

THESIS

MACHINE LEARNING TECHNIQUES FOR ENERGY OPTIMIZATION IN MOBILE  
EMBEDDED SYSTEMS

Submitted by

Brad Kyoshi Donohoo

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2012

Master's Committee:

Advisor: Sudeep Pasricha

Charles Anderson  
Anura P. Jayasumana

Copyright by Brad Kyoshi Donohoo 2012

All Rights Reserved

## ABSTRACT

### MACHINE LEARNING TECHNIQUES FOR ENERGY OPTIMIZATION IN MOBILE EMBEDDED SYSTEMS

Mobile smartphones and other portable battery operated embedded systems (PDAs, tablets) are pervasive computing devices that have emerged in recent years as essential instruments for communication, business, and social interactions. While performance, capabilities, and design are all important considerations when purchasing a mobile device, a long battery lifetime is one of the most desirable attributes. Battery technology and capacity has improved over the years, but it still cannot keep pace with the power consumption demands of today's mobile devices. This key limiter has led to a strong research emphasis on extending battery lifetime by minimizing energy consumption, primarily using software optimizations. This thesis presents two strategies that attempt to optimize mobile device energy consumption with negligible impact on user perception and quality of service (QoS).

The first strategy proposes an application and user interaction aware middleware framework that takes advantage of user idle time between interaction events of the foreground application to optimize CPU and screen backlight energy consumption. The framework dynamically classifies mobile device applications based on their received interaction patterns, then invokes a number of different power management algorithms to adjust processor frequency and screen backlight levels accordingly.

The second strategy proposes the usage of machine learning techniques to learn a user's mobile device usage pattern pertaining to spatiotemporal and device contexts, and then predict energy-optimal data and location interface configurations. By learning where and when a mobile device

user uses certain power-hungry interfaces (3G, WiFi, and GPS), the techniques, which include variants of linear discriminant analysis, linear logistic regression, non-linear logistic regression, and k-nearest neighbor, are able to dynamically turn off unnecessary interfaces at runtime in order to save energy.

## ACKNOWLEDGEMENTS

I would like to thank all the individuals whose encouragement and support has made the completion of this thesis possible.

First, I would like to express my deep gratitude to Prof. Sudeep Pasricha, whose guidance, support, and encouragement made both of the projects presented in this thesis possible. When I arrived at Colorado State University in 2010 and I expressed interest in his areas of research, Prof. Pasricha took me under his wing. Since then, I have received an enormous amount of invaluable advice and knowledge from him that will be forever valuable in my professional and personal life. Thank you, Prof. Pasricha, for providing such a great experience for me during the past two years.

Second, I would like to extend a very sincere thank you to Chris Ohlsen, who acted as my partner while working on both of the strategies presented in this thesis. Chris is one of the best colleagues I have ever had the pleasure of working with – his work ethic, knowledge, and companionship were all vital to my completion of this work.

I would like to thank my committee members Prof. Charles Anderson and Prof. Anura P. Jayasumana for their valuable time and input. Prof. Anderson's teaching and advice allowed me to understand and utilize the machine learning algorithms in this work. My sincere gratitude goes to Prof. Jayasumana for agreeing to be on my thesis committee.

I would also like to thank my colleagues in Multicore Embedded Systems (MECS) lab for their advice and unwavering support. Their willingness to listen to my presentations, provide peer reviews, and offer helpful suggestions improved my work immensely.

Last, but not least, I would like to thank my family. My family in Colorado, the Royvals and the Doans, helped me so much when I moved here and made it feel immediately like home, for which I am very grateful. I would like to thank them, my grandparents, the Cordovas, and especially my parents and brothers for their continuous love, support, and encouragement.

## DEDICATION

*To my parents, Walt and Delia*

*and*

*To my brothers, Bryan and Brