very large and may be composed of old and new technologies such as coal plants or wind farms. The scale and hybrid nature of these types of systems makes them difficult, if not impossible, to manage by humans in terms of physically accessing the infrastructure and also in terms of maintaining an accurate and comprehensive model of the system at any given time. The solution that IoT proposes is to provide a framework that instruments and interconnects the energy infrastructure so that some control and maintenance can be performed automatically by computers while other aspects of the system can be monitored and controlled remotely by humans.

In the US it is estimated that 400,000 people die every year as a result of healthcare errors [Jam13]. These errors range from incorrect administering of prescription drugs to misdiagnosis. One aspect of these errors is that doctors and nurses are overworked and make mistakes. Another aspect is that healthcare is highly complex and it is challenging to create and maintain procedures that ensure patient care. An instrumented and interconnected healthcare system can relieve the burden on doctors and nurses by providing more accurate information on a patient's condition which in turn will lead to fewer mistakes as well as highlight possible procedural shortcomings [NHC09] [RMMCL13].

Manufacturing has a history of automation and automation can be considered one aspect of IoT. Despite the head start, manufacturing can still be improved by exploiting the IoT paradigm. Expansion and integration of the supply chain outside the factory to include

[1] http://www.iiconsortium.org/vertical-markets.htm
[2] http://www.iiconsortium.org/members.htm

more automatic integration with third party suppliers and eventually the customer herself will close the feedback loop between consumer and producer and allow the producer to provide higher quality products to the customer [DT08] [KMRF09].

As our cities become more populated and the price of energy continues to climb, new solutions around transportation will have to be implemented. Regardless of whether the transportation mechanisms are implemented as private, public, or a hybrid of both, the main driver will be efficiency. Efficiency not just in terms of making engines more efficient but in terms of traffic management. Traffic management requires real-time information and the ability to redirect the flow of traffic using different paths. These types of analytics and control require the kind of instrumentation and interconnection that IoT aims to provide.

The public sector affects energy, healthcare, manufacturing, and transportation. It does this either indirectly through laws and policies or directly by having a primary stake such as in the case of energy. In addition it also is responsible for local governance and the management of certain infrastructure such as water and sewage. As other industries migrate towards IoT and as funding to public programs is cut the public sector will be forced to adopt the IoT paradigm. This approach will create a public sector that reacts faster to suite the needs of its citizens such as in the cases of emergency response, crime prevention, or economic variability.

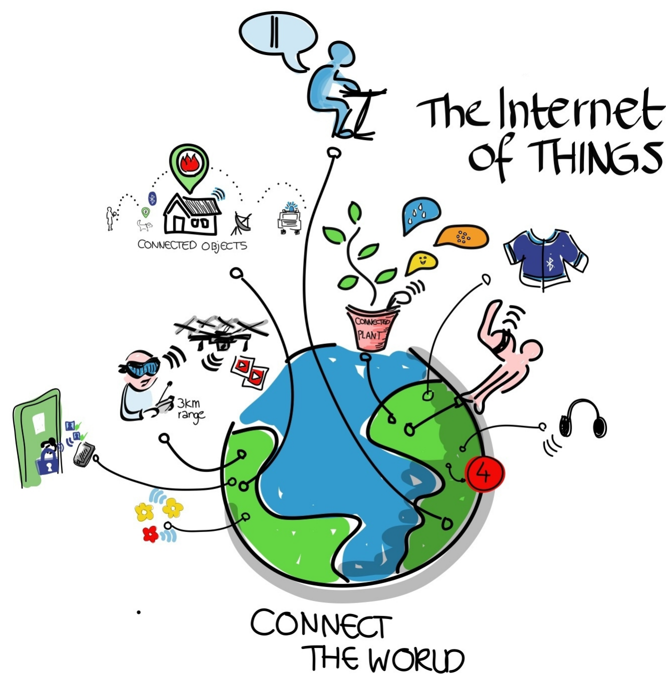


Figure 2.8: Internet of Things (Courtesy of Wilgengebroed on Flickr)

At the heart of IoT, and ubiquitous computing in general, lies context and feedback loops. In order for computing to do intelligent work and to fade into the background it has to be able to make run-time decisions without requiring human input. To accomplish this computing systems need to be able to gather context from the environment via sensors, process that context using models, and if necessary affect the environment in some way using actuators.

The basic notion of using context for human-computer interaction has been around since the early nineties. Schilit et al., 1994 were the first to identify the need for context awareness as a trait that should characterize the dynamic needs of software systems [SAW94]. Although the technology at the time was primitive by today's standards, IBM's Simon being released only a year earlier, most of the scenarios in Schilit's paper are still relevant and unsolved today: device selection by proximity, bandwidth requirements, device screen size, auto-installation of modules and drivers, and proximity triggers.

Contextual awareness on its own is not useful unless you have models and tools in place capable of analyzing and reacting to the information. One of the first examples of how the lack of such tools was hampering software projects was the difficulty IT professionals had with meeting emergent organizational goals. IT professional were given the difficult task of maintaining an expanding and increasingly complex IT infrastructure that was not manageable by human beings. This led to system and deployment failures, the creation of incomplete systems, and in lots of cases cancelled multi-million dollar projects [NFG+06].

IBM and several other industry leaders, such as HP and Microsoft, decided to take on this challenge. Although each company had their own solution that more or less addressed this problem it was IBM's Autonomic Computing initiative that lead the way. In their 2001-2003 white papers IBM clearly articulated the need for automated systems as well as an architectural shift in how software systems should be developed and maintained [IBM06].

The hypothesis of this new approach took the opposite view of what up to that point had been the traditional way of building software. Traditionally software systems were built under the assumption that all requirements would be known ahead of time and that once the system was built it was not bound to change much. These assumptions could not be further from reality. In sharp contrast, the proposed approach was to assume that there would never be a final or finished version of a software system since in an emergent organization there is no such thing as a final requirement: the target is always moving since business goals change [NFG+06].

This lead to the development of an architecture that allowed software engineers to control systems at a much higher level of abstraction using policies. Now when a new business rule was to be implemented it did not necessarily require the creation or modification of code. In addition to management via policy this new architecture presented a way to add

autonomic behaviour. Autonomic behaviour allows a software system to react to changes in its operating environment without human intervention. It constantly evaluates all the data about its state, the state of its environment, the applied polices and configures and optimizes itself to heal or to prevent scenarios that are contrary to policy.

At the heart of this architecture is the Autonomic Manager (AM) as depicted in Figure 2.9. The AM consists of a feedback loop know as MAPE-K loop which has four main parts: monitoring, analyzing, planning, and execution. Each part in the loop is used to determine if the software system, known as an endpoint, needs to be modified in some way in order to fulfill its policy. The endpoints are the most important part of this architecture since they provide access to the device or piece of software that is being controlled. It is at the endpoints that one usually finds the sensors, actuators, and users.

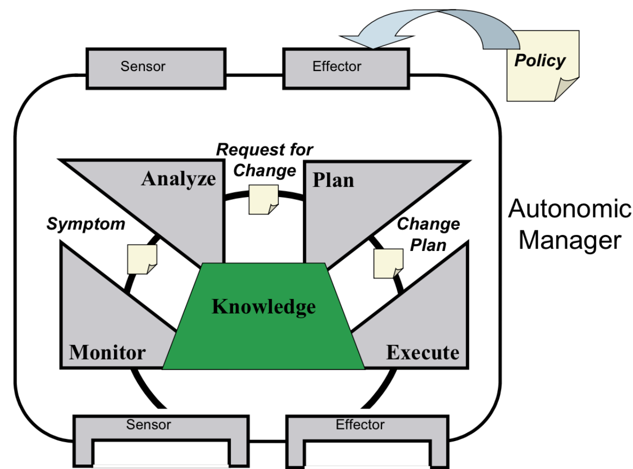


Figure 2.9: Autonomic Manager (AM)
[KC03] [IBM06]

Each AM is stackable and re-usable meaning that it can be used to create a hierarchy of AMs each responsible for a higher level of abstraction. Figure 2.10 shows the Autonomic Computing Reference Architecture (ACRA). At the lowest level, the managed resource, is the resource being managed. This could be as simple as a room heating-cooling system or as complex as a datacenter. On top of the managed resource there is an interface or touchpoint, that allows the AMs to connect to the managed resource. At the AM level several AMs can run in parallel. The AMs at this level are responsible for implementing the defining properties of what makes a system autonomic: self-configuring, self-healing, self-optimizing, and self-protecting; or known as self* properties [IBM06].

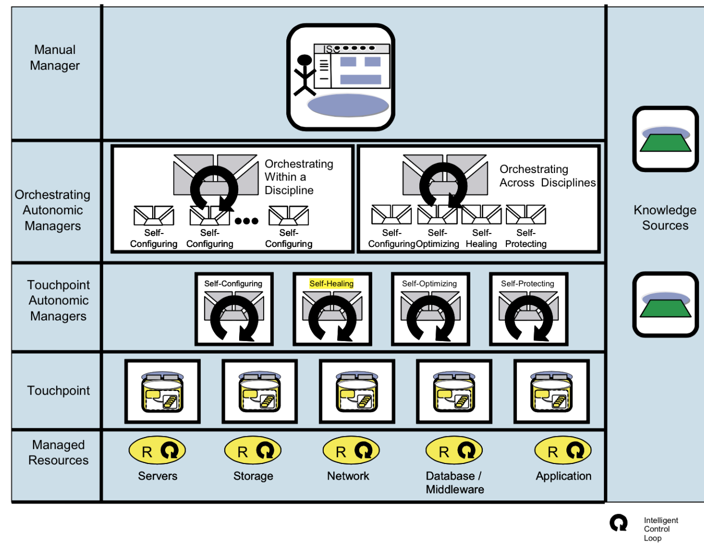


Figure 2.10: Autonomic Computing Reference Architecture (ACRA) [IBM06]

At the orchestration level and above you the AMs introduce higher levels of abstraction. The manual manager at the top level is where the actual user interaction takes place. The height of ACRA is not limited to just these levels and varies with the level of system complexity; however, fundamentally an autonomic system has one or more AMs that consume policies and manage self* properties for a system.

Although the original white paper does not use the word 'context' and 'feedback' explicitly to identify the flow of information, contextual feedback is exactly what needs to happen inside an effective autonomic system. Müller et al. have argued that feedback loops are the most critical part of an autonomic system and that Software Engineering as a field ought to employ Control Theory tools developed by other engineering disciplines for implementing autonomic systems [MKS09] [MPS08].

2.3 Cloud Intrastructure

Designing an application that is contextually aware is challenging enough but equally challenging is building the infrastructure on which such an application can run. The most recent momentum has been to deploy applications in the cloud. The cloud is composed of servers and network infrastructure connected to the Internet. On top of the physical infrastructure there exists an operating system that abstracts the complexity of the underlying hardware. Application developers can request multiple hardware resources and, using the virtualization that these cloud operating systems provide, deploy their applications.

2.3.1 Challenges in the Cloud

One of the drawbacks of current cloud solutions is that cloud infrastructure provides little context. Some of these drawbacks come from the way the network stack has been designed with each layer being responsible for a specific task and with no vertical integration between them [FS11]. This is problematic because if an application needs contextual information about its underlying network infrastructure to make a self* decision it has limited context to make informed decisions.

Another drawback is that if an application needs to run on a different cloud provider the deployment procedure and the performance of the application may vary [IYE11]. Although most cloud providers use the same cloud operating system, some of the custom APIs and nuances of their particular platform make it such that developers have to spend extra time making sure their applications run on different clouds. In addition to deployment and performance other risks and challenges exist, including legal compliance, economic sustainability, and environmental efficiency targets. All these complexities are difficult, if not impossible, to manage by humans. One approach is to use autonomic toolkits that developers can leverage to deploy their applications on heterogenous clouds without requiring complete knowledge of each cloud [FHT+12].

2.3.2 Smart Applications on Virtual Infrastructure

The Smart Applications on Virtual Infrastructure (SAVI) project is a partnership between Canadian industry, academia, and research and education networks aiming to address some of the current drawbacks of cloud infrastructure and explore Future Internet applications. The basic premise of the SAVI infrastructure is to provide applications access to infrastructure context and also give applications the option of running on smaller nodes that are potentially in closer proximity to the user.

The vertical integration and resource granularity that the SAVI testbed provides has allowed researchers in different fields to investigate different approaches including: software defined infrastructure (SDI) that abstract heterogeneous physical resources for the application developer [KLBLG14]; software defined networks (SDN) which allow interconnection of heterogenous resources such as FPGAs and GPUs in a data center [LKBLG14]; and extensions of virtualization for abstracting wireless hardware [WTLN13]. In one form or another, all this research is aimed at improving performance which can translate to improving the user experience or optimizing applications in the case of green computing.

2.3.3 Measuring Latency

Statements about performance are usually expressed in terms of latency and throughput: the time between input/output and the rate of input/output, respectively. In the case of a typical productivity application, such as Microsoft Word, the performance depends on the latency between the user's keyboard and the visible reaction on the screen. In reality when talking about latency it is important to distinguish the different parts of the latency. For example, the latency of the keyboard, the latency to process the keypress, and the latency to draw the result on the screen. Breaking down the latency into smaller parts provides a sense of proportion about the latency and enables the identification of bottlenecks for different applications.

Cloud infrastructure provides general computer resources which can support different types of applications. Each application type has different requirements when it comes to performance; consequently, the factors that affect latency and throughput are going to be different and yield different answers when managing tradeoffs. For example, in a study that looked at running high performance computing applications on Amazon's EC2 platform found that the latencies resulting from the variety in CPU architectures and the speed of the internal network were six to twenty times slower than their existing clusters [JRM+10]. Other studies that looked at deploying video game processing to the cloud found that the latency introduced by the external network, rather than the internal network and the CPU, played a larger role in the overall latency [CWSR12] [CCT+11]. Even though the results from these two very different application domains led to the same conclusion, that EC2 is not a good alternative for either application, the latency types that lead to these conclusions are different.

Knowing which latency to measure is one part of the problem. The other part is determining the best method for doing so. Jackson et al. used a variety of tools and approaches since the latency questions they are trying to answer are complex [JRM+10]. To measure the network latency they used the ping-pong approach which simply measures the round trip time for a given task. For their other tests they used high performance computing benchmarks such as DGEMM, STREAM, and PTRANS[3] to determine the performance of floating point execution, sustainable memory bandwidth, and transfer rate of large arrays from multi-core arrays, respectively.

Similarly, Choy et al. used the ping-pong approach by measuring the round trip time for a TCP handshake to measure network latency for their experiments [CWSR12]. Use of the ping command is a valid way of measuring network latency; however, due to the network and server setup for their experiments they were not able to guarantee that devices on the

[3]http://icl.cs.utk.edu/hpcc/

network would not filter the ping commands. Conversely, other researchers investigating the impact of virtualization on network performance were able to use the ping command as their main method for measuring latency [WN10].

2.4 Software Complexity

Looking at application design from a broad perspective, building great applications is not just about algorithms, frameworks, and features. It is about making the user experience simpler—in the sense that the user has to deal with fewer choices. It may seem counter-productive, that removing choice will make the user happier, but it has been shown that this is indeed the case [Sch09]. In fact, it has been shown that too much choice has the opposite effect; it makes users less happy. For example, walking into a bakery that has three varieties of bagels may be a good amount of choice since one of the three bagels you probably hate and then you only have two bagels to choose from; however, if you walk into a bakery with 18 varieties of bagels the choice becomes much more difficult and when you finally make your decision you are actually less happy because you keep thinking about all the other bagels you could have chosen instead.

It takes mental effort to manage the trade-offs between different bagels and it is this uncertainty that makes humans so miserable when faced with too much choice. So the obvious solution is less choice, right?! The key is to create a perception of less choice but nut without removing options. When you limit choice without limiting overall options then what you are doing is shifting the complexity around.

When you create a product, such as a car or a piece of software, it is very easy to get carried away with features. The source of features can either be your imagination or at the request of the user. A large number of features can have negative effects on users even if users initially think that having all those features is a good thing. Features mean that users have to make choices. Choices require management of trade-offs and that leads to anxiety [THR05]. Marketing research suggests that the best way to deal with a deluge of features is to continue offering a large variety of products but then customizing them for individual users; as in the bagel example. Maintaining features while minimizing the features exposed to users requires a shift in complexity from the user to the back end.

Software design suffers from the same feature creep issues but with the added problem that as software developers we impose, sometimes unnecessarily, the architectures or constructs inherent to computing systems. This has been going on for so long and is so prevalent that we sometimes think of these constructs as features in themselves. Although some of these constructs existed due to hardware limitations and/or the nature of the application itself, the time has come where hardware and social expectations have allowed us

to move beyond some of those features. For example, the use of logins, or filling out pages of profile information, or having to manually filter out geographically sensitive information that is not near your location. Logins, profiles, and geography are nice features to have but if you are trying to simplify your application these are the sort of features and complexities that can be moved to the back end, away from the user.

2.5 Human Communication

Our ability to communicate has grown in parallel with technology. At the dawn of human history communication was primarily composed of a set of simple gestures to the person next to you. As language developed and roads were built we were able to communicate with those in the next village or maybe as far as the next valley. In 1867 Alexander Graham Bell made the first voice transmission over wire. This was a game changer in communication since distance now played less of a role in the act of communicating—talking to someone dozens of kilometres away took seconds as opposed to days. 34 years later Guglielmo Marconi made the first wireless transmission over the Atlantic making physical geography even less of a barrier to human communication.

Since 1901 we have seen improvements in communication technologies that have resulted in higher quality at a lower cost to the point where we take for granted what was not even possible 150 years ago. If you live in an urban center in North America, and in most parts of the world, you have the ability to make a wireless call to anyone. With geography playing less of a role in communications it certainly appears that the world has shrunk but some evidence suggests that we still want to only communicate with those in our village.

There is this notion that we are all connected to everyone else in the world by six degrees of separation [Mil67]. That is, you are six people away from knowing the Pope. Recent work on online social networks investigates this theory as well as the role of geography. In one case the researchers looked at the relationships on the popular blogging website LiveJournal—an online community where members are encouraged to interact with each other via their personal blogs. They discovered that although geography was a common factor between those who were friends on the site, geography alone was a poor predictor of who would actually be friends [LNNK+05].

Other work done by researchers at the largest social network, Facebook, investigated what social and communication insights could be gained by looking at the Facebook social graph [UKBM11]. They found that the Facebook social graph is nearly fully connected and most users have 4.7 degrees separation to everyone else. When looking at the community structure in the graph they found that it closely followed the geographies of countries and cities so even though people are only 4.7 friends from everyone else and communication is

nearly instantaneous it would seem that people are still socializing along familiar geographic, language, or social boundaries. In other words, it appears like your online social network mirrors your offline social network.

Although it may sound comforting to know that you are six, or 4.7, people away from the Pope that in and of itself is not necessarily useful from a communications perspective. You may have 100 people in your Facebook, LinkedIn, or Twitter network but it does not mean that you communicate with them on a regular basis and therefore do not have the necessary rapport to be able to make the six person jump. Intuitively this makes sense; you only have so much time in a day and cognitive energy to manage a finite set of communication channels. Our social signature, that is the people we interact with, changes as we move from one place or another or change jobs or transition from one stage of life to another [SLL+14]. Your digital signature is a finite queue that as new people are added others are removed; this shuffling is highly dependent on your situation: *your context*.

2.6 Prediction Using Location

The access of location information from both mobile devices and non-mobile hardware, such as routers and servers, has given software developers a unique opportunity to complete the spatial picture of all their users and infrastructure. This granularity of location information was not available ten years ago and the sudden abundance of this kind of information has resulted in a myriad of interesting research directions, including the Microsoft Research GeoLife project.

2.6.1 GeoLife

GeoLife is a location based social networking service. This service has been used by researchers at Microsoft to investigate different ways to model and use GPS data. At its core GeoLife uses GPS data accumulated by 107 users between May 2007 and December 2008. The user base comprised 49 women and 58 men. These users were given several different GPS devices, some of which were pure GPS receivers while others were smartphones, and were asked to log their outdoor movements. Users were motivated by a financial incentive to collect as much data as possible: the more GPS data a user collected the more money they received. In the end approximately 24,876,978 GPS points were collected. Most of the data is from Beijing China, but data was also collected from several other cities within China, USA, Japan, and South Korea [ZZXM09b].

2.6.2 GeoLife Research

In one of the first papers published using this data, Zheng et al. looked at ways to model this data with the goal of inferring the importance or popularity of a geographic location [ZZXM09b]. The importance of a location depends on the number of people visiting it as well as the travel experience of each person in that region. For example, a native of Beijing knows the city better than a tourist from Victoria and as such their experience should carry a higher weight on the relationship between person and location (i.e., if two locations have an equal number of people going to it but one of these locations has experienced people from Beijing while the other has inexperienced people from Victoria then the former location should be considered more important since that location has a high proportion of experienced people).

To model this relationship Zheng et al. used the Hyperlink-Induced Topic Search (HITS) model originally developed during the early stages of the world wide web (WWW) as a way of indexing and searching web pages. HITS uses the concept of hubs and authorities where hubs are webpages that are large indexes that point to webpages containing the actual information known as authorities [Kle99]. In web page terms a good hub points to many good authorities and in turn a good authority is pointed to by many good hubs. HITS is applied to modelling location importance by treating users as hubs and locations as authorities. For a given region a user will have a hub score which is used to gauge their experience in a specific region.

To test their approach Zheng et al. compiled a team of 29 people, 14 females and 15 males, who had been in Beijing for more than six years. These people represented individuals who would be considered region experts and as such should be able to identify interesting locations. Each individual was given a list of ten popular locations as determined by Zheng's model. For a baseline the rank-by-count and rank-by-frequency algorithms were also used to generate ten important locations. For each set of results users were asked to rate how representative the results were of a given region, whether the results offered a comprehensive view of that region, and whether they were novel. In all cases Zheng's model outperformed the traditional approaches of rank-by-count and rank-by-frequency.

Subsequent research on this work explored ways to predict relationships between popular locations by not only looking at popularity, or rank, but also by looking at how related locations were to one another [ZZXM09a]. For example, assume that a user has given ratings for locations A and B how can we predict the rating this user would give for C? One approach is to use proximities of A, B, and C to predict the rating of C.

One common approach for making rank based predictions is using Slope One algorithms [LM05]. Slope One algorithms are simple but very effective at predicting ratings based on previous users' ratings. Zheng et al. argued that, in addition to a rank assigned

to a location, the semantic meaning encoded in the geography between locations is also important when making location based recommendations. Their approach outperformed the Slope One algorithms when predicting consecutive locations in the GeoLife dataset.

Using proximity to predict future locations is a powerful tool that Zheng et al. continue to investigate. In more recent works they applied the lessons from previous research to publish work on generating smart itineraries [YZXW10]. Smart itineraries are automatically generated by using regional travel experts to build models which are then used to generate an itinerary for specific start and stop locations. A combination of simulation and user study was used to validate their approach by comparing it to two baseline algorithms: rank-by-time which recommends itineraries that match closely in duration with the user's query duration; and, rank-by-interest which suggests itineraries based on the aggregate interest of travel experts. The results showed that their algorithm produced better recommendations for itineraries with longer durations and worked equally well as rank-by-time and rank-by-interest for shorter durations.

2.7 Summary

This chapter introduced sensors, mobile devices, the Internet of Things, and self adaptive systems as the underlying motivations for this thesis. In Section 2.1 the evolution of mobile devices was used to show how sensors made their way into our environment. These sensors have given rise to a new era in computing known as the Internet of Things (IoT). The IoT era and its impact on industry is explained in Section 2.2 along with its relationship to self adaptive systems. With IoT comes a new set of challenges in the areas of application deployment and context management. We address these challenges as part of our research questions.

Chapter 3

Location Based Social Networking

One of the contributions of this thesis comes in the form of a location based messaging application called Yakkit—a combination of an iPhone application and a set of supporting web services. The goal of this implementation is to help answer our first research question: what kinds of applications can we build when we shift complexity away from the user with the intention of nurturing normal human communication?

3.1 A New Twist on an Old Idea

Research into social networks suggest that geography plays a significant role when looking at existing groups of friends but it plays a lesser role inside those social networks for creating new relationships. Perhaps, this is the result of the way that online social networks have been built and not that geography is not a critical aspect of relationship forming. If we think about our own experience when meeting new people our location plays a significant role in that exchange.

Thinking about this observation more broadly we considered examples of technologies that have amplified the way in which humans naturally communicate while at the same time did not affect communication due to the nature of the technology itself (e.g., a website needs a login so you need to create a special name just to talk to someone). Truck drivers have used the Citizens Band (CB) radio system for decades to socialize and broadcast important information such as police presence on the road: "Bear Taking Pictures" indicating a speed trap.

CB radio was first introduced in the Unites States in 1948 with the goal of providing citizens with basic personal communications. CB radio is still in use today by truckers, cab drivers, and hobbyists. To use this communication system you need a radio, such as the one depicted in Figure 3.1, be tuned into the same channel as the person you want to

talk to, and be physically close enough for the radio waves to reach them. In this sense, CB radio is a natural extension of human communication as it simply allows your voice to carry further. Conceptually this is simple to understand for most people and the concept of geography, more specifically proximity, gives people a familiar perspective (i.e., your voice carries as far as your yelling can be heard).

Figure 3.1: CB Radio Base Station

With the CB notion in mind we endeavoured to design an application that enhanced normal human communication without burdening the user of having to worry about intrinsic qualities of the technology itself such as creating logins, filling out profiles, and managing relationships—create a new tool without forcing the tool's complexities upon the user [LM13].

3.2 Yakkit

To demonstrate our approach we created, and iterated upon, the Yakkit iPhone application and its supporting web services. At its core Yakkit works like a CB radio—it allows you to communicate with those around you. There is no login, no profile, and no requirement to add friends or wait for someone to add you. You start the app and you are able to communicate with those around you immediately—just like CB radio. We extended this basic concept to also allow the pinning of messages to virtual billboards; in this way you could leave a message for someone else. These two concepts of instant communication and message pinning are analogous to the way you would go up to someone on the street and

strike up a conversation or post a message on a community billboard. In this way, Yakkit replicates normal human communication in the virtual world.

The original Yakkit iPhone application was supported by one monolithic web service running on top of a cloud distribution service called iCon Overlay [DLM11]. Figure 3.2 shows a high level overview of the early system. The application monitored the user's location as well as provided the user with a user interface for interacting with the application. Messages were routed through the Yakkit Service and during this routing the Yakkit Service would use its current GPS model of all the users in the system to determine where to forward messages.

Figure 3.2: Original Yakkit Architecture

Although the design did not explicitly include an autonomic manager (AM) nor follow an ACRA architecture, the core ideas of using context and feedback loops for the purpose of making run-time decisions heavily influenced Yakkit's early design. Figure 3.3 depicts how this early implementation was related to the phases that exist inside an AM. At the center of the Yakkit Service is the knowledge base which consists of a k-d tree data structure which holds the current location of all users. The Yakkit Service monitors all its connected users for location updates and adjusts the knowledge base if need be. When a new message arrives the Yakkit Service analyzes the locations of all the users to determine which users the messages should be forwarded to. Once the list of users is computed the messages are sent to each user by indexing the appropriate connections [Des13].

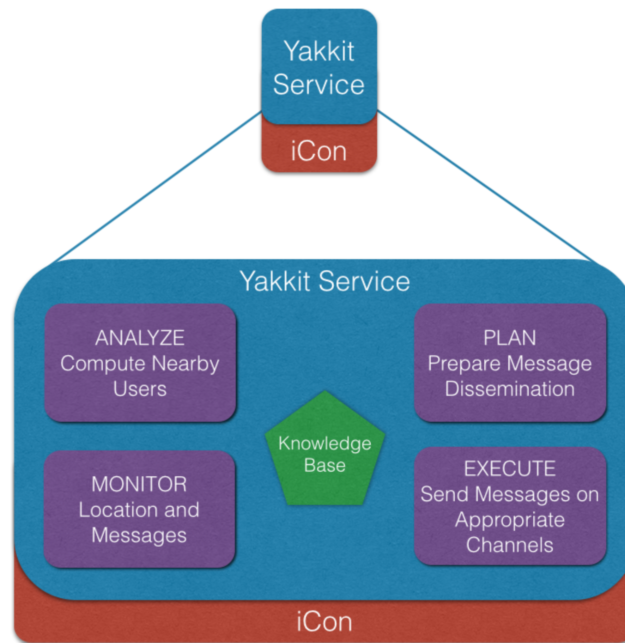


Figure 3.3: Autonomic Manager for Yakkit Service

Figure 3.4(a) shows the UI for this early implementation that encapsulates the chat and billboard concepts. At first glance this interface appears to be similar to Google Hangout or Skype, but what is different is the way in which it immediately connects you to nearby users. In the simple scenario depicted in Figure 3.4(a) a user is trying to find a place to eat. This is a location based activity and should not force you to login or fill out a profile; however, an option should still exist that allows a user to configure the application manually if they feel they want to divulge that kind of information. In the example in Figure 3.4(a) the users have decided to use nicknames to help refer to each other.

Chat is in part useful because it is spontaneous and non-persistent; however, there are situations where you may want to persist a message for someone to view at a later time. For that scenario we added an interface to the Yakkit iPhone application for supporting the concept of billboards. Figure 3.4(b) shows the UI for this part of the application. Again, the goal here was to create something that replicates normal human interaction without imposing technological constraints. Billboard, as its name implies, is a virtual billboard where users can anonymously create billboards tied to a geographic location and post messages. Each billboard has a broadcast area that affects when a user will see a billboard. Figure 3.4(c) shows a birds-eye view of a user (blue dot), the user's broadcast area (green square), and any nearby billboards (red pins). Although in this case the broadcast area is a square there is no technical limitation to the type of shape that this area can be.

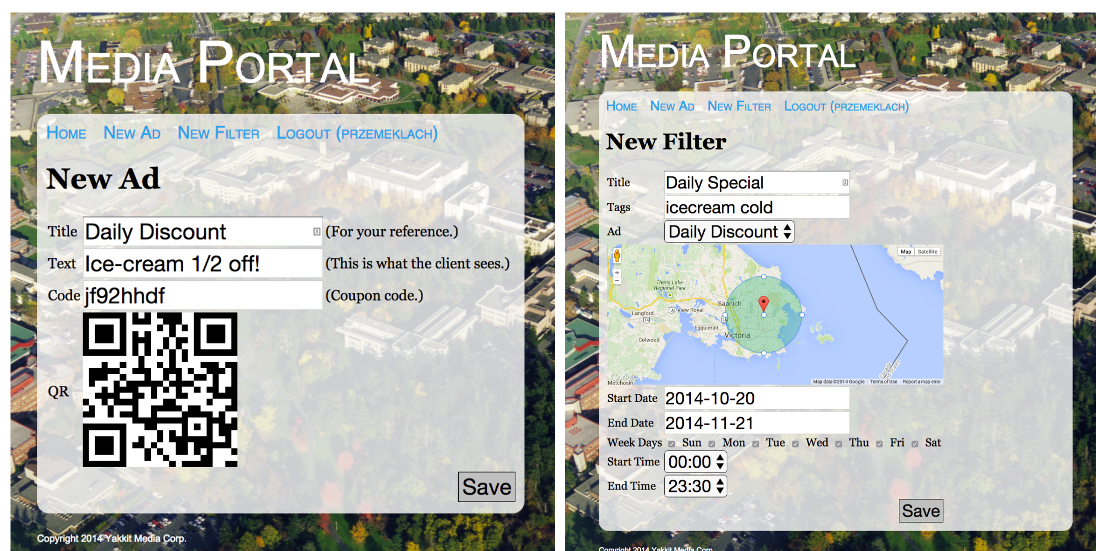
(a) Chat View (b) Billboard View (c) Map View

Figure 3.4: Yakkit iPhone App Interfaces

3.3 Yakkit 2.0

After completing the first version of Yakkit we designed new services to support different types of clients in addition to the iPhone. The monolithic web service we created made it difficult to implement such changes and we decided to re-design the application completely [Lim14]. As part of our continued effort to make Yakkit context aware, we added the ability to create ads and to inject them into a conversation at an appropriate time. Our premise on ads is that if you send advertisements to potential clients who are nearby and actually need your product you are more likely to make a sale.

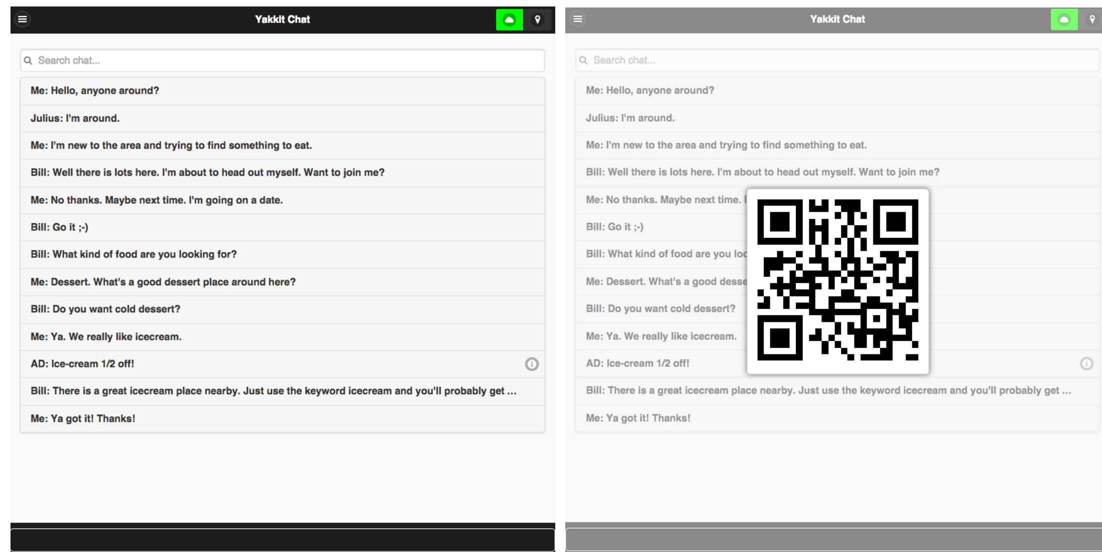
Figure 3.5 shows the user interface for the advertising portal. Ad creation is a two step process: creation and scheduling. Figure 3.5(a) shows the interface for creating ads. Here the user would enter the message they want their clients to see as well as any promotion or discount codes. The portal automatically creates a QR code which the client can take to a store to claim the offer. Figure 3.5(b) shows the interface for scheduling the advertisement. Here the user enters the times of the week during which the advertisement is to be available as well as the area in which it should be broadcast. Using this portal a company such as Starbucks can schedule the broadcasting of coffee discounts to nearby customers in the hope of improving sales during non peak hours.



(a) Ad Creation (b) Ad Scheduling

Figure 3.5: Advertising Portal

Figure 3.6 shows the new browser interface for Yakkit Chat. In this example two users are talking about finding a good place for ice cream. As before, this interaction is possible by these two users being close to one another but now in addition to analyzing their locations,



(a) Chat (b) Ad Presentation

Figure 3.6: Yakkit App

their conversations are also being analyzed to see if the content of that conversation can be supported by an ad. In the 4th message from the bottom in Figure 3.6(a), one of the users talks about ice cream in a positive way "Ya. We really like ice cream". A positive sentiment is inferred from this statement and the subsequent message is an advertisement that includes a discount code for ice cream. When the user clicks on the message they are presented with a QR tag as depicted in Figure 3.6(b) which they can then use to claim in the store.

To support these new features and alleviate the issues surrounding our monolithic Yakkit Service we implemented a new architecture as depicted in Figure 3.7. The Yakkit Service has been decomposed into four services: Locality Service, Web Service, Semantic Service, and Chat Service. Together, these services interact with one another by messaging each other directly or indirectly via the two data stores: Location Data and Ad Data. Each service is designed to be as decoupled as possible (i.e., it is unaware of the application that it is part of). For example, the Locality Service maintains a list of users and computes their proximity to one another, it does not care if it is doing so to support the chat, billboard, or some other future application or service.

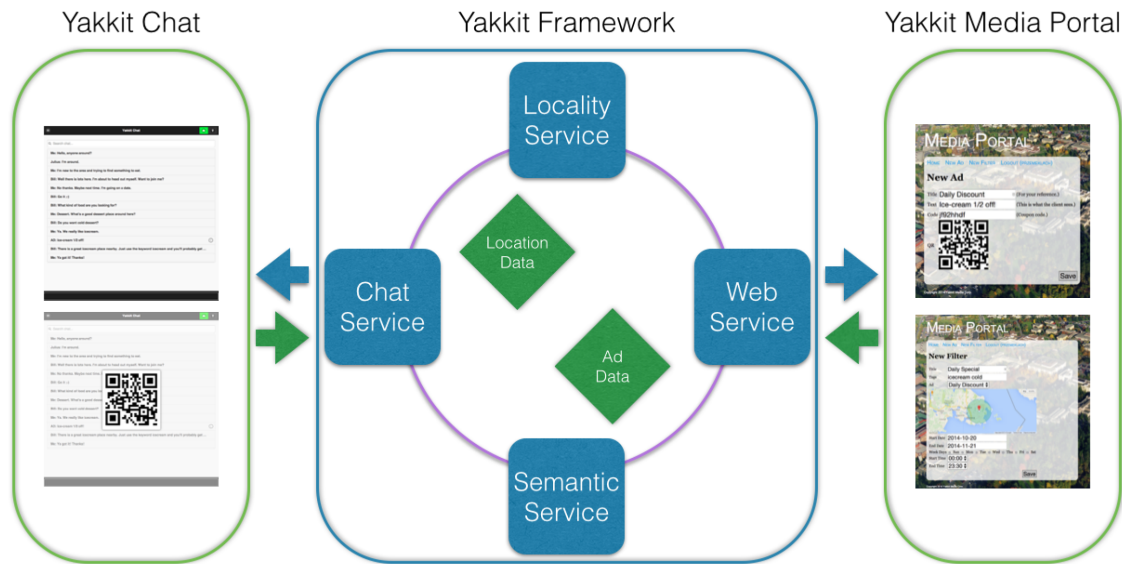


Figure 3.7: Yakkit Version 2.0

Not all services inside of the Yakkit framework are created equal since some have features that are common across different applications while others are not. Services such as the Location Service are not tied to a specific application. Similarly, the Web Service provides merchants with the ability to manage ads as depicted in Figure 3.5, but it does not care how those ads are used. On the other hand, the Semantic Service is an example of a more tightly coupled service as it relies on the Ad Data data store which in turn is managed by the Web Service. Finally, the Chat Service has the highest coupling since in addition to routing messages it uses the Location Service and Semantic Service to do so. Notably, billboard is missing from the new version. The decision to leave billboard out was one of time constraints and not because it is no longer useful. Given this new architecture adding billboard is easier than with the original architecture.

Figure 3.7 shows how the services inside the Yakkit framework support the two new applications namely Yakkit Chat and the Yakkit Media Portal. As new application ideas emerge or if new services need to be added to the framework to make an existing application smarter we believe that this framework with its decoupled approach will make the implementation of such services easier. The logical decoupling of services is further supported by using communication protocols between the services that allow the services to run on different machines on the network. Using this approach we can deploy each service on hardware that is optimized for that specific service to try to optimize the service for a specific user experience (e.g., minimizing latency for Yakkit Chat). The underlying theme of the

Yakkit implementation is moving complexity away from the user. The only way to simplify the user experience while at the same time preserving features is by moving, rather than removing, complexity. In the IT world context and feedback loops were used to simplify the interaction, or user experience, of IT professionals. In the case of IT professionals the need arose because humans were no longer able to react fast enough to the changes in the environment in which their applications were deployed. In the case of Yakkit, and social networking in general, we postulate that complexity should be removed not only in cases where it becomes impossible to manage but in all cases where users have to use software to achieve a goal. To realize this goal, context and feedback loops are treated as first class citizens when making Yakkit design decisions [MKS09].

3.4 Yakkit Challenges and Approaches

During the initial and subsequent implementations of Yakkit we came across two main challenge areas that we decided to explore using experimentation: context analysis and deployment. Context analysis can be challenging because it requires users to first identify what context is relevant and secondly how the context variables affect one another. We chose to investigate geography as our sole context since location is the most interesting from a social networking perspective; it is also the the most pervasive measure.

The Yakkit implementation uses location to figure out who is nearby but location is not enough to create a managed experience since areas with high user density would make Yakkit unusable. To address this problem we set out to model existing user trajectories to see if we could use past user behaviour to help predict what chat messages or billboards current users may be interested in. This approach is also motivated by the aforementioned research into the structure of social networks: if users' online social structure is driven by their offline social structure which in turn is highly dependent on location then perhaps it is worth exploring location context as a first class entity for an online social network.

The current Yakkit framework implementation runs on one server. Yakkit clients connecting from anywhere in the world connect to the Yakkit Chat Service and Yakkit Billboard Service located at the University of Victoria. This is not ideal since users in Europe or Asia will have a worse experience than users in North America. To address this problem we introduced the concept of a Registrar that routes new users to a server nearest to them to see what effect geography has on message latency.

3.5 Summary

This chapter introduced Yakkit—a location based messaging application. Using Yakkit people can instantly communicate with those around them just like they would by going up to someone on the street or shouting in a crowd. Yakkit enhances this normal human communication by using context to extend the range of the communication without the need for the user to have to deal with the complexities of the application. Our first research question was: what kinds of applications can we build when we shift complexity away from the user with the intention of nurturing normal human communications? Yakkit is one such application. In addition, the Yakkit implementation poses further challenges in the areas of deployment and context analysis which are used as motivations for the remaining two contributions of this thesis.

Chapter 4

Data Mining User Trajectories

In Section 2.6.2 we discussed how raw GPS data can be used to find popular locations, identify a relationship between popular locations, and generate itineraries based on those popular locations [ZZXM09b] [ZZXM09a] [YZXW10]. Zheng et al.'s approaches improved on existing algorithms, such as HITS and Slope One, and showed that the GeoLife dataset is useful for predicting where people may want to travel [ZZXM09b].

To build on this work we take a step back and try prediction from a different perspective. We explore whether geographic boundaries, both virtual and real, can be used to make useful predictions using the GeoLife dataset. The second contribution of this thesis comes in the form of experimental results that show whether the anonymous GeoLife dataset is useful for modelling and predicting user destinations. The goal of this experiment is to help answer our second research question: how can we model anonymous user GPS location context to predict meaningful destinations without direct intervention by the user?

4.1 When Location Is Not Enough

In the latest version of Yakkit we used location and a simple semantic service to automatically connect people with one another and personalize ads based on their conversations. In areas with low population densities location may be all that is required for connecting people and creating meaningful manageable conversations. In densely populated areas, such as New York, solely relying on location for inferring connections would result in an experience that is not manageable due to the large number of messages that show up on the screen.

This situation is analogous to what would happen to our original inspiration for Yakkit—the CB radio. If there are too many people nearby on the same channel it would be nearly impossible to talk. People would be cutting each other off and you would never be able to understand what anyone is saying let alone get your own word in. CB radio addresses this

problem by having multiple channels that separate users. In addition, users of the CB radio system follow a voice protocol that helps in preventing them from talking over one another. Since Yakkit exists in the virtual world we are not bound to the same physics as CB radio but we can, yet again, take inspiration from how CB radio designates its channels.

Although the specifics of channel designation vary between countries the basic idea is that specific channels are designated for specific uses (e.g., channels 1-3 are for amateur radio while 9 is reserved for emergencies). A user would choose a specific channel based on their personal goal which could either be to chat with a bunch of other amateur CB radio users or get in touch with the authorities during an emergency. In the case of Yakkit we want to be able to create these channels automatically using context from the environment. To help realize this vision we want to see if we can leverage more location context by modelling the locations that people have visited and using those models to predict where a new user may potential end up. By knowing where we think a user may end up will allow us to filter conversations that are relevant to that user's goal.

4.2 Experiment Setup

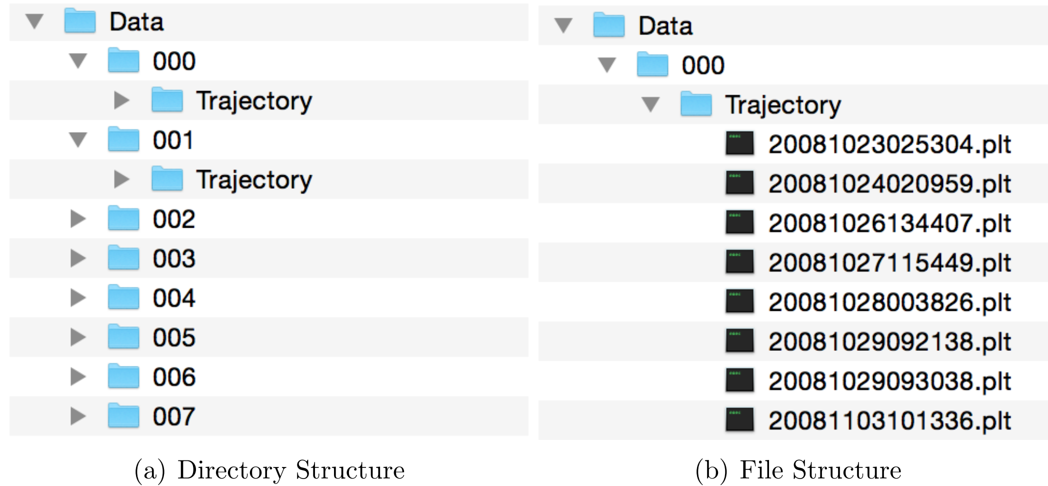
To model our location data we decided to look at the relationship between where trajectories begin and where they end. The hypothesis is that there is a relationship between the start and end points and that we can use this relationship to predict where someone will end up if we know their starting position. To test this approach we used the GeoLife dataset. The dataset contains anonymized GPS points stored in a comma separated values (CSV) text format. Before we could model the dataset, we had to clean and transform it into a format that was easier to work with.

4.2.1 Data Preparation

As with any real world data there is always noise and it is important to understand where to draw line between the signal and the noise [Sil12]. The GeoLife dataset is available through a Microsoft website[1] and contains a series of directories and files as depicted in Figure 4.1.

The root folder contains a numbered directory for each user and in turn each user has a Trajectory folder as seen in Figure 4.1(a). Although the data is anonymous, users were given a unique identifier to help group the location data. Inside the Trajectory folder is a list of files, Figure 4.1(b), which contain the actual GPS data. The contents of these files, Figure 4.1(c), follows a CSV format for describing a GPS position and includes the following fields of interest to us: latitude, longitude, elevation, date, and timestamp.

[1] http://research.microsoft.com/en-us/projects/geolife/



(a) Directory Structure (b) File Structure

```
Geolife trajectory
WGS 84
Altitude is in Feet
Reserved 3
0,2,255,My Track,0,0,2,8421376
0
39.984702,116.318417,0,492,39744.1201851852,2008-10-23,02:53:04
39.984683,116.31845,0,492,39744.1202546296,2008-10-23,02:53:10
39.984686,116.318417,0,492,39744.1203125,2008-10-23,02:53:15
39.984688,116.318385,0,492,39744.1203703704,2008-10-23,02:53:20
39.984655,116.318263,0,492,39744.1204282407,2008-10-23,02:53:25
```

(c) File Contents

Figure 4.1: GeoLife Data Structure

The text based format of this dataset is not a format that is easy to work with, nor is it scalable for processing. To alleviate this problem we imported the dataset into a PostgreSQL database which had the PostGIS spatial extender installed. Each GPS coordinate, or point, is now stored using a geometry data type and projected using the 4326 Spatial Reference Identifier (SRID). An SRID is a unique value used to identify a projecting or local coordinate system in mapping applications. We also simplified the structure of the dataset by aggregating all the points for a single user. Analysis of the dataset and the literature did not reveal any reason why a set of coordinates was broken up into different files. Figure 4.2 shows the schema that was use to store the dataset.

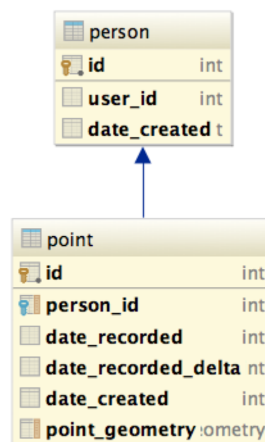


Figure 4.2: Imported Dataset Schema

One benefit of storing the location in a GIS enabled database is that we are able to leverage existing GIS visualization tools, such as qGIS, to examine the data. Figure 4.3 shows the output generated by qGIS with all the individual GPS points. At this scale the points appear as lines even though they are points. As first explained in Section 2.6.1 the data was collected all over the world in countries like China, USA, Japan, and South Korea, with the vast majority concentrated in Beijing. Using this visualization we confirmed that this was indeed the case. We were also able to identify areas that, due to their low point density, would not be appropriate for modelling and would simply be considered noisy data.

Figure 4.3: All GeoLife Points (Scale 1:64,000,000)

To help narrow down our dataset we used two criteria: point density and area. The point density had to be high in order to be able to claim statistical relevancy for our results. The area had to be on a human scale (i.e., an area that a normal person can reasonably be expected to walk or go for a short drive). These requirement are motivated by Yakkit's proximity concept: connect you with users that are nearby as you will never care about something that is 200 km away. With density and area in mind we looked for natural boundaries around Beijing that would create a relatively small area but also contain most of the dataset.

Figure 4.4 shows the street view of downtown Beijing. From the center of the city there are seven highways that radiate out from the center. We chose the outer highway loop as a natural boundary for our dataset. The area enclosed by this boundary is less than 50 km^2 with a perimeter of 187 km and contains 75% of the original dataset.

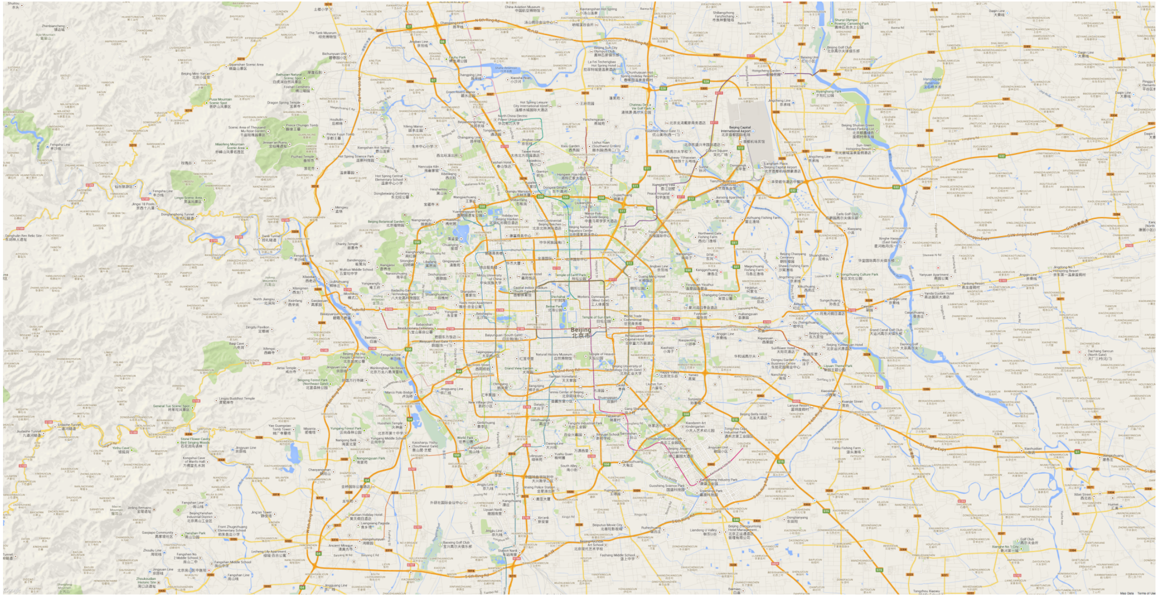


Figure 4.4: Downtown Beijing (Scale 1:200,095)

Next, we created a polygon geometry that described our highway boundary. We then computed the intersection of the boundary with all of our data points to remove any points outside of this boundary. The following four figures show the progress of our approach to cleaning the dataset: Figure 4.5 shows the original point data; Figure 4.6 shows the boundary; Figure 4.7 shows the original point data and the boundary; and, Figure 4.8 shows the filtered data based on the boundary.

Figure 4.5: Downtown Beijing with Original Dataset (Scale 1:200,095)

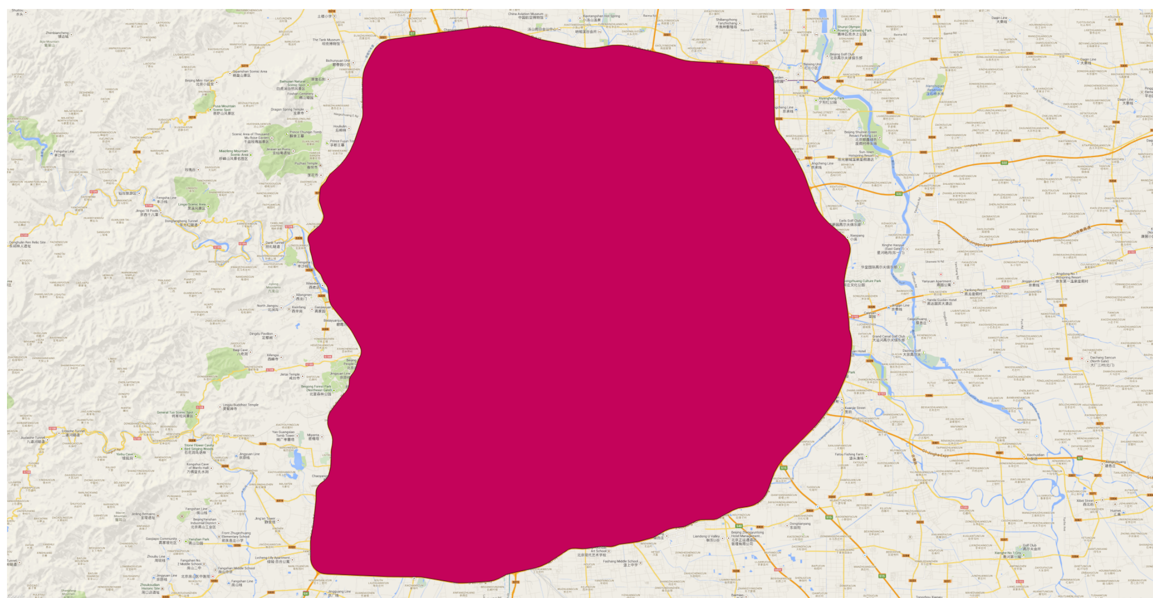


Figure 4.6: Downtown Beijing with Boundary (Scale 1:200,095)

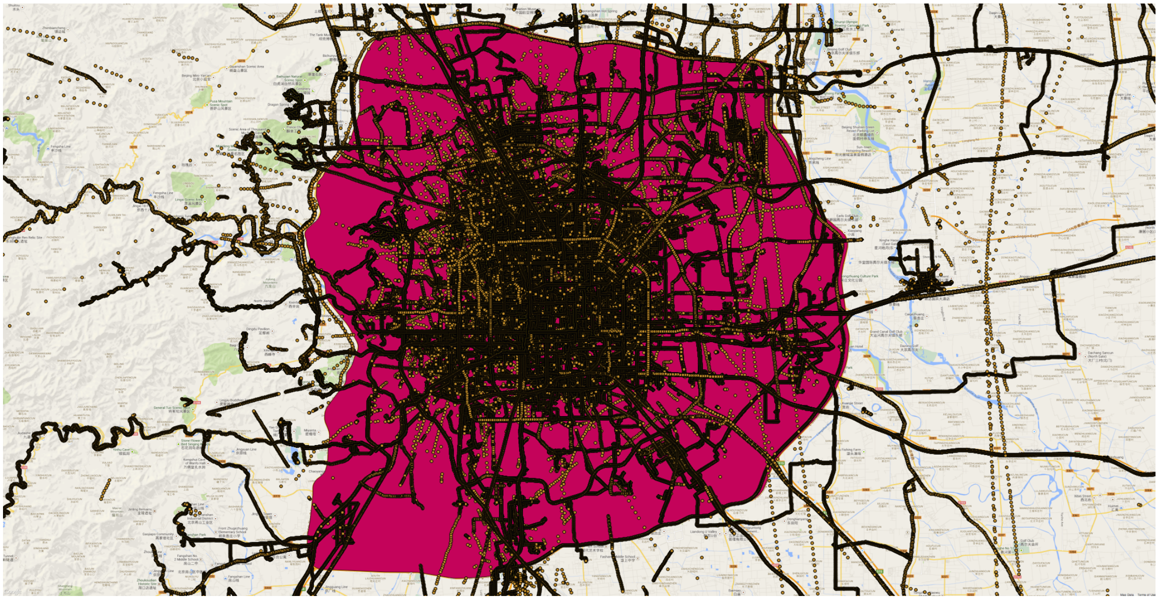


Figure 4.7: Downtown Beijing with Original Dataset and Boundary (Scale 1:200,095)

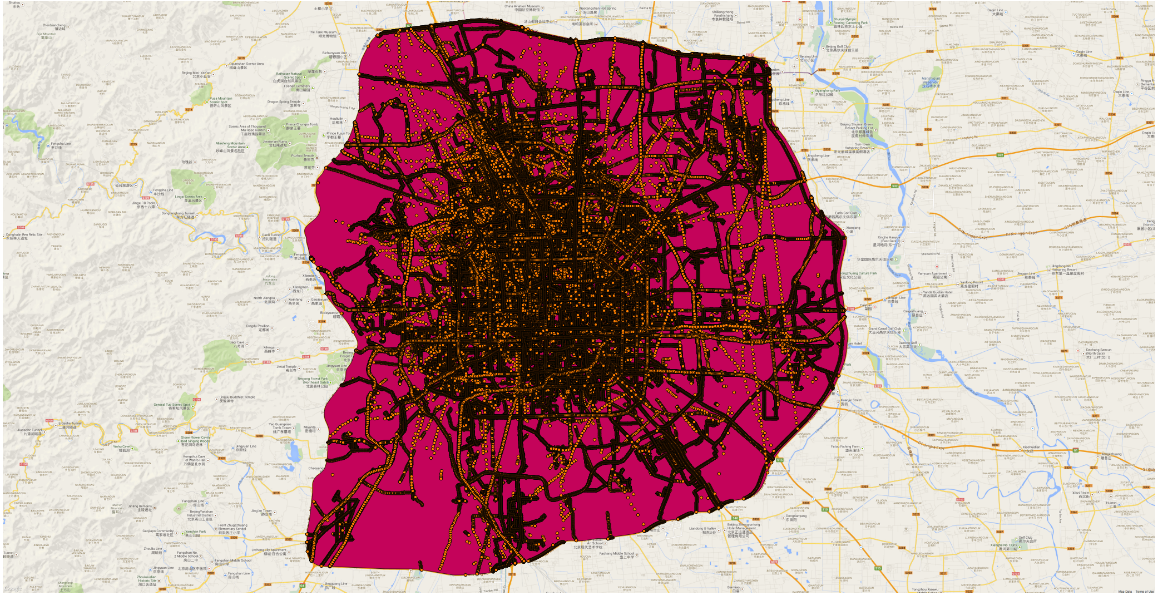


Figure 4.8: Downtown Beijing with Boundary and Filtered Dataset (Scale 1:200,095)

The boundary and the filtered dataset were all stored in the PostgreSQL database in separate tables. Figure 4.9 shows the updated schema. The boundary and boundary_point tables were created so that, if required, we would be able to create new boundaries with

their own respective points while maintaining data from pervious experiments. Although no new boundaries were necessary for our experiments, the current schema certainly supports this capability. Figures 4.5-4.8 were generated by retrieving data from this schema and demonstrated that our dataset import and filter were working well.

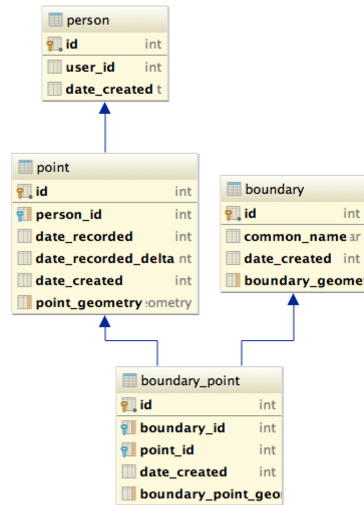


Figure 4.9: Updated Schema to Include Boundary and Filtered Points

4.2.2 What Is a Trajectory?

In the context of our experiments we define a trajectory as follows:

Definition 1. A trajectory is a set of points whose beginning and end map to a user goal: a user goal being something the person is intending on doing such as watching a movie. The first point in a trajectory defines their current position and the last point defines the last position and the attainment of some kind of goal.

As a concrete example of our definition, suppose you just finished watching a movie and you want to find a place to eat. In GPS terms, your current position represents you walking out of the theatre and your final position represents you arriving at a restaurant.

In the GeoLife dataset there is no other context other than the GPS points that tell us directly what a user was doing; however, we can try to infer this by looking at the time deltas between points. It is reasonable to assume that one of the reasons why there is a large delta between points is that person arrived at their intended destination such as the movie theatre or restaurant. Once inside the recording of GPS points stops and does not

start again until they have completed that activity and begun moving again. It is these transitions that we want to identify by looking for large time deltas in our dataset.

The next obvious question is: what to choose for this maximum time delta? Here we applied a bit of intuition and statistical analysis. For our experiment we decided to use 30 minutes since anything that takes longer than 30 minutes to complete is probably what we would consider a goal such as watching a movie or eating at a restaurant. Statistically speaking, when looking at the density of all time deltas our maximum time delta should fall in the region of lower density. The reason for this is that the number of time deltas that we would consider to be transition points is going to be much lower than the density of points while a person is moving. If our maximum time delta does not fall into that region then we may be creating separate trajectories, identifying separate goals, when in fact that is not the case.

To confirm that our maximum time delta falls into the right density region we applied the Kernel Density function to the time deltas for all of our points [Sil86]. A Kernel Density function is useful because it helps us visualize the distribution of one variable; in this case time delta. A common alternative to the Kernel Density is to use a histogram; however, the selection of the bin size can have a dramatic effect on the shape of the plot and we therefore chose not to use it.

Figure 4.10 shows the Kernel Density function for the 99th percentile of our dataset. From this plot it is clear that most of the density exists at two seconds followed by one and five seconds. Values higher than five can be considered as outliers and good candidates for creating separate trajectories. These results validate choosing 1800 seconds as a reasonable maximum time delta. The R source code for computing these Kernel Densities can be found in Appendix A.2.

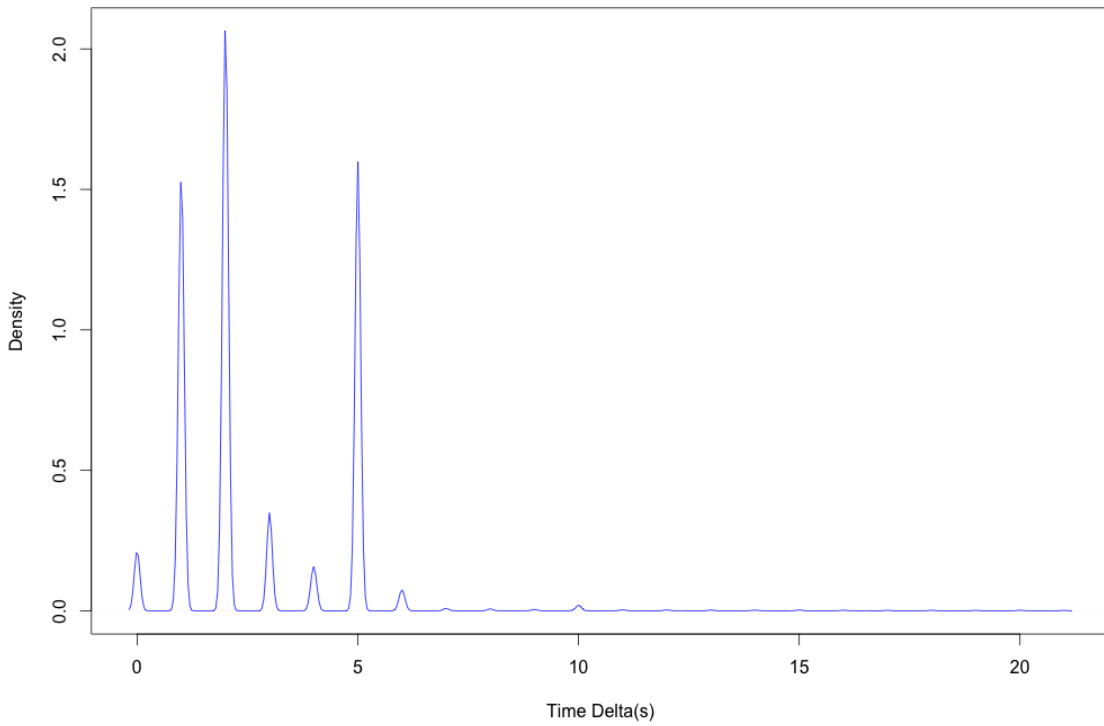


Figure 4.10: Relative Kernel Densities of Time Deltas Between Points 99th Percentile

Using 30 minutes as our maximum time delta we computed the trajectories for each user. Appendix A.4 contains the Python source code used for computing our trajectories. The TrajectoryBuilder class has three functions: __build__user__ (lines 17-159), build (lines 161-263), and crop (lines 268-282). The work for computing trajectories for each user is an independent task therefore we started a new process for each user in order to take advantage of as many cores as possible and speed up our computation.

The build function is responsible for managing all currently running computations and maximizing the load on the system. Lines 228-243 show how this is implemented. This code scales up to a machine running 182 CPUs since there are 182 users for which trajectories need to be computed. The __build_user__ function is what actually does the trajectory computation. The core logic for deciding whether a point is part of a current trajectory or a new trajectory is controlled by lines 52, 54, and 56.

Once the end of a trajectory is reached it is saved to the database (lines 70-85). Similar to the GPS points, the trajectories were stored using a geometry data type but instead using a projection with SRID 32650. Geographic coordinate systems and projections are in and of

themselves models for drawing a point on a map and as such some projections/models are better than others under different situations. SRID 4326 was a good choice for storing the projecting of our original GPS points because 4326 uses latitude and longitude to describe a position; and, that was the position description in the original dataset. One drawback to 4326 is that it can sometimes distort data at small scales and it also makes it more difficult to work with when you try to, for example, calculate the distance between two points and you get a result in degrees.

The Universal Transverse Mercator (UTM) coordinate systems slices the globe into 60 longitudinal strips, projection zones, with each zone being projected onto its own plane as depicted in Figure 4.11. The result of this approach is that scale distortion is minimized within each zone. In addition, position is measured in meters using Easting and Northing relative to the UTM zone. Luckily Beijing fits perfectly into UTM zone 50 so to take advantage of the higher precision of this coordinates system we stored all trajectories, as well as the boundary, in UTM zone 50 using the SRID 32650.

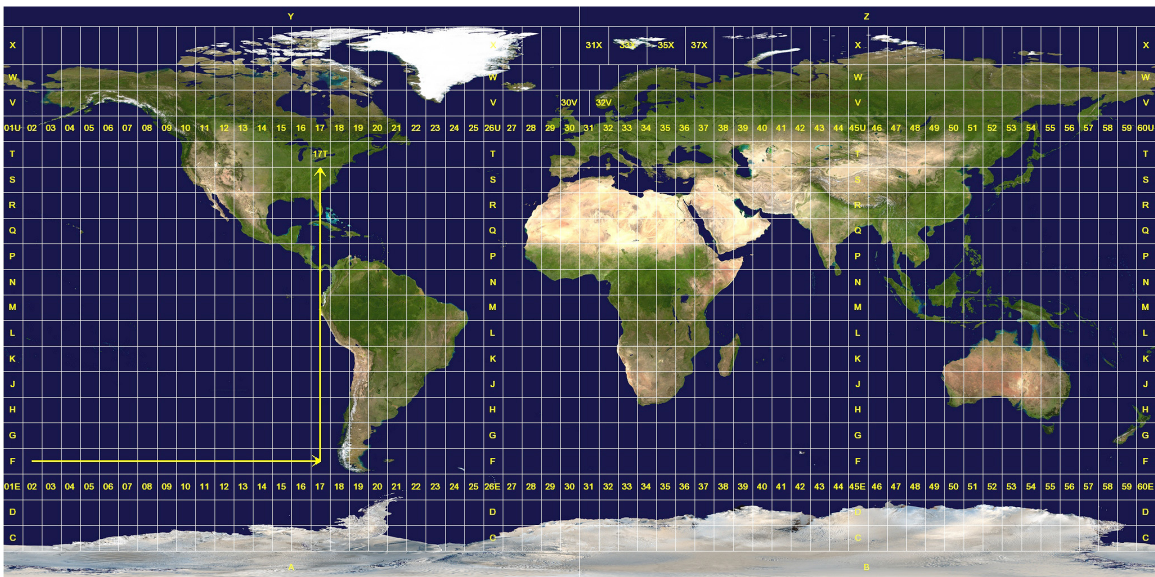


Figure 4.11: UTM Zones (Courtesy Wikimedia Commons)

To accommodate the trajectory data as well as the ability to modify the maximum time delta threshold for different experiments we extended our existing schema as seen in Figure 4.12.

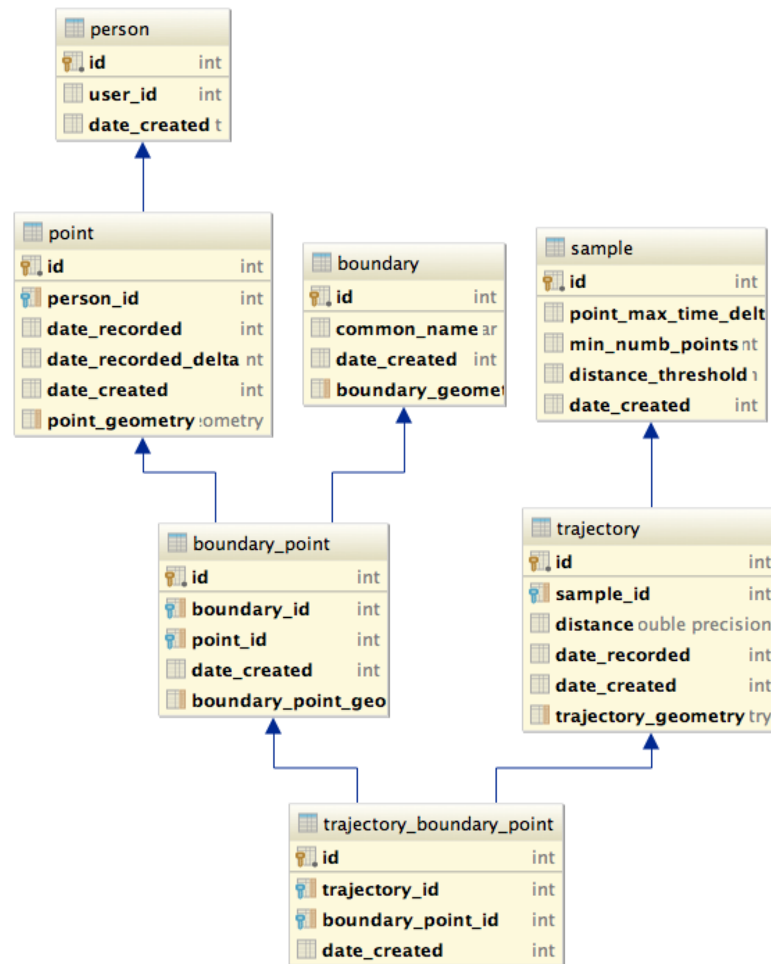


Figure 4.12: Updated Schema to Include Trajectory and Sample

Once we created the trajectories we used qGIS to visually verify our trajectories. The results are depicted in Figure 4.13. To our surprise there appeared to be several very long trajectories that zig zagged across the map. Given that the boundary area is approximately 50 km^2 we did not expect to see trajectories cutting across the entire map let alone ones that were straight lines. What we did expect to see was trajectories that roughly followed the shapes of the road which as can be seen in the figure was the case most of the time.

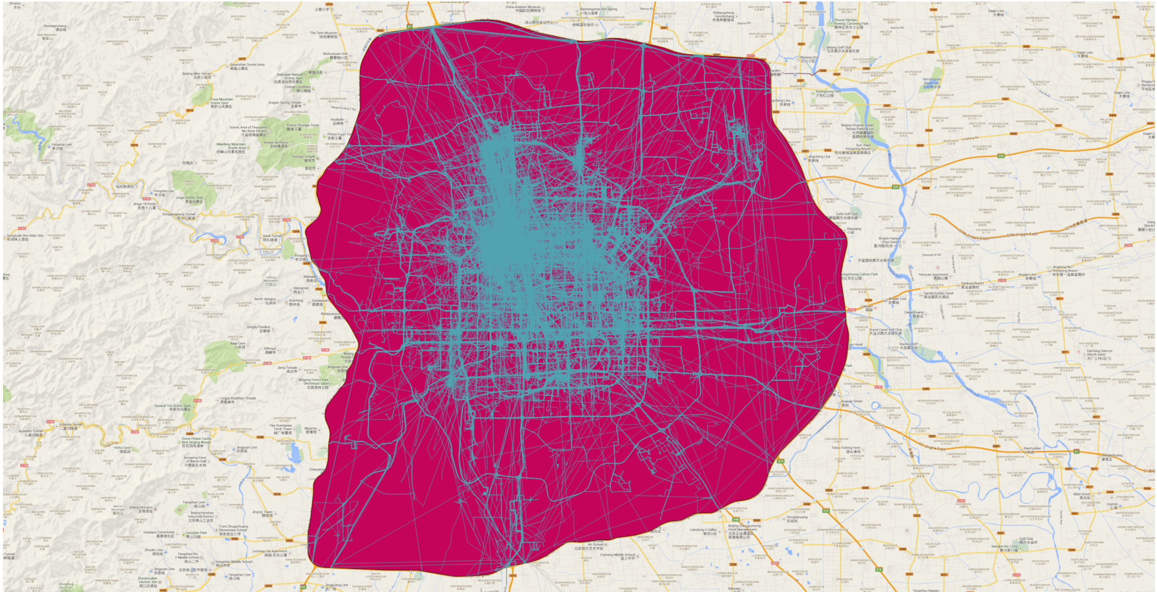


Figure 4.13: Downtown Beijing Trajectories

To cleanup this data we decided to ignore trajectories that were longer than 15 km. We arrived at this number again using a combination of intuition and statistical analysis. The target application for this kind of modelling is Yakkit and since Yakkit is all about providing you with information that is nearby we only care about modelling data that describes entities that are within walking distance or a short drive away. To ensure that this 15 km limit was not removing a large number of our trajectories, and possibly rendering our results statistically invalid, we computed the Kernel Density for all the data as well as the 99th percentile. The crop function in Appendix A.4 was applied to flag trajectories that were longer than the 15 km limit (lines 278 and 284).

Figure 4.14 show the Kernel Density results. We also computed a local maxima to identify highest distance density of 2 km (i.e., the most frequent trajectory distance). This distance confirms that our choice of our maximum time delta was reasonable since it produced most trajectories in the 2 km range which is a reasonable distance for our application scenario. These distributions also confirmed that our choice of 15 km was a reasonable cutoff that would not remove a large part of our dataset. In absolute terms a 15 km cutoff kept 18,107 out of the 23,707 total trajectories or 76.4%. Figure 4.15 shows the resulting trajectories after removing trajectories longer than 15 km. The R source code for computing these Kernel Densities can be found in Appendix A.3.

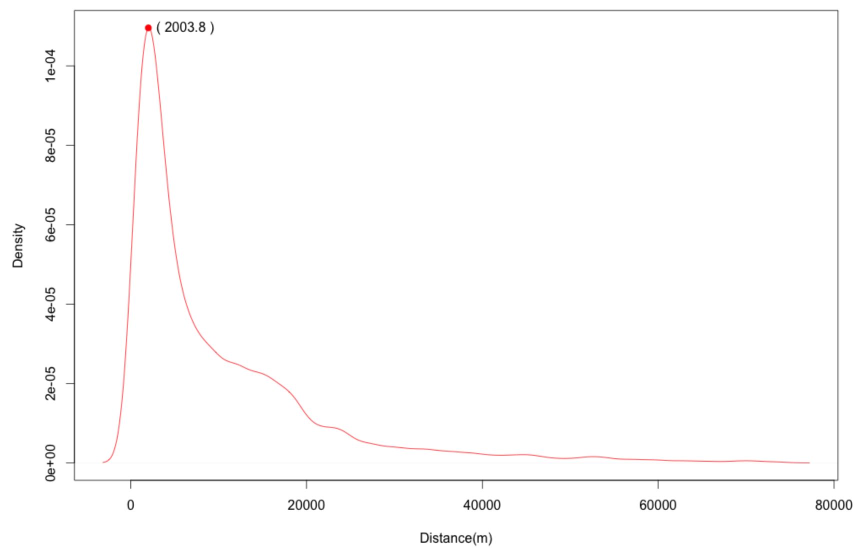


Figure 4.14: Kernel Density of Trajectory Distances 99th Percentile

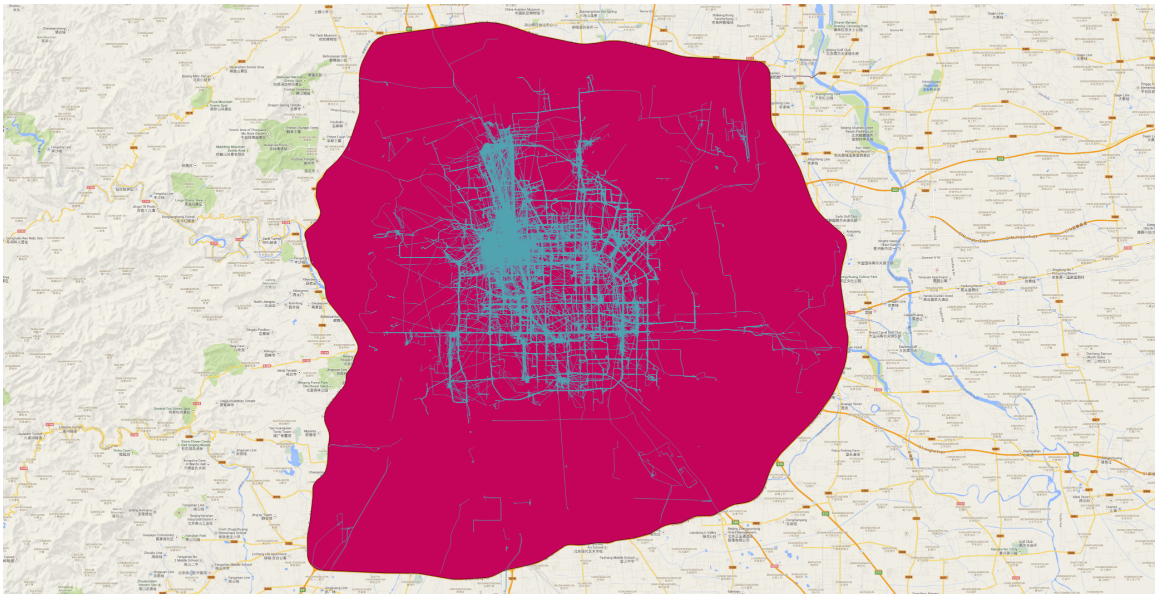


Figure 4.15: Downtown Beijing Trajectories with Length Less Than 15 km (Scale 1:200,095)

4.2.3 Modelling and Classification

The basic idea behind our approach was to divide the entire boundary area into a grid of sectors. For each sector we computed a probability distribution, or model, that identified destination sectors based on the trajectories that started in each sector. Several different sector sizes were used ranging from 1,000 m to 4,000 m. Figure 4.16 shows an example of what the grid looks like using a 6,000 m sector size. Different sector sizes were used to see what effect changing the sector size had on the outcome of the classification. The steps for modelling each sector were as follows:

1. Fetch all trajectories whose start point exists inside the current sector: source sector.

2. For each trajectory in the source sector determine the destination sector.

3. For each destination sector count the number of terminating trajectories.

4. Once all destination sector counts have been computed store the probabilities as a model for the source sector

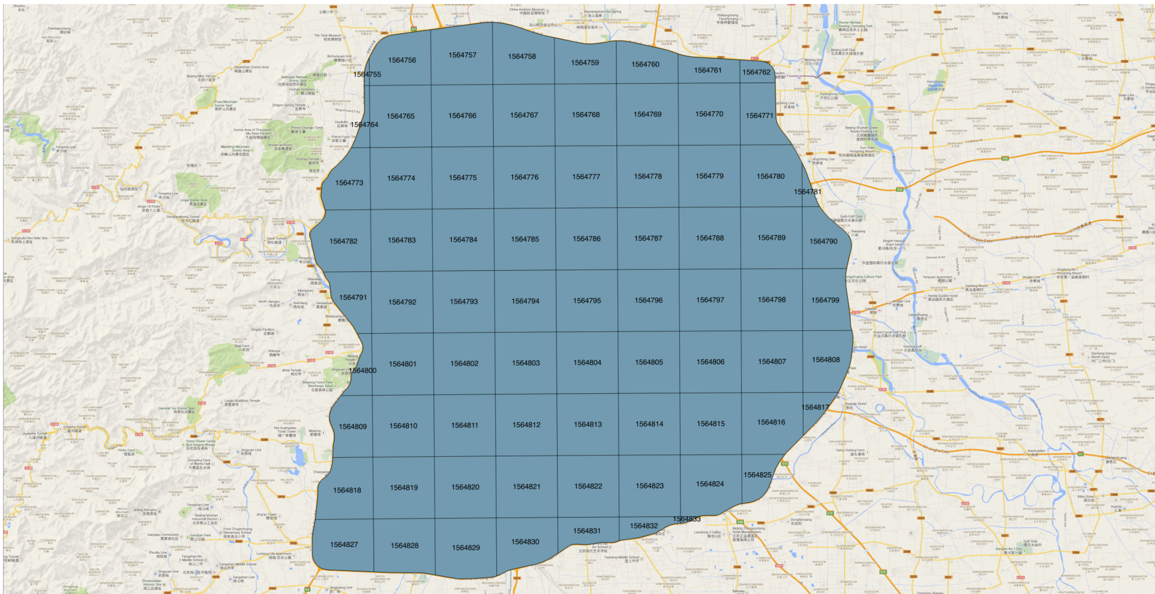


Figure 4.16: Downtown Beijing with 6,000 m Sectors (Scale 1:200,095)

To help illustrate the modelling process consider the scenario below where we model one specific sector for a 1,000 m sector size. In this work sector size refers to the length of one side of a sector. Figure 4.17 shows our starting sector 1,175 highlighted in blue. The remaining sectors for this sector size are shown in transparent blue with the map showing underneath.

Next we fetch all the trajectories, red lines, whose start points intersect with sector 1,175 as shown in Figure 4.18. We then determine the destination sectors, orange squares, for all these trajectories as shown in Figure 4.19. Finally, we compute the probability for each destination sector and store the results as a model for sector 1,175. Table 4.1 shows the model for this specific sector. During classification when we are trying to predict a destination for a user that started in sector 1,175 we would choose the destination sector with the highest probability: 23.3% which in this case happens to also be the starting sector. This entire process is repeated for each sector in the entire grid.

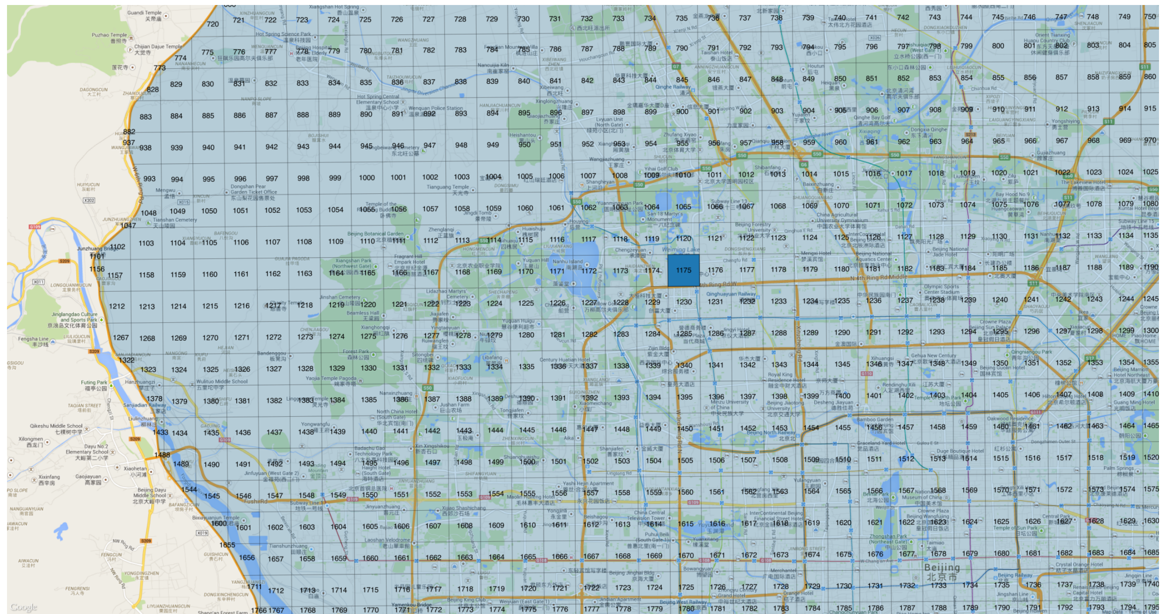


Figure 4.17: Source Sector 1,175 With 1,000 m Sectors In Background (Scale 1:65,000)

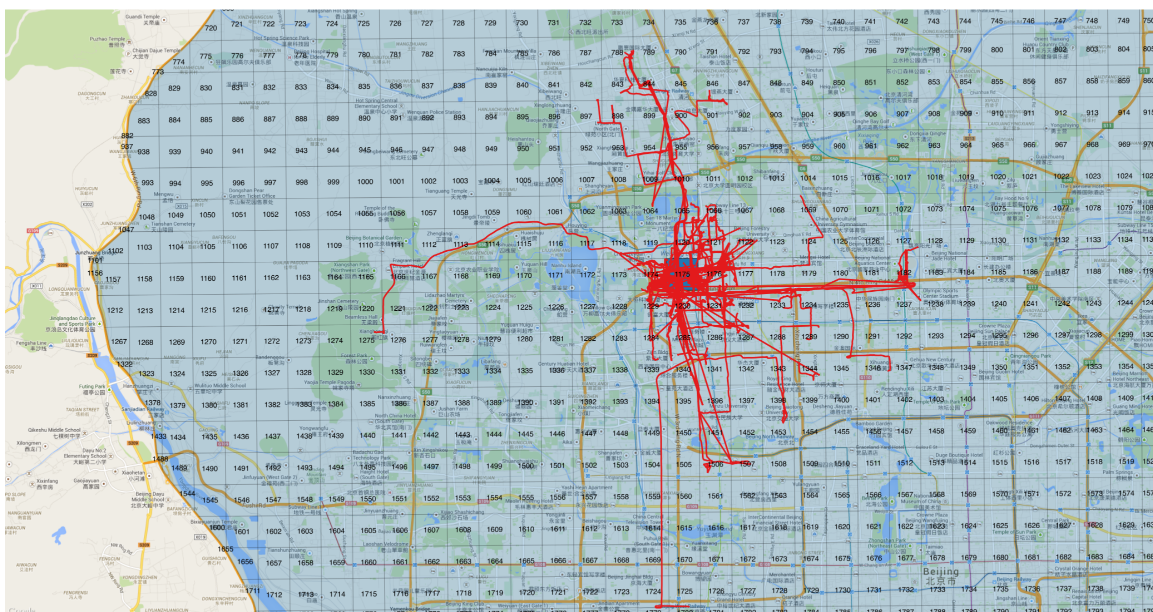


Figure 4.18: Source Sector 1,175 Trajectories With 1,000 m Sectors In Background (Scale 1:65,000)

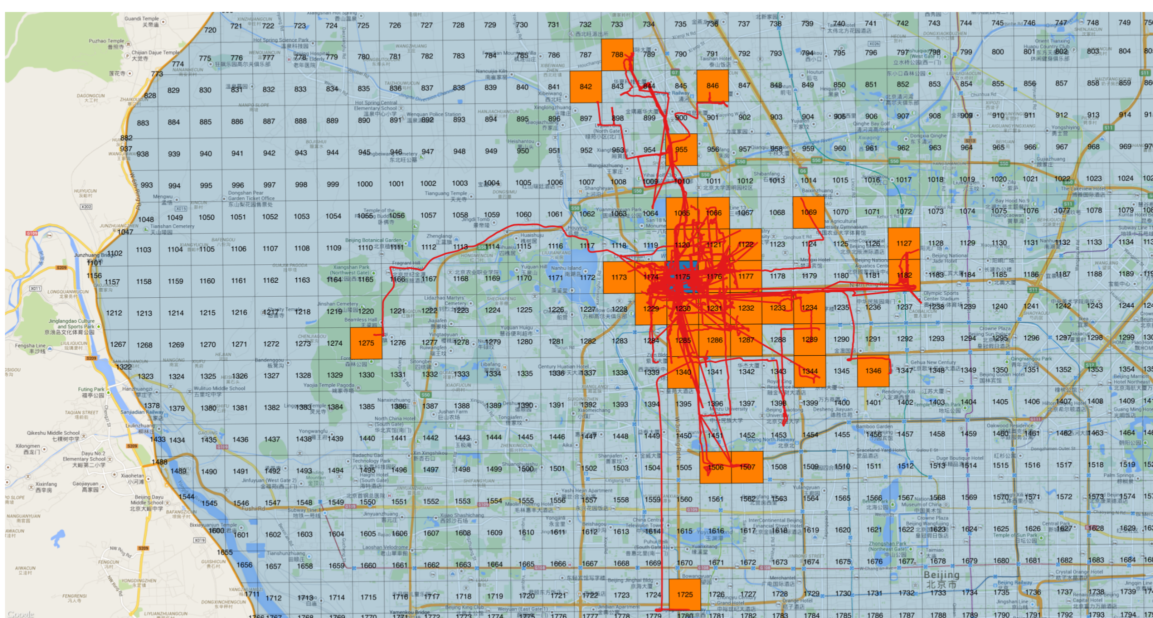


Figure 4.19: Source Sector 1,175 Destination Sectors With 1,000 m Sectors In Background (Scale 1:65,000)

Table 4.1: Sector 1,175 Model

Source Sector	**Destination Sector**	**Probability (%)**
1175	1175	23.3
1175	1174	21.9
1175	1232	8.15
1175	1176	7.65
1175	788	5.06
1175	1230	4.84
1175	1182	3.26
1175	1287	3.26
1175	1065	2.81
1175	1121	2.75

4.2.4 The Experiment

We conducted 26 different experiments using different sector sizes as well as different combinations of sector sizes. The sector size affects what we are able to say in terms of user behaviour. For example, if we use source and destination sector sizes of 100 m then we can possibly say something about the block or store level: users that start on street A have a 33% chance of ending on street B. If we use source and destination sector sizes of 1,500 m then we can possibly say something at the city or district level: users that start in Beijing Botanical Gardens have a 33% chance of ending up in the business district. Finally using a combination of source and destination sector sizes, 1,500 m and 100 m respectively, we could say something like users that start in China town have a 43% probability of ending on street B.

For each combination of sector sizes we used 10-fold cross validation for testing our classifier. At the beginning of a run all the trajectories were shuffled using the Fisher Yates algorithm and split into 10 buckets: lines 18 to 55 in Appendix A.5. Each experiment was run 10 times. Each time $\frac{9}{10}$ of the buckets were used for creating the model (lines 129-196) while the remaining bucket was used for prediction (lines 303-392). Figure 4.20 shows our final schema including tables for storing grid sectors, probabilities, and our experimental results.

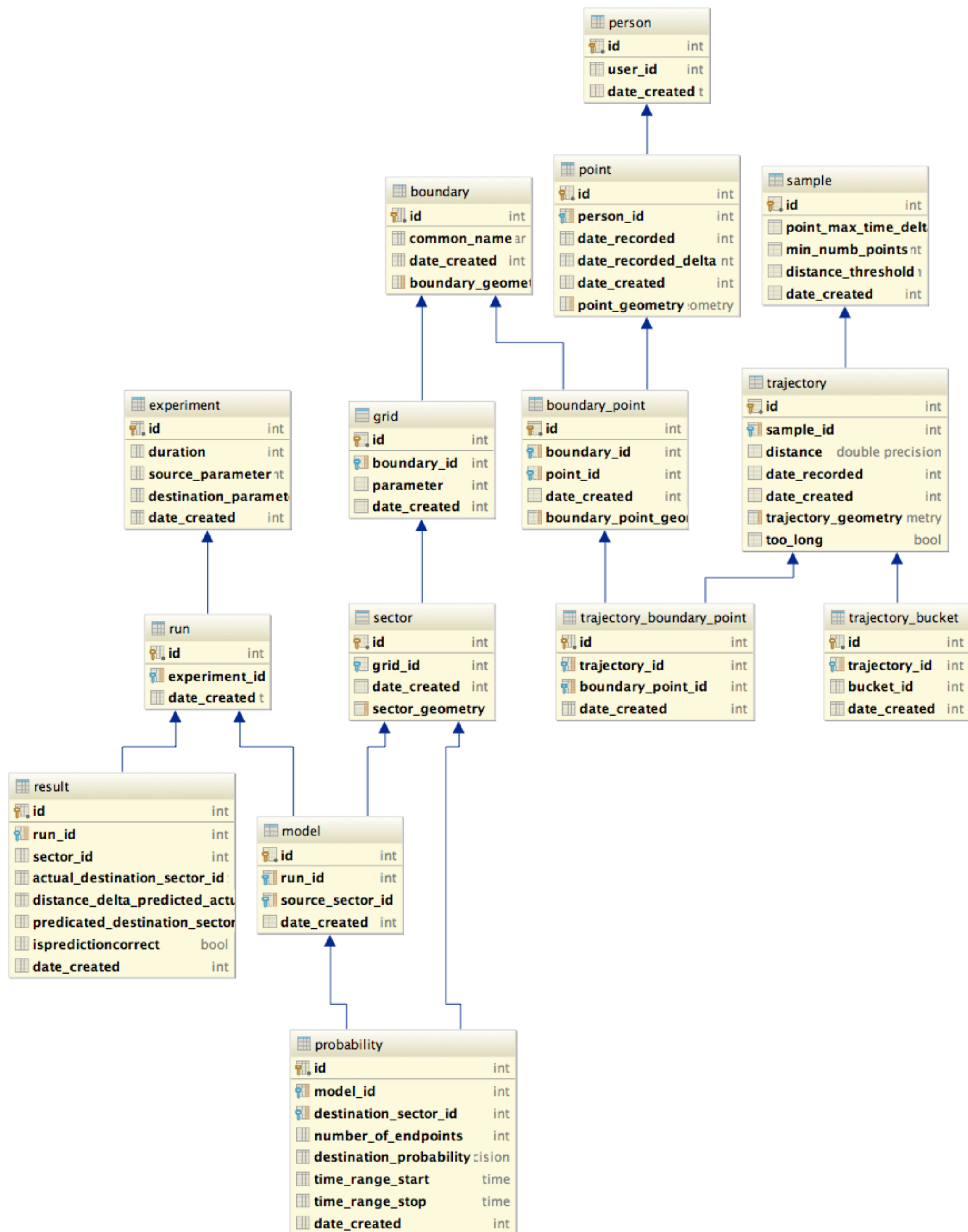


Figure 4.20: Final Schema

4.3 Experimental Results

Four different categories of experiments were run: Symmetric-Original, Symmetric-Shifted, Asymmetric-Original, and Asymmetric-Shifted. The symmetric experiments used source and destination sectors of equal size. The asymmetric experiments locked the source sector at 1,000 m and varied the destination sector size between 1,500 m and 4,000 m. The shifted versions of the symmetric and asymmetric experiments shifted each sector by a factor of $\frac{1}{2}$ of its size relative to its origin. The purpose of the shifted experiments was to evaluate the effect of sector boundaries on classification.

Figure 4.21 shows a summary view for the Symmetric-Original and Symmetric-Shifted experiments. The horizontal red line in each graph indicates the minimum success that must be achieved by the classifier. For sector sizes 1,000 m, 1,500 m, and the Symmetric-Original version of 2,000 m the classifier failed to achieve this minimum standard. We achieved best results using the 3,500 m sector size during the Symmetric-Original experiment which correctly predicted destinations 85.6% of the time. We achieved worst results using the 1,500 m sector size during the Symmetric-Original experiment which correctly predicted destinations 36.5% of the time.

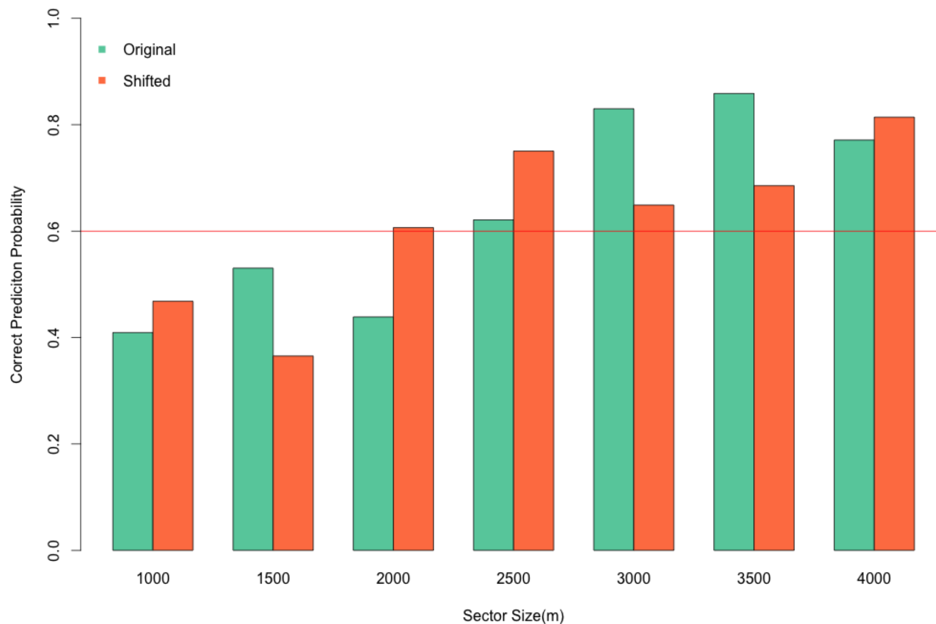


Figure 4.21: Symmetric Original and Shifted Experiment Results - Side by side comparison of original and shifted experiments showing the effect of sector boundaries on classification.

Figure 4.22 shows a summary view for the Asymmetric-Original and Asymmetric-Shifted experiments. As before, the horizontal red line in each graph indicates the minimum success that must be achieved by the classifier. The horizontal blue line indicates the success rate that was achieved for the 1,000 m symmetric experiments and helps illustrate the relative change in success rates using the symmetric and asymmetric versions of these experiments. We achieved best results using the 3,500 m sector size using the original version of the destination sector which correctly predicted destinations 77.5% of the time. We achieved worst results using the 1,500 m sector size using the shifted version of the destination sector which correctly predicated destinations 32.1% of the time.

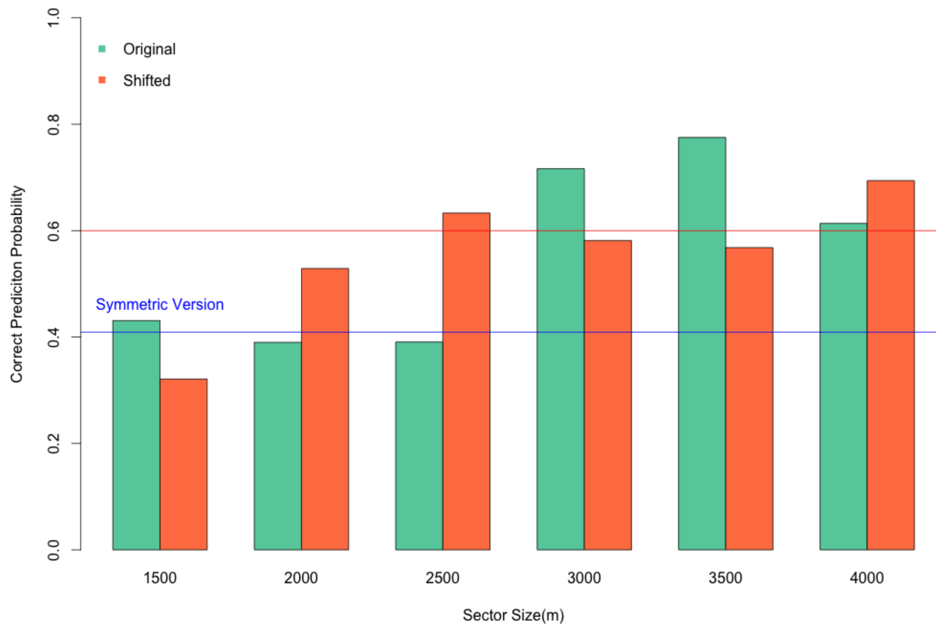


Figure 4.22: Asymmetric Original and Shifted Experiment Results - Side by side comparison of original and shifted experiments showing the effect of sector boundaries on classification.

Figure 4.23 shows the Kernel Density of the distance error for incorrectly predicted destination sectors as a fraction of sector size. This data includes results for all 14 symmetric experiments for original as well as shifted values. By inspection, most of the false predictions were off by one sector size. Figure 4.24 shows the same graph but this time using the results from the 12 asymmetric experiments. Similarly to the symmetric set of experiments the incorrectly predicted sectors were only off by one sector.

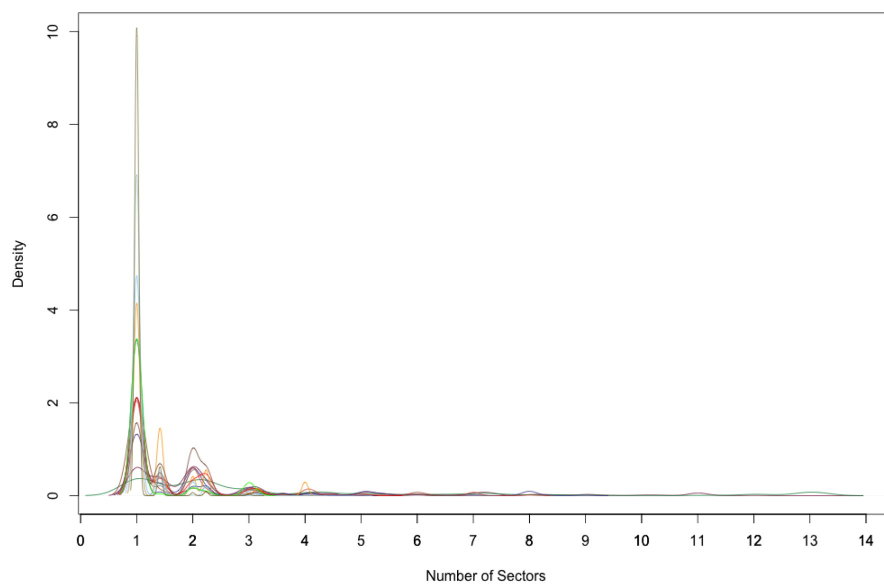


Figure 4.23: Kernel Density of Number of Sectors Error as a Fraction of Symmetric Sector Size for False Predictions (Original and Shifted)

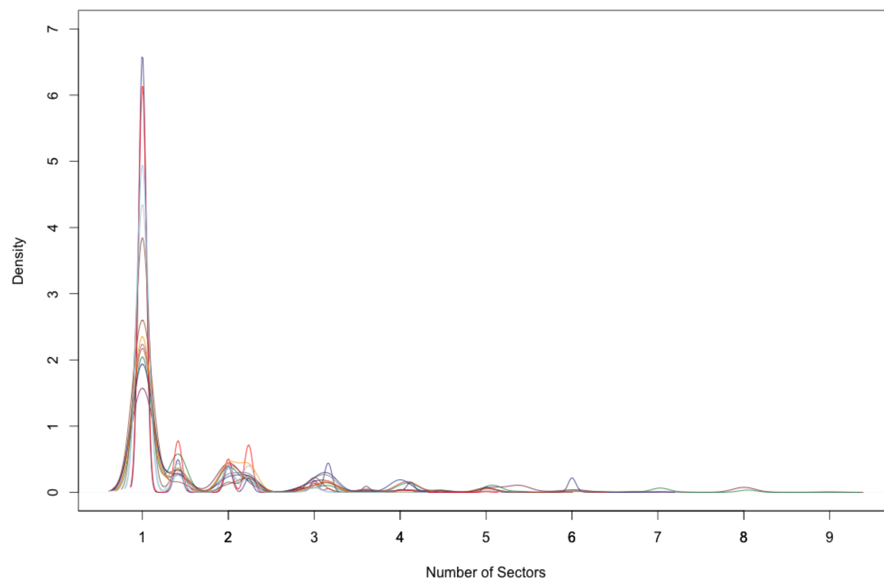


Figure 4.24: Kernel Density of Number of Sectors Error as a Fraction of Asymmetric Sector Size for False Predictions (Original and Shifted)

4.4 Discussion

The general trend in almost all the experiments was an increase in prediction success with an increase in sector size. Intuitively this make sense: you tend to catch more fish when you cast a wider net. In the Symmetric-Original and Symmetric-Shifted experiments we did not attain usable results until the sector size was increased to 2,000 m. The effect of shifting sector boundaries had an effect on outcome especially when you consider how close some experiments came to hitting the 60% threshold. Table 4.2 shows the relative difference, as a percentage, between the original and shifted versions of the experiment for each sector size.

Table 4.2: Delta in Prediction Success Between Symmetric Original and Shifted Experiments

Sector Size (m)	**Success Prediction Delta (%)**
1000	5.9
1500	-16.5
2000	16.8
2500	12.9
3000	-18.1
3500	-17.3
4000	4.3

In part, it was these results that prompted us to run the asymmetric experiments. During the asymmetric experiments we held the source sector size constant at 1,000 m and varied the destination. 1,000 m was chosen since it was the smallest sector size we used and we wanted to see if we could hit the 60% target. In $\frac{9}{12}$ of the experiments this approach outperformed the symmetric version and in $\frac{5}{12}$ of the experiments the 60% mark was attained. Of course, this performance improvement came at a cost since in order to get it we had to increase the size of the destination sector and this in turn limits us to what we can say about this model: users that start on street A have a 43% chance of ending in China town. As with the symmetric experiments we also computed the results for the shifted versions. Table 4.3 shows the relative difference, as a percentage, between the original and shifted versions of the experiment for each destination sector size.

Table 4.3: Delta in Prediction Success Between Asymmetric Original and Shifted Experiments

Destination Sector Size (m)	**Success Prediction Delta (%)**
1500	-11.0
2000	13.9
2500	24.2
3000	-13.5
3500	-20.7
4000	8.0

Although our classifier and model present useful results they did not yield good enough results for the kind of resolution that we were hoping for. In the case of Yakkit we were hoping to find this approach useful at a sector size of 50 or 100 m; however, performance was already below the 60% threshold at a sector size of 1,000 m. Since we did not attain meaningful results until the 2,000 m sector size this classifier and model on their own are not enough to filter information at the level we are hoping to.

As mentioned earlier, the results are still useful. Using the classifier at a sector size of 2,000 m and above we could claim to be able to filter chat messages and advertising at a district level. For example, we would be able to classify a user heading to Central Park in New York and we could use that information to filter chat and advertisements only from that area. The effect of varying the sector boundaries and the fact that we were only off by one sector in most cases leaves us hopeful that with more work the resolution of this approach can be increased.

4.5 Threats to Validity

The decision to use the outermost highway around Beijing as a boundary may have removed valid trajectories and created an unreasonable expectation for what a boundary should be. In a real world scenario this would be a justifiable problem since we would arguably have more data and the decision for what a boundary is would be much more difficult. In our case we happen to be lucky in that the most dense part of the GeoLife dataset falls within our boundary. We consider the investigation into boundary sizes outside of the scope of this thesis and part of future work.

The choice of maximum time delta and maximum trajectory distance for creating and cropping trajectories directly affects our models and results. Changing this value can increase or decrease the number of resulting trajectories as well as affect our ability to claim that a trajectory represents a user goal. Similarly, cropping trajectories based on length

affects the results and indicates, perhaps incorrectly, that goals supported by long trajectories are not relevant. To mitigate the potential effects of choosing these values poorly we computed Kernel Density functions to ensure that the values we chose did not accidentally remove large portions of our dataset.

The approach of looking only at start and stop positions of a trajectory may be overly simplistic: "It is good to have an end to journey toward; but it is the journey that matters, in the end" – Ernest Hemingway. Although this is true, our results indicate our approach is useful when trying to make predictions at a larger scale: city or district. This means our approach can be integrated into Yakkit for basic message filtering or applied to totally different applications such as city traffic management.

As first mentioned in Section 4.4, the boundaries created by the sectors affect our ability to correctly classify destinations. When applied to different datasets for different regions the resulting boundaries may generate worse results. Although we do not present a solution to this problem we ran shifted versions of our experiments to show how big of an effect these boundaries had on our results. In addition, the fact that in cases where our classifier was wrong it was only wrong by one sector size provides even more evidence that more work is required for deciding how to lay out the sectors.

4.6 Summary

This chapter described our approach and presented experimental results for a model and classifier based on geography and trajectories. Instead building models directly using points and trajectories, as in previous works [ZZXM09b] [ZZXM09a] [YZXW10], we first segmented our region of interest into different sized sectors. Using these sectors we built a model for each sector that described the probability of ending up in a destination sector given the current sector. We tested our approach using different sector sizes and found that are results are not ideal for predictions on a scale required for the Yakkit application. Nevertheless, our results are still useful as they can be used for predicting movements on a larger scale such as user movements between different districts of a city.

Chapter 5

Yakkit Service Deployment and Latency

In Section 2.3.3 we discussed latency as a measure of performance. Latency can always be broken down into smaller parts and each latency part determines the tools that can be used to measure it as well as what solutions can be applied to minimizing it [JRM+10] [CWSR12] [CCT+11] [WN10]. In the case of Yakkit Chat, latency is the time difference a user sending a message and all the nearby users receiving that message. In this scenario there are two factors that contribute to the total latency experienced by the user: network latency and processing latency.

Since we are measuring and comparing the round trip times of messages on a computer network we used the ping-pong approach to collect our data [WN10]. The final contribution of this thesis comes in the form of experimental results obtained through emulation that show the effect of distance on application latency. The goal of this experiment is to help answer our third and final research question: does deploying location based chat services closer to a user have a positive or negative effect on message latency?

5.1 As Fast as Possible

In a CB radio system, or any radio system for that matter, geography and distance play a central role in the user experience. In cases where the radio system uses line of sight, as in the case of most CB systems, a geographical obstruction like a mountain may prevent communication. In other situations, when communication happens over long distances the quality of the transmission may degrade resulting in dropped connections or long delays. Yakkit Chat is susceptible to the same kind of interference although for different technical reasons. Since servers are used to route messages between users, the location of the server

can play a role in the quality of the experience. Just as using radio over long distances may degrade the experience so too can the quality of Yakkit Chat if the supporting servers are far away from the users.

One of the goals of the Yakkit Chat Service is to provide an experience as close to reality as possible. What this means is that when sending messages using the Yakkit iPhone application via the Yakkit Chat Service, users should feel as if they were standing right in front of someone and talking to them directly. In addition, as a consequence of Yakkit's geographical nature, users in a particular location will only ever interact with other users in that same location. As such, the Yakkit Chat Service can be deployed at multiple locations at once without the need for any kind of synchronization. In theory, services that are deployed closer to the user should result in lower latencies and a better user experience.

Although the goal of this experiment was to observe the effect of distance on latency and not hit a particular latency target, it is important to have a sense of proportion when discussing latency from an application point of view. For a user to perceive a system reacting "instantaneously" the latency needs to be at most 0.1 seconds. If a system reacts between 0.1 and 1.0 seconds the user will perceive the experience as uninterrupted although they will notice a delay. Finally, if a system reacts anywhere between 1.0 and 10 seconds the user will certainly notice an interruption and needs to be presented with some kind of progress bar or status indicator [Mil68]. In the case of Yakkit the goal is to achieve latencies on the order of 0.1 to 1.0 seconds.

5.2 Experiment Setup

To observe the effects of distance on latency we set up an experiment spanning the North American continent using SAVI and Amazon infrastructures. In total, five services were deployed at four different locations running three different types of services. Figure 5.1 shows a map with the four locations of the servers. The Yakkit Chat Service was deployed on the Victoria and Carleton servers. In addition, the Victoria server also contains the Registrar Service. The Amazon EC2 Oregon and Amazon EC2 Virginia servers ran bots that simulated Yakkit Chat usage and collected data. Each experiment was run five times for a duration of 150 seconds with a 20 second ramp up time.

Figure 5.1: Server Locations Latency Experiment

The purpose of the Registrar Service was to listen for incoming connections from new bots posing as Yakkit Chat clients as well as for incoming connections from servers running Yakkit Chat Services. When a new client connected to the Registrar Service would look at its list of currently running Yakkit Chat Service servers and determine the closest Yakkit Chat Service using Euclidean Distance as shown in Equation 5.1. Additionally the Registrar Service notifies clients of any changes to the available Yakkit Chat Services. In this way the clients are always connected to the closest possible instance of the Yakkit Chat Service as new instances are booted or taken offline to simulate node failure.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{5.1}$$

The purpose of the Yakkit Chat Service was to route messages based on the list of currently connected clients for that service instance. The service maintained a k-d tree data structure with the current positions of all its clients as well as a list of open connections to those clients. When a new message arrived the service would do a lookup in the k-d tree to find other nearby clients, if any, and forward the message using their respective connections. This service was duplicated on both the Victoria and Carleton servers.

To help with testing we created and deployed a Bot Service at two locations. The purpose of the Bot Service is to emulate users connecting and sending messages from a similar location. The Bot Service is composed of five classes which can be found in Appendix A.1. The BotManager (lines 242-276) is responsible for starting and stopping bots during the experiment.

Since we did not want to introduce potential latencies due to system load, we ran the

emulation using 10 users, emulated by bots, initiated from the exact same location. Each Bot (lines 186-239) was started by the BotManager as a separate process. Each bot used the asynchronous YakkitClient class (lines 43-140) to send and receive messages. In addition, each Bot started a separate process using the Ping class (lines 143-183) which recorded ping times during the experiment.

The purpose of collecting ping data was to help further clarify whether latency was coming from the network or if it was inherent in the system. Use of separate process and asynchronous libraries for the bots was a design decision to help minimize the chance of causing any latency from the experiment itself.

We implemented message passing, and all communications, using WebSockets in order to minimize protocol level latencies that may exist such as in the case of HTTP where a handshake needs to occur each time a request is made. Messages were randomly selected from a list of 15 at a random interval between 0 and 20 seconds (lines 199-214 and 232). Message length was between 26 and 77 characters. These message lengths represent a reasonable length for what one might see in a text chat conversation.

When a message is sent out a creation timestamp is attached (line 236). When a message arrives at the receiving end the transit time is calculated (line 87) based on the delta between creation and arrival times. This data is then saved in a database for offline analysis (lines 92-94). The ping process used the Python subprocess model to run the ping command and pipe the output back into the application (lines 167-175). This output is then parsed and saved in a database for offline analysis (lines 178-181).

5.3 Experimental Results

To observe the effects of distance on latency we varied the servers to which users are connected. Figure 5.2 depicts the scenario where we varied users connected to the geographically closest server. Figure 5.3 depicts the scenario where users are connected to the geographically furthest server. In the closest case, users in Oregon were routed through Victoria and users in Virginia were routed through Carleton. In the furthest case users in Oregon were routed through Carleton and users in Virginia were routed through Victoria.

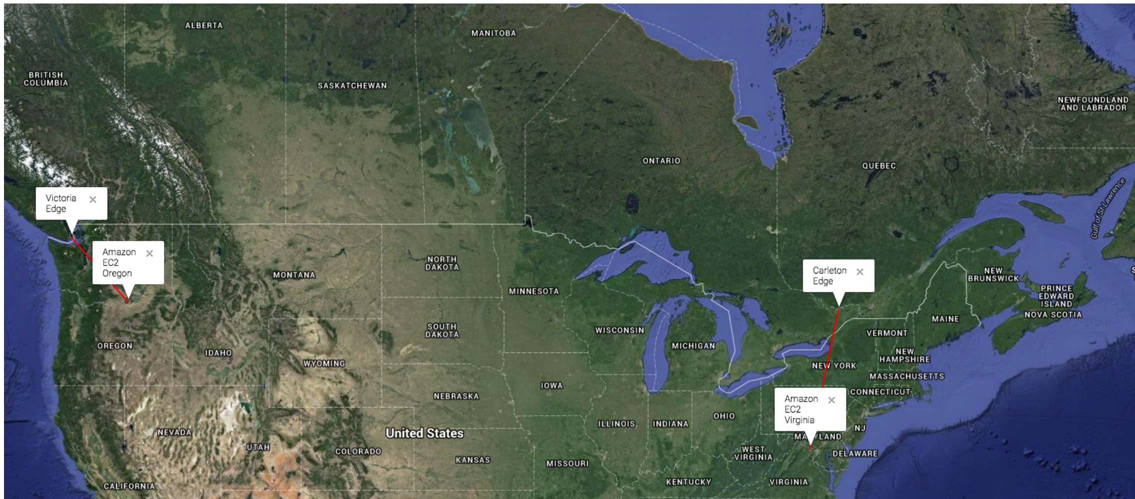


Figure 5.2: Message Routing Best Case

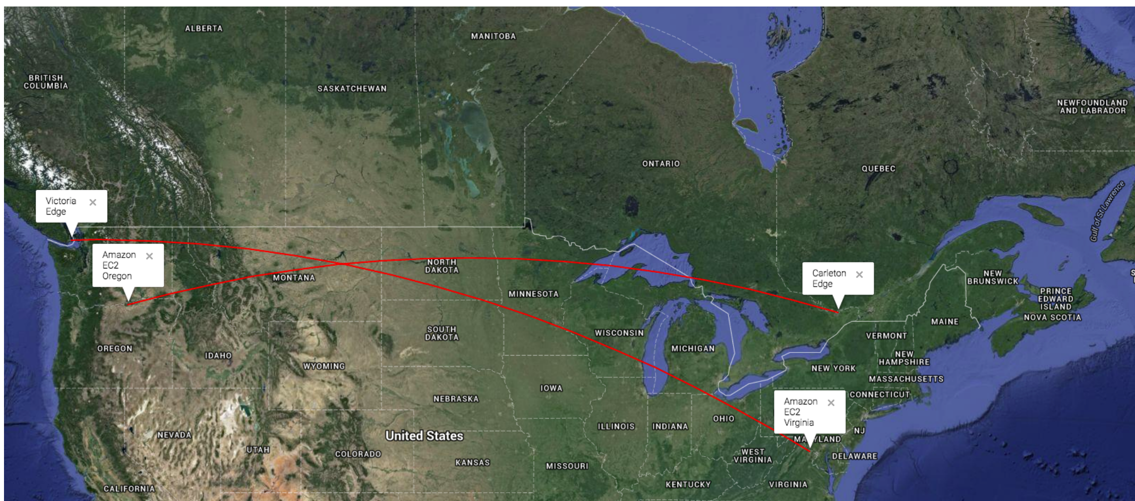


Figure 5.3: Message Routing Worst Case

For each scenario we ran five experiments using the 10 aforementioned bots at the Oregon and Virginia servers. Figure 5.4 shows the results for the closest scenario where the bots connected to the closest Yakkit Chat Service instance. Figure 5.4(a) shows the results for the Oregon to Victoria test. The average message transit time was 0.2178 seconds with a standard deviation of 0.1141 seconds. The average ping time was 0.0158 seconds with a standard deviation of 0.0133 seconds. Figure 5.4(b) shows the results for the Virginia to Carleton test. The average message transit time was 0.1843 seconds with a standard deviation of 0.0930 seconds. The average ping time was 0.0299 seconds with a standard

deviation of 0.0009 seconds.

Figure 5.5 shows the results for the furthest scenario where the bots connected to the furthest Yakkit Chat Service instance. Figure 5.5(a) shows the results for the Oregon to Carleton test. The average message transit time was 0.2410 seconds with a standard deviation of 0.1068 seconds. The average ping time was 0.0907 seconds with a standard deviation of 0.0015 seconds. Figure 5.5(b) shows the results for the Virginia to Victoria test. The average message transit time was 0.2590 seconds with a standard deviation of 0.0893 seconds. The average ping time was 0.0715 seconds with a standard deviation of 0.0011 seconds.

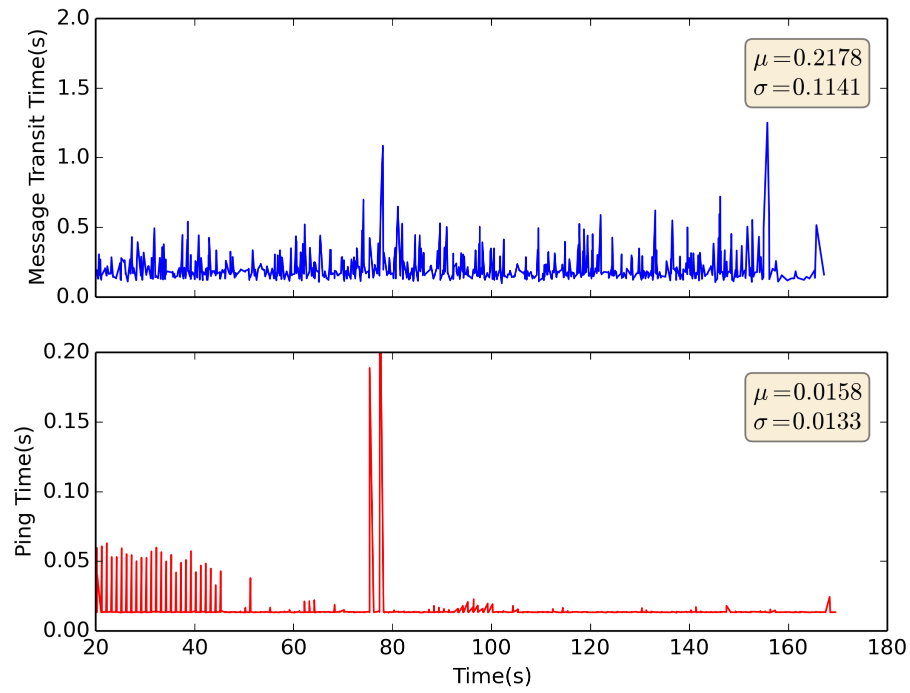
5.4 Discussion

Table 5.1 summarizes the results of the experiment and Figures 5.4 and 5.5 show the details for each experiment. The difference in the average message transit time for bots running in Oregon and connecting to either Victoria or Carleton was 0.0232 seconds. When comparing this difference to the total message transit times for each location we get 10.65% for Oregon to Victoria and 9.623% for Oregon to Carleton. This results in a 1.025% difference between using the closer Victoria service vs. the further Carleton service. Similarly, the difference in the average message transit time for bots running in Virginia and connecting to either Victoria or Carleton was 0.0747 seconds. When comparing this difference to the total message transit time for each location we get 40.53% for Virginia to Carleton and 28.84% for Virginia to Victoria. This results in a 11.69% difference between using the close Carleton service vs. the further Victoria service.

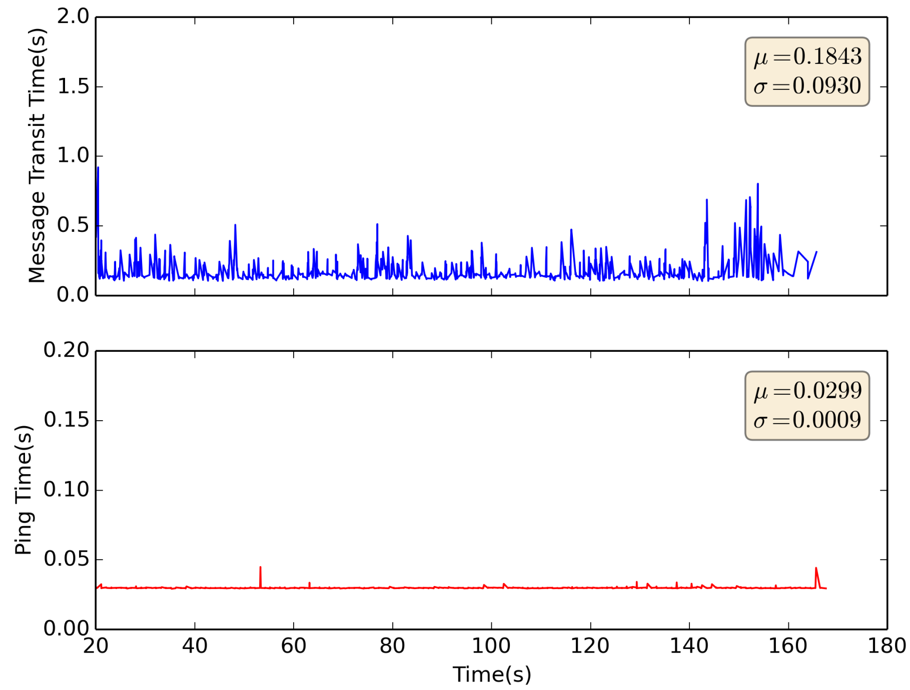
Table 5.1: Experimental Result Summary

Bot Location	**Delta Between Services (s)**	**Delta Between Services (%)**
Oregon	0.0232	1.025
Virginia	0.0747	11.69

Although the closer services performed better as expected, the difference, especially in the Oregon case, was much smaller than anticipated. Figure 5.4(a) is a concrete example of why we chose to record ping times in addition to the message transit times. At 70 seconds there is a noticeable spike in the ping time which then seems to affect the message transit time. At 150 seconds there is a noticeable spike in the message transit time but without any corresponding spike in the ping time. In the former case we can argue that the network is a major factor in that latency whereas in the latter case the internal latency of our application is a major factor. If we wanted to implement an autonomic manager (AM) to manage the

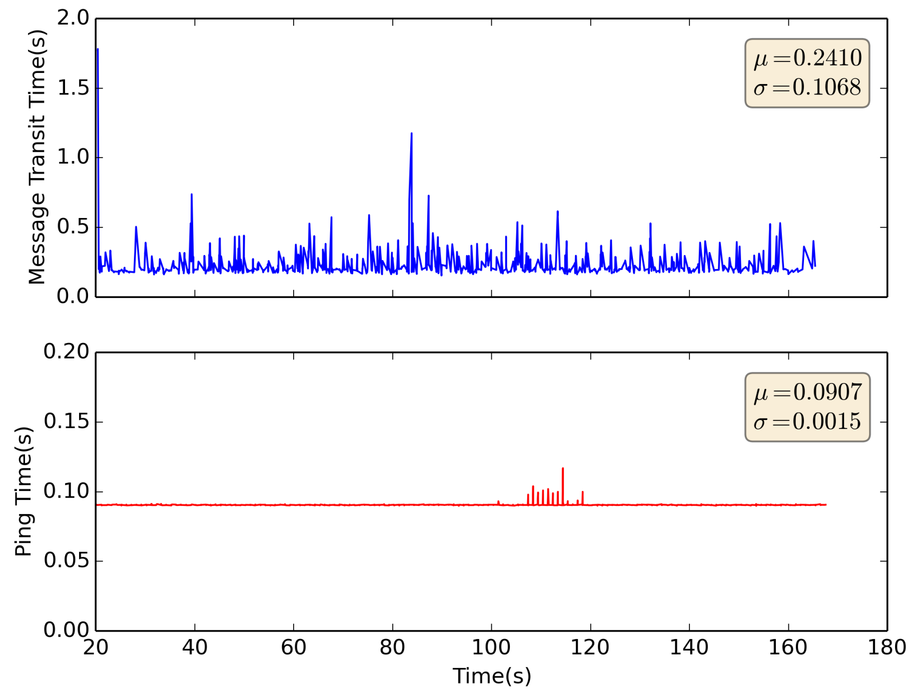


(a) Oregon to Victoria Ping and Message Transit Results

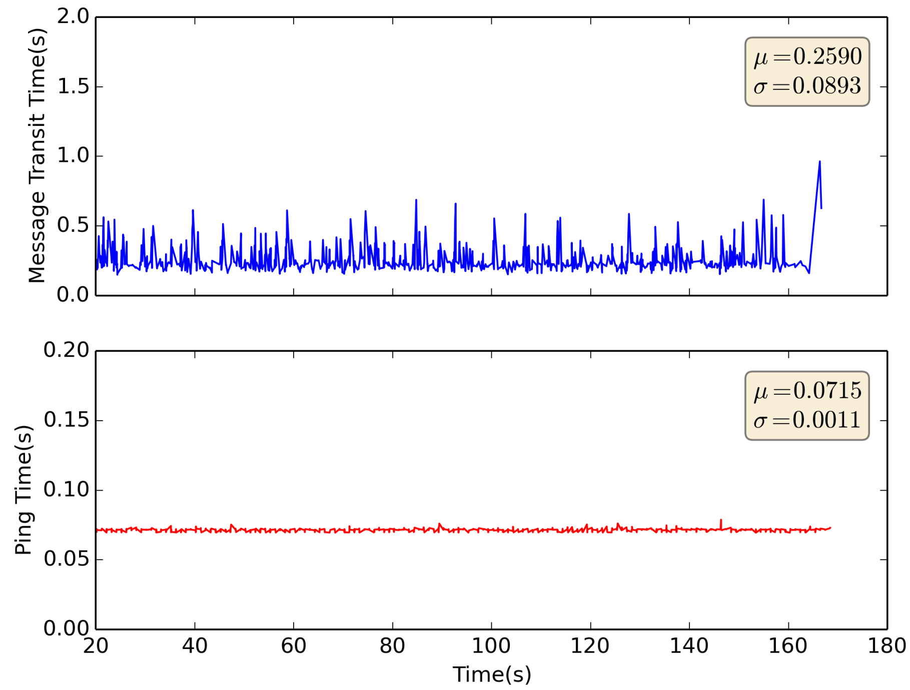


(b) Virginia to Carleton Ping and Message Transit Results

Figure 5.4: Closest Proximity Experiment Results



(a) Oregon to Carleton Ping and Message Transit Results



(b) Virginia to Victoria Ping and Message Transit Results

Figure 5.5: Farthest Proximity Experiment Results

overall latency of the system, this characterization of latency sources would be a critical piece of information for the AM to adapt accordingly.

The stability of the remaining ping times for the duration of the experiment validates our ability to compare message transit times. With a small number of users, in our case 10 on each coast, it may not be advantageous to deal with the overhead of implementing geographically based deployments; however, the Yakkit Chat Service may not scale linearly and as the number of users grows the advantages of geographic services may become attractive.

5.5 Threats to Validity

In the experiments we simulated users by running processes on the same machine and sending messages at random intervals between 0-20 seconds. In reality, users can be in an area that is several kilometres across and send messages that have a greater variety than our message pool. We feel that the scale of our experiments helps in mitigating these concerns. If our experiments tried to compare latencies between Victoria and Vancouver then these concerns may indeed be valid; however, since we are comparing Victoria to Carleton, which are approximately 3,500 km apart, any local changes are small.

We ran all experiments during regular business hours and over public networks. Since network traffic is something that cannot be controlled the results of the experiment of different days may yield different results. In addition, we have no control over the actual path that messages take. Figures 4.2 and 4.3 present "as the crow flies" simplifications of what is actually going on. It may be the case that some links are slower because they are either poorly configured or busy rather than due to Euclidean distance. In one sense our results are still valid in this scenario because they can be used to comment on which server provides a lower latency, and better user experience, however we would not be able to claim that geographic distribution is a key indicator of latency. Further experiments during different parts of the day and using different servers is part of future work.

We used system time to timestamp messages for both departure and arrival. Although all systems were synchronized using NTP timeservers at Amazon, the way in which NTP performs synchronization is dependent on the underlying network and as such susceptible to network congestion. Although this is a valid concern it is not applicable in our case since all messages originated from and arrived at the same server. In future experiments, if messages are timestamped by any other server other than the one that originates the message the precision of the system clock synchronization via NTP will have to be investigated.

During all experiments the messages were encoded and sent using JSON format. Different segments of an underlying network treat different types of traffic using different qualities of service. Our JSON messages may be treated differently then say if we were to send a

jpeg image or a video stream. The results here are only applicable to messages composed of text and this data should not be used to extrapolate to other types of traffic as that traffic may be treated differently resulting in different delivery times.

5.6 Summary

This chapter presented an experiment to determine if deploying the Yakkit Chat service closer and farther away from the user has an affect on message delivery times. Using two Yakkit Chat services, one deployed on the East Coast and the other one the West Coast, we simulated users sending messages. Our results indicate that for text message traffic there is no perceivable difference when using a closer vs. farther away Yakkit Chat service.

Chapter 6

Conclusions

6.1 Summary

This thesis investigated the challenges and opportunities that modern mobile applications face as a result of mobile devices and the context that they provide. In Chapter 2 we argued how the evolution of mobile devices has brought us a step closer to the ubiquitous computing vision proposed by Mark Weiser [Wei95]. Yet despite this progress, the applications that run on mobile devices are holding us back from this ubiquitous vision. This is, in part, due to the fact that mobile applications are being designed as extensions of their desktop counterparts rather than as an extension of their natural environment.

In Chapter 3 we presented an application implementation called Yakkit—a location based messaging application. Yakkit extends our natural ability to communicate with one another in much the same way as citizens band (CB) radio does. With CB radio you can instantly communicate with those around you without having to create a login, profile, or list of friends. Using location context Yakkit immediately allows you to chat with other nearby users and also post messages on nearby billboards. The geographic nature of the Yakkit applications makes it more like an extension of the natural environment rather than an alternate reality in a virtual world.

Developing ubiquitous applications, like Yakkit, requires a shift in complexity. The tasks that users had performed, such as maintaining a list of friends, has to now be managed by the application using real-time context. This presents many challenges such as knowing how to model context correctly and executing logic in a timely manner. In Chapters 4 and 5 we explored two such challenges and experimented with possible solutions. In Chapter 4 we used GPS data to try to model user behaviour to predict user destinations. In Chapter 5 we looked at the effects of application deployment, particularly in the form of latency, on the user experience in the Yakkit application.

6.2 Contributions

The main contributions of this thesis are as follows:

- Chapter 3 — taking inspiration from citizen band (CB) radio we developed the Yakkit application which allows users to instantly communicate with those around them. The implementation consisted of several iterations of the back-end service, the development of an iPhone client application, the development of a web-based client application, and a web portal for creating advertisements.
- Chapter 4 — introduces the idea of using anonymous GPS data to create trajectories for the purpose of modelling possible user destinations. A model and algorithm are proposed and tested on the GeoLife dataset using ten-fold cross validation. The experimental results indicate that our approach is viable for scenarios where the granularity of the predicted region is on the scale of 4,000 m^2 or higher.
- Chapter 5 — proposes a duplication of the Yakkit service at two geographically independent regions in order to determine whether the proximity of the underlying service affects the user experience. Through experimentation we observed that for our text based Yakkit Chat client the proximity of the Yakkit service made no statistically relevant difference despite being located on the opposite coasts of North America.

6.3 Future Work

The era of the Internet of Things (IoT) has only just begun and it is important to explore the development of contextual applications with the goal of simplifying the user experience rather than adding more options that will lead to an impending usability crisis.

6.3.1 Sentiment Analysis

Since the original publication of Yakkit in 2011, most of the research has revolved around geographic context and application deployment. Although these two areas require further research, it would be worthwhile to investigate whether inferring the semantic meaning of conversations can be as useful as the location itself? Perhaps the combination of a simple location service and a simple semantic service would yield better results than simply building more complicated location services? Semantic analysis is a growing field both inside and outside of computer science [Liu12]. Any further research in this direction would have a large pool of previous research to draw on from different perspectives.

6.3.2 Modelling Locations

Location will continue to be one of the most critical pieces of context for Yakkit. Modelling location is the next step in Yakkit's evolution towards the goal of ubiquity; however, more research is needed in modelling location in order for it to be truly useful for Yakkit. The application of sectors to location modelling is still a viable option but needs more analysis. One approach that requires further investigation is to base the sector sizes and initial starting positions on the density and location of trajectory starting positions. Perhaps generating these sectors based on the underlying trajectory data will yield better results at smaller sector sizes.

The infrastructure created for the work in this thesis presents an opportunity to retry the experiment in Chapter 4 using different combinations of boundaries, trajectory definitions, and sector sizes. Although we carefully designed the experiment using the most reasonable definitions of a boundary, trajectory, and sector, it would be interesting to see the effect of varying each one of those variables and observing the effect on classification.

6.3.3 Deployment

For our latency experiment we considered servers in North America and text messages; however, network quality can vary greatly in different parts of the world and network traffic may be treated differently by internet service providers (ISPs) depending on its type. In our experiment we conclude that in North America the proximity of the Yakkit services did not affect the user experience. Trying this experiment again using servers in Europe or Asia and using messages that contain text and video may yield different results. Since Yakkit will eventually support text and video and is accessible globally, this kind of test is necessary moving forward.

Bibliography

[Bar05] E. Bardram. The Trouble with Login: On Usability and Computer Security in Ubiquitous Computing. *Personal and Ubiquitous Computing*, 9(6):357–367, 2005.

[Bar09] J. E. Bardram. Activity-Based Computing for Medical Work in Hospitals. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(2):10, 2009.

[BSM10] L. Backstrom, E. Sun, and C. Marlow. Find Me If You Can: Improving Geographical Prediction with Social and Spatial Proximity. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 61–70. ACM, 2010.

[CCT+11] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei. Measuring the Latency of Cloud Gaming Systems. In *Proceedings of the 19th ACM International Conference on Multimedia (ACMMM)*, pages 1269–1272. ACM, 2011.

[CJZ+09] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu. Trajectory Simplification Method for Location-Based Social Networking Services. In *Proceedings of the 2009 International Workshop on Location Based Social Networks (LBSN)*, pages 33–40. ACM, 2009.

[CWSR12] S. Choy, B. Wong, G. Simon, and C. Rosenberg. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-User Latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2012.

[Des13] R. J. Desmarais. *Adaptive Solutions to Resource Provisioning and Task Allocation Problems for Cloud Computing*. PhD thesis, Department of Computer Science University of Victoria, 2013.

[DLM11] R. J. Desmarais, P. Lach, and H. A. Müller. YaKit: A Locality Based Messaging System Using iCon Overlay. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 148–159. ACM, IBM, 2011.

[DT08] A. Dada and F. Thiesse. Sensor Applications in the Supply Chain: The Example of Quality-Based Issuing of Perishables. In *The Internet of Things*, pages 140–154. Springer, 2008.

[E+07] N. B. Ellison et al. Social Network Sites: Definition, History, and Scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2007.

[EA12] P. C. Evans and M. Annunziata. Industrial Internet: Pushing The Boundaries of Minds and Machines. Technical report, General Electric White Paper, 2012.

[FHT+12] A. J. Ferrer, F. HernáNdez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, et al. OPTIMIS: A Holistic Approach to Cloud Service Provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012.

[FS11] K. R. Fall and W. R. Stevens. *TCP/IP Illustrated, Volume 1: The protocols.* Addison-Wesley, 2011.

[Gar12] R. Garner. *Search and Social: The Definitive Guide to Real-Time Content Marketing.* John Wiley & Sons, 2012.

[HO04] N. Hristova and G. M. O'Hare. Ad-Me: Wireless Advertising Adapted to the User Location, Device and Emotions. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 10–pp. IEEE, 2004.

[IBM93] IBM Corporation. IBM/BellSouth Joint Effort Produces Simon. *Computer Dealer News*, 9(26):41, 1993.

[IBM06] IBM Corporation. An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, 2006.

[IYE11] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 104–113. IEEE, 2011.

[Jam13] J. T. James. A New, Evidence-Based Estimate of Patient Harms Associated with Hospital Care. *Journal of Patient Safety*, 9(3):122–128, 2013.

[JRM+10] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 159–168. IEEE, 2010.

[KC03] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[KLBLG14] J.-M. Kang, T. Lin, H. Bannazadeh, and A. Leon-Garcia. Software-Defined Infrastructure and the SAVI Testbed. In *Testbeds and Research Infrastructure: Development of Networks and Communities*, pages 3–13. Springer, 2014.

[Kle99] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[KMRF09] S. Karpischek, F. Michahelles, F. Resatsch, and E. Fleisch. Mobile Sales Assistant an NFC-Based Product Information System for Retailers. In *First International Workshop on Near Field Communication (NFC)*, pages 20–23. IEEE, 2009.

[Lim14] T. Lima. Websocket-Based Chat Service Requirements and Development. Technical report, 2014.

[Liu12] B. Liu. Sentiment Analysis And Opinion Mining. *Synthesis Lectures on Human Language Technologies*, 5(1):1–167, 2012.

[LKBLG14] T. Lin, J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia. Enabling SDN Applications on Software-Defined Infrastructure. In *IEEE Network Operations and Management Symposium (NOMS)*, pages 1–7, 2014.

[LM05] D. Lemire and A. Maclachlan. Slope One Predictors for Online Rating-Based Collaborative Filtering. In *Proceedings of SIAM Data Mining (SDM)*, volume 5, pages 1–5. SIAM, 2005.

[LM13] P. Lach and H. Müller. Towards Smarter Task Applications. In *9th World Congress on Services (SERVICES)*, pages 141–146. IEEE, 2013.

[LNNK+05] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, and A. Tomkins. Geographic Routing in Social Networks. National Academy of Sciences, 2005.

[MAB+10] M. L. Mazurek, J. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, et al. Access Control for Home Data Sharing:

Attitudes, Needs and Practices. In *Proceedings of the Conference on Human Factors in Computing Systems (SIGCHI)*, pages 645–654. ACM, 2010.

[Mil67] S. Milgram. The Small World Problem. *Psychology Today*, 2(1):60–67, 1967.

[Mil68] R. B. Miller. Response Time in Man-Computer Conversational Transactions. In *Proceedings Fall Joint Computer Conference, Part I*, pages 267–277. ACM, 1968.

[MKG+08] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth Of The Flickr Social Network. In *Proceedings of the 1st Workshop on Online Social Networks (WOSN)*, pages 25–30. ACM, 2008.

[MKS09] H. Müller, H. Kienle, and U. Stege. Autonomic Computing Now You See It, Now You Don't. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 32–54. Springer, 2009.

[MLC+13] M. Madden, A. Lenhart, S. Cortesi, U. Gasser, M. Duggan, A. Smith, and M. Beaton. Teens, Social Media, and Privacy. *Pew Research Center*, 2013.

[MPS08] H. Müller, M. Pezzè, and M. Shaw. Visibility of Control in Adaptive Systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS)*, pages 23–26. ACM, 2008.

[NFG+06] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, et al. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* 2006.

[NHC09] D. Niyato, E. Hossain, and S. Camorlinga. Remote Patient Monitoring Service Using Heterogeneous Wireless Access Networks: Architecture and Optimization. *IEEE Journal on Selected Areas in Communications*, 27(4):412–423, 2009.

[Nor02] D. A. Norman. *The Design of Everyday Things.* Basic Books, 2002.

[RMMCL13] J. Rodríguez-Molina, J.-F. Martínez, P. Castillejo, and L. López. Combining Wireless Sensor Networks and Semantic Middleware for an Internet of Things-Based Sportsman/Woman Monitoring Application. *Sensors*, 13(2):1787–1835, 2013.

[SAW94] B. Schilit, N. Adams, and R. Want. Context-aware Computing Applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 85–90, 1994.

[Sch09] B. Schwartz. *The Paradox of Choice*. HarperCollins, 2009.

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC, 1986.

[Sil12] N. Silver. *The Signal and the Noise: Why so Many Predictions Fail-But Some Don't*. Penguin Press, 2012.

[SLL+14] J. Saramäki, E. Leicht, E. López, S. G. Roberts, F. Reed-Tsochas, and R. I. Dunbar. The Persistence of Social Signatures in Human Communication. *Proceedings of the National Academy of Sciences*, 111(3):942–947, 2014.

[SNLM11] S. Scellato, A. Noulas, R. Lambiotte, and C. Mascolo. Socio-Spatial Properties of Online Location-Based Social Networks. *Proceedings of the Fifth International Conference on Weblogs and Social Media (ICWSM)*, 11:329–336, 2011.

[TBK99] D. P. Truex, R. Baskerville, and H. Klein. Growing Systems in Emergent Organizations. *Communications of the ACM*, 42(8):117–123, 1999.

[THR05] D. V. Thompson, R. W. Hamilton, and R. T. Rust. Feature Fatigue: When Product Capabilities Become Too Much of a Good Thing. *Journal of Marketing Research*, 42(4):431–442, 2005.

[TM12] D. I. Tamir and J. P. Mitchell. Disclosing Information about the Self is Intrinsically Rewarding. *Proceedings of the National Academy of Sciences of the United States of America*, 109(21):8038–8043, 2012.

[UKBM11] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The Anatomy of the Facebook Social Graph. *Computing Research Repository (CoRR)*, 2011.

[VM10] N. M. Villegas and H. A. Müller. Managing Dynamic Context to Optimize Smart Interactions and Services. In *The Smart Internet*, pages 289–318. Springer, 2010.

[Wei95] M. Weiser. The Computer for the 21st Century. *Scientific American*, 272(3):78–89, 1995.

[WN10] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th Conference on Information Communications (INFOCOM)*, pages 1163–1171. IEEE, 2010.

[WTLN13] H. Wen, P. Tiwary, and T. Le-Ngoc. Current Trends and Perspectives in Wireless Virtualization. In *2013 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)*, pages 62–67, 2013.

[YZXW10] H. Yoon, Y. Zheng, X. Xie, and W. Woo. Smart Itinerary Recommendation Based on User-Generated GPS Trajectories. In *Ubiquitous Intelligence and Computing*, pages 19–34. Springer, 2010.

[ZCL+10] Y. Zheng, Y. Chen, Q. Li, X. Xie, and W.-Y. Ma. Understanding Transportation Modes Based on GPS Data for Web Applications. *ACM Transactions on the Web (TWEB)*, 4(1):1, 2010.

[ZZM+11] Y. Zheng, L. Zhang, Z. Ma, X. Xie, and W.-Y. Ma. Recommending Friends and Locations Based on Individual Location History. *ACM Transactions on the Web (TWEB)*, 5(1):5, 2011.

[ZZXM09a] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Correlation Between Locations Using Human Location History. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 472–475. ACM, 2009.

[ZZXM09b] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *International Conference on World Wide Web (WWW)*, pages 791–800. ACM, 2009.

Appendix A

Source Code

A.1 Bot Source Code

This section contains the Python source code for the bots that were run on the Virginia and Oregon servers. The main class, BotManager, starts a number of bot processes which then start their own ping process. Messages are sent and received using each bots respective web socket client. Messages are chose from a predefined list (lines 200-214) and the experiment results are all recorded in a PostgreSQL database.

```
1  from multiprocessing import Process, Queue
2
3  import time
4  from random import randint
5  from subprocess import Popen, PIPE
6  import psycopg2
7  import json
8  from ws4py.client.threadedclient import WebSocketClient
9  import logging as lg
10
11
12 class WebSocketClient(WebSocketClient):
13     client = None
14     is_open = False
15
16     def opened(self):
17         self.is_open = True
18         self.client.opened()
19
20     def received_message(self, raw_message):
21         msg = json.loads(raw_message.__str__())
22         event = msg['EVENT']
23         content = msg['CONTENT']
24
25         print 'Received: ' + event + ':' + content
26
27         if event == 'update_vm':
28             url = "ws://%s:%s/" % (content['ADDRESS'], content['PORT'])
29
30             self.client.update_websocket(url)
31             self.client.register(self.client._location[0], self.client.
       _location[1])
32             del self
33         elif event == 'message':
34             self.client.received_message(content)
35         else:
36             lg.error("!! Invalid event")
37
38     def closed(self, code, reason=None):
39         is_open = False
40         self.client.closed(code, reason)
41
```

```
42
43 class YakkitClient:
44     _location = (0, 0)
45
46     def __init__(self, url, bot_number, conn_string, run_number,
        start_time, ramp_time, ping_ip):
47         self.update_websocket(url)
48         self._bot_number = bot_number
49         self._conn_string = conn_string
50         self._run_number = run_number
51         self._start_time = start_time
52         self._ramp_time = ramp_time
53         self.bot_start_time = time.time()
54         self.is_open = False
55         self._ping_ip = ping_ip
56         self.registered = False;
57
58         self._db_connection = psycopg2.connect(conn_string)
59         self._db_cursor = self._db_connection.cursor()
60
61     # Sends coordinates to server for registration
62     def register(self, lat, lgt):
63         # print 'Bot ' + str(self._bot_number) + ' registered.'
64         self._location = (lat, lgt)
65         message = json.dumps({"EVENT": "register",
66                               "CONTENT": {"LAT": lat,
67                                           "LGT": lgt}})
68         print 'Bot registering.'
69         self.websocket.send(message)
70         self.registered = True
71
72     # Builds a JSON and sends to the server
73     def send_message(self, content):
74
75         message = json.dumps({'EVENT': 'message', 'CONTENT': content})
76
77         while not self.registered:
78             pass
79
80         self.websocket.send(message)
81
```

```
82     # Called when a message is received. Override it to manipulate the
    data.
83     def received_message(self, message):
84         message_decoded = json.loads(message.data.decode('utf-8'))
85
86         if 'timestamp' in message_decoded:
87             transit_time = time.time() - message_decoded['timestamp']
88             current_time = time.time()
89
90             if time.time() > (self.bot_start_time + self._ramp_time):
91                 self._db_cursor.execute(
92                     'INSERT INTO transit (run_id, duration, bot_number,
     run_time, time_created) VAlUES (%s, %s, %s, %s, %s);',
93                     [self._run_number, transit_time, self._bot_number,
    (current_time - self._start_time), current_time])
94                 self._db_connection.commit()
95
96     def update_location(self, lat, lgt):
97         self._location = (lat, lgt)
98         message = json.dumps({"EVENT": "update",
99                               "CONTENT": {"LAT": lat,
100                                          "LGT": lgt}})
101         self.websocket.send(message)
102
103     def update_websocket(self, url):
104         self.registered = False
105         self.websocket = WebSocketClient(url, protocols=['http-only', '
    chat'])
106         self.websocket.client = self
107         print 'Connecting to ' + url
108
109         self.websocket.connect()
110         print 'ok'
111
112         if not '/ws' in url:
113             ip = url[5:-6]
114             self._ping_ip.put(ip)
115
116             if ip != '142.104.17.133':
117                 if ip == '134.117.57.145':
118                     print 'Bot ' + str(self._bot_number) + ' connected
    to Carleton.'
```

```
119                 elif ip == '142.104.17.132':
120                     print 'Bot ' + str(self._bot_number) + ' connected
    to Victoria.'
121                 else:
122                     print 'Bot ' + str(self._bot_number) + ' connected
    to Unknown.'
123
124                 if time.time() > (self.bot_start_time + self._ramp_time
    ):
125                     self._db_cursor.execute('INSERT INTO vm_switch (
    new_vm, run_id, time_created) VALUES(%s, %s, %s);',
126                                             [ip, self._run_number, (
    time.time() - self._start_time)])
127                     self._db_connection.commit()
128
129     def opened(self):
130         print 'Socket opened.'
131         self.is_open = True
132
133     def closed(self, code, reason=None):
134         print 'WS Closed: ' + str(code) + ':' + reason
135         self.is_open = False
136
137     def close(self):
138         self.websocket.terminate()
139         self._db_connection.close()
140         del self
141
142
143 class Ping(Process):
144     def __init__(self, poison_pill, conn_string, run_number, start_time
    , ramp_time, ping_ip):
145         self._db_connection = psycopg2.connect(conn_string)
146         self._db_cursor = self._db_connection.cursor()
147         self._start_time = start_time
148         self._ramp_time = ramp_time
149         self._run_number = run_number
150         self._ping_ip = ping_ip
151         self._poison_pill = poison_pill
152
153         Process.__init__(self)
154
```

```
155     def run(self):
156         while self._ping_ip.empty():
157             pass
158 
159         ip = self._ping_ip.get()
160         p = Popen(['ping', ip], stdout=PIPE)
161         ping_start_time = time.time()
162 
163         while self._poison_pill.empty():
164 
165             if not self._ping_ip.empty():
166                 ip = self._ping_ip.get()
167                 p = Popen(['ping', ip], stdout=PIPE)
168             line = p.stdout.readline()
169             if not line:
170                 break
171 
172             try:
173                 ping_duration = float(line[line.index('time'):-4].split
        ('=')[1]) / 1000
174                 current_time = time.time()
175                 experiment_time = current_time - self._start_time
176 
177                 if time.time() > (ping_start_time + self._ramp_time):
178                     self._db_cursor.execute(
179                         'INSERT INTO ping (run_id, duration,
        experiement_time, time_created) VAlUES (%s, %s, %s, %s);',
180                         [self._run_number, ping_duration,
        experiment_time, current_time])
181                     self._db_connection.commit()
182             except ValueError:
183                 continue
184 
185 
186 class Bot(Process):
187     def __init__(self, poison_pill, bot_number, conn_string, run_number
        , start_time, ramp_time, bot_lat, bot_long):
188         self._conn_string = conn_string
189         self._run_number = run_number
190         self._start_time = start_time
191         self._ramp_time = ramp_time
192         self._bot_number = bot_number
```

```
193         self.bot_lat = bot_lat
194         self.bot_long = bot_long
195         self._ping = None
196         self._poison_pill = poison_pill
197         self._new_target_ip = Queue()
198 
199         self._quotes = list()
200         self._quotes.append('After all is said and done, more is said
        than done.')
201         self._quotes.append('When I was born, I was so surprised I didn
        \'t talk for a year and a half.')
202         self._quotes.append('I\'m not afraid to die.  I just don\'t
        want to be there when it happens.')
203         self._quotes.append('The secret of life is not to do what you
        like, but to like what you do.')
204         self._quotes.append('Love is not about who you live with. It\'s
         about who you can\'t live without.')
205         self._quotes.append('A real friend is someone who walks in when
         the rest of the world walks out')
206         self._quotes.append('Opportunity may knock only once, but
        temptation leans on the doorbell.')
207         self._quotes.append('Good supervision is the art of getting
        average people to do superior work.')
208         self._quotes.append('Wit is educated insolence.')
209         self._quotes.append('Education is the best provision for the
        journey to old age.')
210         self._quotes.append('One swallow does not make the spring.')
211         self._quotes.append('Pleasure in the job puts perfection in the
         work.')
212         self._quotes.append('We are what we repeatedly do.')
213         self._quotes.append('Wishing to be friends is quick work, but
        friendship is a slow ripening fruit.')
214         self._quotes.append('There is safety in numbers.')
215 
216         Process.__init__(self)
217 
218     def run(self):
219         url = 'ws://142.104.17.133:8000'
220 
221         ping_poison_pill = Queue()
222         ping_ip = Queue()
223 
```

```
224         ping = Ping(ping_poison_pill, self._conn_string, self.
    _run_number, self._start_time, self._ramp_time, ping_ip)
225         ping.start()
226 
227         chat = YakkitClient(url, self._bot_number, self._conn_string,
    self._run_number, self._start_time,
228                             self._ramp_time, ping_ip)
229         chat.register(self.bot_lat, self.bot_long)
230 
231         while self._poison_pill.empty():
232             duration_sleep = randint(0, 20)
233             time.sleep(duration_sleep)
234 
235             quote = self._quotes[randint(0, len(self._quotes) - 1)]
236             chat.send_message(json.dumps({'text': quote, 'timestamp':
    time.time()}))
237 
238         ping_poison_pill.put('Time to die!')
239         ping.join()
240 
241 
242 class BotManager:
243     def __init__(self, number_of_bots, conn_string, run_number, bot_lat
    , bot_long):
244         self._number_of_bots = number_of_bots
245         self._conn_string = conn_string
246         self._run_number = run_number
247         self._bots = []
248         self._poison_pills = []
249         self._bot_lat = bot_lat
250         self._bot_long = bot_long
251 
252     def start_bots(self):
253         print 'Bot manager starting bots.'
254         start_time = time.time()
255         ramp_time = 10.0
256 
257         while len(self._bots) < self._number_of_bots:
258             poison_pill = Queue()
259             bot = Bot(poison_pill, len(self._bots), self._conn_string,
    self._run_number, start_time, ramp_time,
260                       self._bot_lat, self._bot_long)
```

```
261
262                 bot.start()
263
264                 self._poison_pills.append(poison_pill)
265                 self._bots.append(bot)
266
267         def stop_bots(self):
268             print 'Bot manager stopping bots.',
269             for pill in self._poison_pills:
270                 pill.put('Time to die!')
271
272             for bot in self._bots:
273                 bot.join()
274
275             print ''.ljust(35) + '[OK]'
```

A.2 Time Delta Kernel Density Source

This section contains the R source code for generation the Kernel Density graphs for the time deltas. A SQL query is executed to extract the time deltas between all the points that fall within our experimental boundary (line 9). Line 17 computes the 99th percentile for this data and line 19 computes the actual kernel density using R's built in kernel density function. The remainder of the function draws the graph.

```
1  library(RPostgreSQL)
2  library(ggplot2)
3
4  db_driver <- dbDriver('PostgreSQL')
5  db_connection <- dbConnect(db_driver, dbname='location-data', user='przemeklach', password='9hsP3G27kGGj')
6
7  graph_time_delta_kernel_density <- function() {
8
9    query <- sprintf('SELECT
10                     abs(point.date_recorded_delta) AS delta
11                     FROM person
12                     LEFT JOIN point ON person.id = point.person_id
13                     WHERE point.id IN (SELECT point_id
14                                        FROM boundary_point);')
15
16     result <- dbGetQuery(db_connection, query)
17     percentile_dataset <- result[result$delta < quantile(result$delta, 0.99),]
18
19     time_detlas_density <- density(percentile_dataset)
20     plot(time_detlas_density, main="", col="blue", xlab='Time Delta(s)', ylab='Density')
21 }
22
23 graph_time_delta_kernel_density()
24
25 dbDisconnect(db_connection)
26 dbUnloadDriver(db_driver)
27 rm(list=c('db_connection', 'db_driver'))
```

A.3 Trajectory Distance Kernel Density Source

This section contains the R source code for generation the Kernel Density graphs for the trajectory distances. A SQL query is executed to extract all the distances of the trajectories for our experiment. Line 13 computes the 99th percentile for this data and line 14 computes the actual kernel density using R's built in kernel density function. The remainder of the function draws the graph.

```
1  library(RPostgreSQL)
2  library(ggplot2)
3  install.packages("calibrate")
4  library(calibrate)
5
6
7  db_driver <- dbDriver('PostgreSQL')
8  db_connection <- dbConnect(db_driver, dbname='location-data', user='przemeklach', password='9hsP3G27kGGj')
9
10 graph_trajectory_distance_kernel_density <- function() {
11   result <- dbGetQuery(db_connection, 'SELECT distance from trajectory;')
12
13   percentile_dataset <- result[result$distance < quantile(result$distance, 0.99),]
14   distance_density <- density(percentile_dataset)
15   plot(distance_density, main='', col="red", xlab='Distance(m)', ylab='Density',)
16
17   max_index <- which.max(distance_density$y)
18   y_value = distance_density$y[max_index]
19   x_value = distance_density$x[max_index]
20
21   points(x_value, y_value, col='red', pch=19)
22   textxy(x_value, y_value, paste('(', round(x_value, digits=1),')'), cex=1.0, pos=4)
23 }
24
25 graph_trajectory_distance_kernel_density()
26
27 dbDisconnect(db_connection)
28 dbUnloadDriver(db_driver)
29 rm(list=c('db_connection', 'db_driver'))
```

A.4 Trajectory Generation Source

This section contains the Python source code for generating trajectories. To maximize the number of cores on the executing system the computing is parallelized for each person. In theory, this source code scales up to 182 CPU's since there are 182 persons in our dataset. For each person, only the points that fall within the boundary are selected (lines 23-35). Using these points, trajectories are created and saved back to the database (lines 42-154).

```
1  __author__ = 'Przemek'
2
3  import psycopg2
4  import calendar
5  import time
6  from multiprocessing import Process, cpu_count, Value, Lock
7  import datetime
8
9  DB_CONNECTION_STRING = 'dbname=location-data user=przemeklach password=
       JLYacUJ8ez99Ws'
10
11
12 class TrajectoryBuilder():
13     def __init__(self, point_max_time_delta, min_numb_points):
14         self.point_max_time_delta = point_max_time_delta
15         self.min_numb_points = min_numb_points
16
17     def __build_user__(self, person, sample_id, current_numb_of_process
       , cnp_lock):
18         db_connection = psycopg2.connect(DB_CONNECTION_STRING)
19         db_cursor = db_connection.cursor()
20
21         # Use inner join starting with boundary points to return only
       the points inside the boundary
22         # for a given user. This should speed things up I hope.
23         db_cursor.execute('''
24                                 SELECT
25                                   point.date_recorded_delta,
26                                   point.id,
27                                   boundary_point.id,
28                                   point.date_recorded
29                                 FROM boundary_point
30                                   INNER JOIN point ON boundary_point.
       point_id = point.id
31                                   INNER JOIN person ON point.person_id =
       person.id
32                                 WHERE
33                                   person.id = (%s)
34                                 ORDER BY point.date_recorded;
35                                 ''', [person[0]])
36
37         person_boundary_points = db_cursor.fetchall()
```

```
38        current_trajectory_points = list()
39        current_trajectory_boundary_point_ids = list()
40        current_trajectory_points_date_recorded = list()
41
42        for i, boundary_point in enumerate(person_boundary_points):
43            date_recorded_delta = boundary_point[0]
44            current_point_id = boundary_point[1]
45            current_trajectory_boundary_point_ids.append(boundary_point
    [2])
46            current_trajectory_points_date_recorded.append(
    boundary_point[3])
47
48            # If there are no points in the trajectory just add the
    current point. Once there is one or more points
49            # check to make sure the delta is below threshold. If it's
    below just keep adding points. If it's above
50            # save the trajectory, start a new trajectory, and add the
    current point as the first point in that
51            # trajectory.
52            if len(current_trajectory_points) == 0:
53                current_trajectory_points.append(current_point_id)
54            elif date_recorded_delta <= self.point_max_time_delta:
55                current_trajectory_points.append(current_point_id)
56            elif date_recorded_delta > self.point_max_time_delta:
57                # If we have enough points then save it. Otherwise
    start new.
58                if len(current_trajectory_points) > self.
    min_numb_points:
59                    # Create a trajectory entry.
60                    current_trajectory_date_recorded = max(
    current_trajectory_points_date_recorded)
61
62                    db_cursor.execute('INSERT INTO trajectory (
    sample_id, trajectory_geometry, distance, date_recorded,
    date_created) VALUES (%s, %s, %s, %s, %s) RETURNING id;', [
    sample_id, None, None, current_trajectory_date_recorded, calendar.
    timegm(time.gmtime())])
63                    trajectory_id = db_cursor.fetchone()[0]
64
65                    # Add entries for link between trajectory and
    boundary points.
```

```
66                     for boundary_point_id in
    current_trajectory_boundary_point_ids:
67                         db_cursor.execute('INSERT INTO
    trajectory_boundary_point(trajectory_id, boundary_point_id,
    date_created) VALUES (%s, %s, %s)', [trajectory_id,
    boundary_point_id, calendar.timegm(time.gmtime())])
68
69                     # Create line geometry.
70                     db_cursor.execute('''
71                                         UPDATE trajectory
72                                         SET trajectory_geometry = line.
    new_line
73                                         FROM
74                                           (SELECT
75                                              st_setsrid(st_makeline(
    trajectory_points.boundary_point_geometry), 32650) AS new_line
76                                             FROM
77                                              (SELECT
78                                                 boundary_point.
    boundary_point_geometry
79                                               FROM boundary_point
80                                                 INNER JOIN point ON
    boundary_point.point_id = point.id
81                                               WHERE point_id IN %s
82                                               ORDER BY date_recorded)
    AS trajectory_points) AS line
83                                         WHERE
84                                           trajectory.id = (%s);
85                                         ''', [tuple(
    current_trajectory_points), trajectory_id])
86
87                     # Compute distance.
88                     db_cursor.execute('''
89                                         UPDATE trajectory
90                                         SET distance = st_length(line.
    trajectory_geometry)
91                                         FROM
92                                           (SELECT
93                                              trajectory_geometry
94                                            FROM trajectory
95                                            WHERE id = (%s)) AS line
96                                         WHERE id = (%s);
```

```
97                                     ''', [trajectory_id,
    trajectory_id])
98
99                  current_trajectory_points = list()
100                 current_trajectory_boundary_point_ids = list()
101                 current_trajectory_boundary_point_ids.append(
    boundary_point[2])
102                 current_trajectory_points.append(current_point_id)
103                 current_trajectory_points_date_recorded = list()
104             elif i == len(person_boundary_points) - 1:
105                 # If we are on the very last point for a person then we
     check if the point should
106                 # be added based on the time delta and whether the
    trajectory has enough points and
107                 # then we save. This code is almost exactly the same as
     the above for saving trajectories
108                 # except in the way we re-set our variables.
109                 if date_recorded_delta > self.point_max_time_delta:
110                     # If we have enough points then save it. Otherwise
    start new.
111                     if len(current_trajectory_points) > self.
    min_numb_points:
112                         # Create a trajectory entry.
113                         current_trajectory_date_recorded = max(
    current_trajectory_points_date_recorded)
114
115                         db_cursor.execute('INSERT INTO trajectory (
    sample_id, trajectory_geometry, distance, date_recorded,
    date_created) VALUES (%s, %s, %s, %s, %s) RETURNING id;', [
    sample_id, None, None, current_trajectory_date_recorded, calendar.
    timegm(time.gmtime())])
116                         trajectory_id = db_cursor.fetchone()[0]
117
118                         # Add entry for link between trajectory and
    boundary points.
119                         db_cursor.execute('INSERT INTO
    trajectory_boundary_point(trajectory_id, boundary_point_id,
    date_craeted) VALUES (%s, %s, %s)', [trajectory_id,
    current_point_id, calendar.timegm(time.gmtime())])
120
121                         # Create line geometry.
122                         db_cursor.execute('''
```

```
123                                             UPDATE trajectory
124                                             SET trajectory_geometry =
    line.new_line
125                                             FROM
126                                               (SELECT
127                                                   st_setsrid(
    st_makeline(trajectory_points.boundary_point_geometry), 32650) AS
    new_line
128                                                FROM
129                                                  (SELECT
130                                                     boundary_point.
    boundary_point_geometry
131                                                   FROM boundary_point
132                                                     INNER JOIN point
    ON boundary_point.point_id = point.id
133                                                   WHERE point_id IN %s
134                                                   ORDER BY
    date_recorded) AS trajectory_points) AS line
135                                             WHERE
136                                               trajectory.id = (%s);
137                                             ''', [tuple(
    current_trajectory_points), trajectory_id])
138
139                         # Compute distance.
140                         db_cursor.execute('''
141                                             UPDATE trajectory
142                                             SET distance = st_length(
    line.trajectory_geometry)
143                                             FROM
144                                               (SELECT
145                                                  trajectory_geometry
146                                                FROM trajectory
147                                                WHERE id = (%s)) AS line
148                                             WHERE id = (%s);
149                                             ''', [trajectory_id,
    trajectory_id])
150                 current_trajectory_points = list()
151                 current_trajectory_points_date_recorded = list()
152
153         db_connection.commit()
154         db_connection.close()
155
```

```
156         with cnp_lock:
157             current_numb_of_process.value -= 1
158 
159         print ('User ' + str(person[0]) + ' [DONE]')
160 
161     def build(self):
162         db_connection = psycopg2.connect(DB_CONNECTION_STRING)
163         db_cursor = db_connection.cursor()
164         start_time = time.time()
165 
166         # Create tables if they don't exist yet.
167         db_cursor.execute('CREATE TABLE IF NOT EXISTS sample (id SERIAL
        PRIMARY KEY, point_max_time_delta INTEGER, min_numb_points INTEGER
        , distance_threshold FLOAT, date_created INTEGER);')
168         db_cursor.execute('CREATE TABLE IF NOT EXISTS trajectory (id
        SERIAL PRIMARY KEY, sample_id INTEGER, distance Float,
        date_recorded INTEGER, too_long BOOLEAN, date_created INTEGER,
        FOREIGN KEY(sample_id) REFERENCES sample(id) ON DELETE CASCADE);')
169         db_connection.commit()
170 
171         db_cursor.execute('SELECT addgeometrycolumn(\'trajectory\', \'
        trajectory_geometry\', 32650, \'LINESTRINGZ\', 3);')
172         db_connection.commit()
173 
174         db_cursor.execute('CREATE TABLE IF NOT EXISTS
        trajectory_boundary_point (id SERIAL PRIMARY KEY, trajectory_id
        INTEGER, boundary_point_id INTEGER, date_created INTEGER, FOREIGN
        KEY(trajectory_id) REFERENCES trajectory(id) ON DELETE CASCADE,
        FOREIGN KEY(boundary_point_id) REFERENCES boundary_point(id));')
175         db_connection.commit()
176 
177         # Drop indexes if they exist. This will speedup inserts.
178         db_cursor.execute('DROP INDEX IF EXISTS idx_sample_id;')
179         db_cursor.execute('DROP INDEX IF EXISTS idx_trajectory_id;')
180         db_cursor.execute('DROP INDEX IF EXISTS idx_boundary_point_id;'
        )
181         db_connection.commit()
182 
183         # Check to see if scenario with thresholds already exists.
184         try:
185             db_cursor.execute('SELECT id FROM sample WHERE
        point_max_time_delta = (%s) AND min_numb_points = (%s);', [self.
```

```
    point_max_time_delta, self.min_numb_points])
186             existing_sample = db_cursor.fetchone()
187
188             if existing_sample is not None:
189                 response = raw_input("Trajectories with thresholds
    already exist. Overwrite (Yes/No)? ")
190
191                 if response == 'Yes':
192                     existing_sample_id = existing_sample[0]
193
194                     print 'Deleting sample and associated trajectories
    ...'.ljust(50),
195                     db_cursor.execute('DELETE FROM sample WHERE id = (%
    s);', [existing_sample_id])
196                     db_connection.commit()
197                     print '[OK]'
198
199                 elif response == 'No':
200                     print 'Goodbye!'
201                     exit()
202                 else:
203                     print 'Unrecognized choice. Goodbye!'
204                     exit()
205
206             print 'Building new trajectories...'.ljust(50)
207
208         except psycopg2.ProgrammingError:
209             db_connection.rollback()
210
211         # If we get this far then create new threshold
212         # and generate trajectories using threshold.
213         db_cursor.execute('INSERT INTO sample (point_max_time_delta,
    min_numb_points, distance_threshold, date_created) VALUES (%s, %s,
    %s, %s) RETURNING id;', [self.point_max_time_delta, self.
    min_numb_points, None, calendar.timegm(time.gmtime())])
214         sample_id = db_cursor.fetchone()[0]
215         db_connection.commit()
216
217         db_cursor.execute('SELECT * FROM person')
218         persons = db_cursor.fetchall()
219
220         processes = []
```

```
221         current_numb_of_process = Value('i', 0)
222         cnp_lock = Lock()
223 
224         # Temporarily disable vacuum to speed things up.
225         db_cursor.execute('ALTER TABLE trajectory SET (
    autovacuum_enabled = false, toast.autovacuum_enabled = false);')
226         db_connection.commit()
227 
228         for person in persons:
229             p = Process(target=self.__build_user__, args=(person,
    sample_id, current_numb_of_process, cnp_lock))
230 
231             # If we reached max cores just wait for a process
232             # to finish before starting next user.
233             while current_numb_of_process.value >= cpu_count():
234                 pass
235 
236             with cnp_lock:
237                 current_numb_of_process.value += 1
238 
239             processes.append(p)
240             p.start()
241 
242         for p in processes:
243             p.join()
244 
245         # Remove all the trajectories that cross outside the current
    boundary.
246         db_cursor.execute('DELETE FROM trajectory WHERE NOT st_contains
    ((SELECT boundary_geometry from boundary WHERE id = (%s)),
    trajectory.trajectory_geometry);', [sample_id])
247         db_connection.commit()
248 
249         db_cursor.execute('ALTER TABLE trajectory SET (
    autovacuum_enabled = true, toast.autovacuum_enabled = true);')
250         db_connection.commit()
251 
252         # Create indexes.
253         print 'Creating indexes...'.ljust(35),
254         db_cursor.execute('CREATE INDEX idx_sample_id ON trajectory(
    sample_id);')
```

```
255         db_cursor.execute('CREATE INDEX idx_trajectory_id ON
       trajectory_boundary_point(trajectory_id);')
256         db_cursor.execute('CREATE INDEX idx_boundary_point_id ON
       trajectory_boundary_point(boundary_point_id);')
257         db_connection.commit()
258         print '  [DONE]'
259 
260         end_time = time.time()
261 
262         print '\nSUMMARY'
263         print ('Duration: '.ljust(35) + str(str(datetime.timedelta(
       seconds=(end_time - start_time)))))
264 
265     # Set the too_long flag for each trajectory based on
266     # maximum distance. You figure out the max value via using the
267     # kernel density function in R. Although the schema supports
268     # multiple samples this function is currently only implemented
269     # assuming only one sample and will need to be update.
270     def crop(self, distance_max):
271         db_connection = psycopg2.connect(DB_CONNECTION_STRING)
272         db_cursor = db_connection.cursor()
273 
274         db_cursor.execute('SELECT id FROM sample WHERE
       point_max_time_delta = (%s) AND min_numb_points = (%s);', [self.
       point_max_time_delta, self.min_numb_points])
275         sample_id = db_cursor.fetchone()[0]
276 
277         if sample_id:
278             db_cursor.execute('''
279                                 UPDATE trajectory
280                                 SET too_long = TRUE
281                                 WHERE distance > (%s) AND sample_id =
       (%s);
282             ''', [distance_max, sample_id])
283 
284             db_cursor.execute('''
285                                 UPDATE trajectory
286                                 SET too_long = FALSE
287                                 WHERE distance <= (%s) AND sample_id =
       (%s);
288             ''', [distance_max, sample_id])
289 
```

```
290             db_cursor.execute('UPDATE sample SET distance_threshold =
    (%s) WHERE id = (%s);', [distance_max, sample_id])
291             db_connection.commit()
292         else:
293             print 'Sample does not exist. Goodbye!'
294 
295         db_connection.close()
296 
297 
298 data = TrajectoryBuilder(point_max_time_delta=1800, min_numb_points=20)
      # 1800 = 30 min, 900 = 15 min
299 data.build()
300 data.crop(15000)
```

A.5 Experiment Source

This section contains the Python source code for the experiments. Each experiment consists of a source and destination sector size as defined by source parameter and destination parameter, respectively. Each experiment is run 10 times (line 426). Each run represents one part of the ten fold cross validation and consists of a model generation part (line 441) and a classification part (line 445). To maximize the number of cores the generation of a model for each sector as well as the classification are parallelized. The final classification results as well as the intermediate models for each part of the experiment are stored in the database.

```
1 import psycopg2
2 import random
3 import numpy
4 import calendar
5 from multiprocessing import Process, cpu_count, Value, Lock
6 from collections import defaultdict
7 import datetime
8 import time
9
10 DB_CONNECTION_STRING = 'dbname=location-data user=przemeklach password=
    JLYacUJ8ez99Ws'
11
12
13 class Setup:
14     def __init__(self):
15         self.db_connection = psycopg2.connect(DB_CONNECTION_STRING)
16         self.db_cursor = self.db_connection.cursor()
17
18     def generate_buckets(self):
19         # Create table for storing trajectories and their
20         # respective buckets.
21         self.db_cursor.execute('''
22                                 CREATE TABLE IF NOT EXISTS
    trajectory_bucket (
23                                   id            SERIAL PRIMARY KEY,
24                                   trajectory_id INTEGER,
25                                   bucket_id     INTEGER,
26                                   FOREIGN KEY (trajectory_id)
    REFERENCES trajectory (id),
27                                   date_created  INTEGER
28                                 );
29         ''')
30         self.db_connection.commit()
31
32         # Get all the trajectories that exist in boundary and are
33         # of the right length.
34         self.db_cursor.execute('SELECT id FROM trajectory WHERE
    too_long = FALSE;')
35         trajectory_ids_tuples = self.db_cursor.fetchall()
36         trajectory_ids = numpy.array([i[0] for i in
    trajectory_ids_tuples])
37
```

```
38         # Shuffle trajectory ids using Fisher Yates
39         random.shuffle(trajectory_ids)
40 
41         # Split trajectories into 10 buckets.
42         buckets = numpy.array_split(trajectory_ids, 10)
43 
44         # print len(trajectory_ids)
45 
46         # Save buckets to db.
47         for bucket_numb, bucket in enumerate(buckets):
48             for trajectory_id in bucket:
49                 self.db_cursor.execute('INSERT INTO trajectory_bucket (
    trajectory_id, bucket_id, date_created) VALUES (%s, %s, %s);', [
    trajectory_id, bucket_numb, calendar.timegm(time.gmtime())])
50 
51             self.db_connection.commit()
52 
53         # Create index on trajectory fk
54         self.db_cursor.execute('CREATE INDEX
    idx_trajectory_bucket_trajectory_id ON trajectory_bucket(
    trajectory_id);')
55         self.db_connection.commit()
56 
57 
58 class Experiment:
59     def __init__(self):
60         self.db_connection = psycopg2.connect(DB_CONNECTION_STRING)
61         self.db_cursor = self.db_connection.cursor()
62 
63     @staticmethod
64     def __compute_model_for_sector__(sector_id, trajectory_ids, run_id,
     destination_sectors_ids, current_numb_of_process, cnp_lock):
65         process_db_connection = psycopg2.connect(DB_CONNECTION_STRING)
66         process_db_cursor = process_db_connection.cursor()
67 
68         # Find all the trajectories in our modelling buckets that start
     in current sector.
69         process_db_cursor.execute('''
70                                     WITH trajectories_in_buckets AS (
71                                         SELECT
72                                             st_startpoint(
    trajectory_geometry) AS start_point,
```

```
73                                              trajectory.id
        AS trajectory_id
74                                          FROM trajectory
75                                          WHERE trajectory.id IN %s AND
    too_long = FALSE
76                                      )
77
78                                      SELECT trajectories_in_buckets.
    trajectory_id
79                                      FROM trajectories_in_buckets
80                                      WHERE st_intersects(
81                                          (SELECT sector_geometry
82                                           FROM sector
83                                           WHERE id = (%s)),
    trajectories_in_buckets.start_point
84                                      );
85          ''', (tuple(trajectory_ids), sector_id,))
86
87          trajectory_ids_starting_in_sector_tuple = process_db_cursor.
    fetchall()
88          trajectory_ids_starting_in_sector = [i[0] for i in
    trajectory_ids_starting_in_sector_tuple]
89
90          destination_counter = defaultdict(int)
91
92          # For every trajectory that starts in the current sector
93          # figure out which sector it ends in and count it.
94          for trajectory_id_starting_in_sector in
    trajectory_ids_starting_in_sector:
95              process_db_cursor.execute('''
96                                          WITH destination_sectors AS (
97                                              SELECT sector_geometry, id
98                                              FROM sector
99                                              WHERE id IN %s)
100                                         SELECT id
101                                         FROM destination_sectors
102                                         WHERE st_intersects(
    destination_sectors.sector_geometry, (SELECT st_endpoint(
    trajectory_geometry)
103
                                FROM trajectory
```

```
104
                                WHERE id = (%s) AND too_long = FALSE));
105           ''', (tuple(destination_sectors_ids),
trajectory_id_starting_in_sector,))
106
107           destination_grid_id_tuple = process_db_cursor.fetchone()
108           destination_grid_id = destination_grid_id_tuple[0]
109           destination_counter[destination_grid_id] += 1
110
111       total_number_of_destinations = sum(destination_counter.values()
)
112
113       if total_number_of_destinations > 0:
114           # Store results in model.
115           process_db_cursor.execute('INSERT INTO model(run_id,
source_sector_id, date_created) VALUES (%s, %s, %s) RETURNING id;',
 [run_id, sector_id, calendar.timegm(time.gmtime())])
116           model_id = process_db_cursor.fetchone()[0]
117           process_db_connection.commit()
118
119           for destination_sector, count in destination_counter.
iteritems():
120               process_db_cursor.execute('INSERT INTO probability(
model_id, destination_sector_id, number_of_endpoints,
destination_probability, date_created) VALUES (%s, %s, %s, %s, %s);
', [model_id, destination_sector, count, count/float(
total_number_of_destinations), calendar.timegm(time.gmtime())])
121
122           process_db_connection.commit()
123
124       process_db_connection.close()
125
126       with cnp_lock:
127           current_numb_of_process.value -= 1
128
129   def __compute_model__(self, modeling_buckets_ids, run_id,
source_parameter, destination_parameter, shifted):
130       # Get a list of all the source and destination sectors
131       self.db_cursor.execute('''
132                                 SELECT sector.id
133                                        FROM sector
134                                        WHERE grid_id = (SELECT id
```

```
135                                                             FROM grid
136                                                             WHERE grid.
    parameter = (%s) AND grid.shifted = (%s));
137
138         ''', [source_parameter, shifted])
139
140         source_sectors_ids_tuple = self.db_cursor.fetchall()
141         source_sectors_ids = None
142
143         if len(source_sectors_ids_tuple) > 1:
144             source_sectors_ids = [i[0] for i in
    source_sectors_ids_tuple]
145         else:
146             source_sectors_ids = source_sectors_ids_tuple[0]
147
148         destination_sectors_ids = None
149
150         # If the source and destination are the same then just copy.
151         if destination_parameter == source_parameter:
152             destination_sectors_ids = list(source_sectors_ids)
153         else:
154             self.db_cursor.execute('''
155                                         SELECT sector.id
156                                                 FROM sector
157                                                 WHERE grid_id = (SELECT id
158                                                                 FROM grid
159                                                                 WHERE grid
    .parameter = (%s) AND grid.shifted = (%s));
160
161             ''', [destination_parameter, shifted])
162
163             destination_sectors_ids_tuple = self.db_cursor.fetchall()
164             if len(destination_sectors_ids_tuple) > 1:
165                 destination_sectors_ids = [i[0] for i in
    destination_sectors_ids_tuple]
166             else:
167                 destination_sectors_ids = destination_sectors_ids_tuple
    [0]
168
169         # Get all the trajectories that are found in our modelling
    buckets.
```

```
170         self.db_cursor.execute('SELECT trajectory_id from
    trajectory_bucket WHERE bucket_id IN %s', (tuple(
    modeling_buckets_ids),))
171         trajectory_ids_for_modelling_tuples = self.db_cursor.fetchall()
172         trajectory_ids_for_modelling = [i[0] for i in
    trajectory_ids_for_modelling_tuples]
173 
174         processes = []
175         current_numb_of_process = Value('i', 0)
176         cnp_lock = Lock()
177 
178         # For each sector compute the probabilities.
179         for count, sector_id in enumerate(source_sectors_ids):
180             p = Process(target=self.__compute_model_for_sector__, args
    =(sector_id, trajectory_ids_for_modelling, run_id,
    destination_sectors_ids, current_numb_of_process, cnp_lock))
181             processes.append(p)
182             p.start()
183 
184             with cnp_lock:
185                 current_numb_of_process.value += 1
186 
187             # If we reached max cores just wait for a process
188             # to finish before starting next user.
189             while current_numb_of_process.value >= cpu_count():
190                 pass
191 
192         for p in processes:
193             p.join()
194 
195         self.db_connection.commit()
196         self.__run_cleanup_and_reindex__()
197 
198     def create_tables(self):
199         # Create tables for experiments and runs.
200         self.db_cursor.execute('''
201                               CREATE TABLE IF NOT EXISTS experiment (
202                                 id                        SERIAL PRIMARY
    KEY,
203                                 duration                  INTEGER,
204                                 source_parameter          INTEGER,
205                                 destination_parameter INTEGER,
```

```
206                                     shifted                BOOL,
207                                     date_created           INTEGER
208                                 );
209         ''')
210         self.db_connection.commit()
211
212         self.db_cursor.execute('''
213                                 CREATE TABLE IF NOT EXISTS run (
214                                     id              SERIAL PRIMARY KEY,
215                                     experiment_id   INTEGER,
216                                     date_created    INTEGER,
217                                     FOREIGN KEY (experiment_id)
    REFERENCES experiment(id) ON DELETE CASCADE
218                                 );
219         ''')
220         self.db_cursor.execute('CREATE INDEX idx_run_experiment_id ON
    run(experiment_id);')
221         self.db_connection.commit()
222
223         # Create the model and probability tables for storing the
    models for each sector.
224         self.db_cursor.execute('''
225                                 CREATE TABLE IF NOT EXISTS model (
226                                     id                  SERIAL PRIMARY KEY,
227                                     run_id              INTEGER,
228                                     source_sector_id    INTEGER,
229                                     date_created        INTEGER,
230                                     FOREIGN KEY (source_sector_id)
    REFERENCES sector(id),
231                                     FOREIGN KEY (run_id) REFERENCES run(
    id) ON DELETE CASCADE
232                                 );
233         ''')
234         self.db_cursor.execute('CREATE INDEX idx_model_source_sector_id
     ON model(source_sector_id);')
235         self.db_cursor.execute('CREATE INDEX idx_model_run_id ON model(
    run_id);')
236         self.db_connection.commit()
237
238         self.db_cursor.execute('''
239                                 CREATE TABLE IF NOT EXISTS probability
    (
```

```
240                                 id                        SERIAL
    PRIMARY KEY,
241                                 model_id                  INTEGER,
242                                 destination_sector_id     INTEGER,
243                                 number_of_endpoints       INTEGER,
244                                 destination_probability FLOAT,
245                                 time_range_start          TIME,
246                                 time_range_stop           TIME,
247                                 date_created              INTEGER,
248                                 FOREIGN KEY (model_id) REFERENCES
    model(id) ON DELETE CASCADE,
249                                 FOREIGN KEY (destination_sector_id)
    REFERENCES sector(id)
250                             );
251         ''')
252         self.db_cursor.execute('CREATE INDEX idx_probability_model_id
    ON probability(model_id);')
253         self.db_cursor.execute('CREATE INDEX
    idx_probability_destination_sector_id ON probability(
    destination_sector_id);')
254         self.db_connection.commit()
255
256         # Create table to store testing results.
257         self.db_cursor.execute('''
258                             CREATE TABLE IF NOT EXISTS result (
259                                 id
    SERIAL PRIMARY KEY,
260                                 run_id
    INTEGER,
261                                 sector_id
    INTEGER,
262                                 actual_destination_sector_id
    INTEGER,
263                                 distance_delta_predicted_actual
    INTEGER,
264                                 predicated_destination_sector_id
    INTEGER,
265                                 isPredictionCorrect
    BOOL,
266                                 date_created
    INTEGER,
```

```
267                                     FOREIGN KEY (run_id) REFERENCES run(
    id) ON DELETE CASCADE
268                                   );
269             ''')
270             self.db_cursor.execute('CREATE INDEX idx_result_run_id ON
    result(run_id);')
271             self.db_connection.commit()
272 
273         def delete_tables(self):
274             self.db_connection = psycopg2.connect(DB_CONNECTION_STRING)
275             self.db_cursor = self.db_connection.cursor()
276 
277             self.db_cursor.execute('DROP TABLE IF EXISTS result;')
278             self.db_cursor.execute('DROP TABLE IF EXISTS probability;')
279             self.db_cursor.execute('DROP TABLE IF EXISTS model;')
280             self.db_cursor.execute('DROP TABLE IF EXISTS run;')
281             self.db_cursor.execute('DROP TABLE IF EXISTS experiment;')
282 
283             self.db_connection.commit()
284 
285         def clear_tables(self):
286             self.db_cursor.execute('DELETE FROM experiment;')
287             self.db_connection.commit()
288 
289             self.__run_cleanup_and_reindex__()
290 
291         def __run_cleanup_and_reindex__(self):
292             old_isolation_level = self.db_connection.isolation_level
293             self.db_connection.set_isolation_level(0)
294             self.db_cursor.execute('VACUUM ANALYZE result;')
295             self.db_cursor.execute('VACUUM ANALYZE probability;')
296             self.db_cursor.execute('VACUUM ANALYZE model;')
297             self.db_cursor.execute('VACUUM ANALYZE run;')
298             self.db_cursor.execute('VACUUM ANALYZE experiment;')
299             self.db_connection.set_isolation_level(old_isolation_level)
300 
301             self.db_connection.commit()
302 
303         @staticmethod
304         def __run_test__(testing_trajectory_ids, source_parameter,
    destination_parameter, run_id, shifted):
305             process_db_connection = psycopg2.connect(DB_CONNECTION_STRING)
```

```
306         process_db_cursor = process_db_connection.cursor()
307
308         for trajectory_id in testing_trajectory_ids:
309                 # For a given grid parameter, find the sector where
        this trajectory starts.
310                 process_db_cursor.execute('''
311                                                 WITH sectors_with_parameter
         AS (SELECT
312                                                             sector.
        sector_geometry,
313                                                             sector.id
314                                                           FROM sector
315                                                           WHERE grid_id =
         (SELECT id
316
          FROM grid
317
          WHERE grid.parameter = (%s) AND grid.shifted = (%s)))
318                                                 SELECT
        sectors_with_parameter.id
319                                                 FROM sectors_with_parameter
320                                                 WHERE st_intersects(
321                                                     (SELECT st_endpoint(
        trajectory_geometry)
322                                                      FROM trajectory
323                                                      WHERE id = (%s) AND
        too_long = FALSE), sectors_with_parameter.sector_geometry);
324                 ''', [source_parameter, shifted, trajectory_id])
325
326                 start_sector_id = process_db_cursor.fetchone()[0]
327
328                 # For a given grid parameter, find the sector where
        this trajectory ends.
329                 process_db_cursor.execute('''
330                                                 WITH sectors_with_parameter
         AS (SELECT
331                                                             sector.
        sector_geometry,
332                                                             sector.id
333                                                           FROM sector
334                                                           WHERE grid_id =
         (SELECT id
```

```
335
      FROM grid
336
      WHERE grid.parameter = (%s) AND grid.shifted = (%s)))
337                                             SELECT
    sectors_with_parameter.id
338                                             FROM sectors_with_parameter
339                                             WHERE st_intersects(
340                                                 (SELECT st_endpoint(
    trajectory_geometry)
341                                                  FROM trajectory
342                                                  WHERE id = (%s) AND
    too_long = FALSE), sectors_with_parameter.sector_geometry);
343                 ''', [destination_parameter, shifted, trajectory_id])
344
345                 actual_end_sector_id = process_db_cursor.fetchone()[0]
346
347                 # See if the model predicts the same ending sector.
348                 process_db_cursor.execute('''
349                                             SELECT
    destination_sector_id
350                                             FROM probability
351                                             WHERE model_id = (
352                                               SELECT id
353                                               FROM model
354                                               WHERE source_sector_id =
    (%s) AND run_id = (%s))
355                                             ORDER BY
    destination_probability DESC
356                                             LIMIT 1;
357                 ''', [start_sector_id, run_id])
358
359                 predicted_end_sector_id_tuple = process_db_cursor.
    fetchone()
360
361                 # Some sectors may not have a model i.e.: no data for
    that sector.
362                 # If that is the case then we make no prediction and
    don't include
363                 # it in results. If the prediction is incorrect we
    compute the distance
364                 # between the sectors to determine how far off we are.
```

```
365                 if predicted_end_sector_id_tuple:
366                     predicted_end_sector_id =
    predicted_end_sector_id_tuple[0]
367
368                     if actual_end_sector_id == predicted_end_sector_id:
369                         is_prediction_correct = True
370                         distance_delta_predicted_actual = 0.0
371                     else:
372                         is_prediction_correct = False
373                         process_db_cursor.execute('''
374                                                         WITH
    actual_end_sector AS (
375                                                             SELECT
    st_centroid(sector_geometry) AS geom
376                                                             FROM sector
377                                                             WHERE id = (%s)
378                                                         ),
    predicted_end_sector AS (
379                                                             SELECT
    st_centroid(sector_geometry) AS geom
380                                                             FROM sector
381                                                             WHERE id = (%s)
382                                                         )
383                                                         SELECT st_distance(
    actual_end_sector.geom, predicted_end_sector.geom)
384                                                         FROM
    actual_end_sector, predicted_end_sector;
385                         ''', [actual_end_sector_id,
    predicted_end_sector_id])
386                         distance_delta_predicted_actual =
    process_db_cursor.fetchone()[0]
387
388                     process_db_cursor.execute('''
389                                                     INSERT INTO result(
    run_id, sector_id, actual_destination_sector_id,
    distance_delta_predicted_actual, predicated_destination_sector_id,
    ispredictioncorrect, date_created)
390                                                     VALUES (%s, %s, %s, %s,
     %s, %s, %s);
391                     ''', [run_id, start_sector_id, actual_end_sector_id
    , distance_delta_predicted_actual, predicted_end_sector_id,
    is_prediction_correct, calendar.timegm(time.gmtime())])
```

```
392
393                             process_db_connection.commit()
394
395     def execute(self, source_parameter, destination_parameter, shifted)
    :
396         start_time = time.time()
397
398         # Check to see if experiment with same source and destination
    parameters
399         # has already been run. If so, inform the user.
400         self.db_cursor.execute('SELECT * FROM experiment WHERE
    source_parameter = (%s) AND destination_parameter=(%s) AND shifted
    =(%s);', [source_parameter, destination_parameter, shifted])
401         existing_experiment = self.db_cursor.fetchone()
402
403         if existing_experiment is not None:
404             response = raw_input("Experiment with parameters already
    exists. Re-run experiment? [Yes/No]: ")
405
406             if response == 'No':
407                 print 'Goodbye!'
408                 exit()
409             elif response == 'Yes':
410                 self.db_cursor.execute('DELETE FROM experiment WHERE
    source_parameter = (%s) AND destination_parameter=(%s);', [
    source_parameter, destination_parameter])
411                 self.db_connection.commit()
412                 self.__run_cleanup_and_reindex__()
413             else:
414                 print 'Unrecognized command. Goodbye!'
415                 exit()
416
417         # Get all the bucket ids
418         self.db_cursor.execute('SELECT DISTINCT (bucket_id) FROM
    trajectory_bucket;')
419         bucket_ids_tuples = self.db_cursor.fetchall()
420         bucket_ids = [i[0] for i in bucket_ids_tuples]
421
422         self.db_cursor.execute('INSERT INTO experiment(source_parameter
    , destination_parameter, shifted, date_created) VALUES (%s, %s, %s,
     %s) RETURNING id;', [source_parameter, destination_parameter,
    shifted, calendar.timegm(time.gmtime())])
```

```
423         experiment_id = self.db_cursor.fetchone()[0]
424         self.db_connection.commit()
425 
426         # Execute 10 runs. Each run with a different bucket as the
    testing set.
427         for run in range(0, 10):
428             print ('Run ' + str(run))
429             self.db_cursor.execute('INSERT INTO run(experiment_id,
    date_created) VALUES (%s, %s) RETURNING id;', [experiment_id,
    calendar.timegm(time.gmtime())])
430             run_id = self.db_cursor.fetchone()[0]
431             self.db_connection.commit()
432 
433             testing_bucket_id = bucket_ids[run]
434             modeling_buckets_ids = list()
435 
436             for bucket_id in bucket_ids:
437                 if bucket_id != testing_bucket_id:
438                     modeling_buckets_ids.append(bucket_id)
439 
440             # Compute model for this run.
441             print ('Computing model for run ' + str(run) + '...').ljust
    (35),
442             self.__compute_model__(modeling_buckets_ids=
    modeling_buckets_ids, run_id=run_id, source_parameter=
    source_parameter, destination_parameter=destination_parameter,
    shifted=shifted)
443             print '[DONE]'
444 
445             # Compute recommendations based on model.
446             self.db_cursor.execute('SELECT trajectory_id FROM
    trajectory_bucket WHERE bucket_id = (%s);', [testing_bucket_id])
447             testing_trajectory_ids_tuples = self.db_cursor.fetchall()
448             testing_trajectory_ids = numpy.array([i[0] for i in
    testing_trajectory_ids_tuples])
449 
450             # Split the test into as many buckets as there are cpu's.
451             testing_trajectory_ids_buckets = numpy.array_split(
    testing_trajectory_ids, cpu_count())
452 
453             print ('Computing predictions for run ' + str(run) + '...')
    .ljust(35),
```

```
454             processes = []
455             for testing_bucket in testing_trajectory_ids_buckets:
456                 p = Process(target=self.__run_test__, args=(
    testing_bucket, source_parameter, destination_parameter, run_id,
    shifted))
457                 processes.append(p)
458                 p.start()
459 
460             for p in processes:
461                 p.join()
462 
463             print '[DONE]'
464 
465         end_time = time.time()
466         duration = end_time - start_time
467 
468         self.db_cursor.execute('''
469                                 UPDATE experiment
470                                 SET duration = (%s)
471                                 WHERE id = (%s);
472         ''', [duration, experiment_id])
473 
474         self.db_connection.commit()
475         self.db_connection.close()
476 
477         print '\nSUMMARY'
478         print ('Duration: '.ljust(35) + str(str(datetime.timedelta(
    seconds=duration))))
479 
480 if __name__ == '__main__':
481     #setup = Setup()
482     #setup.generate_buckets()
483 
484     #experiment = Experiment()
485     #experiment.delete_tables()
486     #experiment.create_tables()
487     #experiment.execute(source_parameter=55000, destination_parameter
    =55000)
488 
489     # experiment = Experiment()
490     # experiment.execute(source_parameter=4000, destination_parameter
    =4000, shifted=False)
```

```
491
492     # experiment = Experiment()
493     # experiment.execute(source_parameter=4000, destination_parameter
        =4000, shifted=True)
494     # experiment.execute(source_parameter=2500, destination_parameter
        =2500, shifted=True)
495
496     #source_parameters = range(9000, 11000, 500)
497     destination_parameters = range(1500, 4500, 500)
498
499     for destination_parameter in destination_parameters:
500         print 'RUNNING: ' + str(1000) + ':' + str(destination_parameter
        )
501
502         experiment = Experiment()
503         experiment.execute(source_parameter=1000, destination_parameter
        =destination_parameter, shifted=False)
504
505         experiment = Experiment()
506         experiment.execute(source_parameter=1000, destination_parameter
        =destination_parameter, shifted=True)
```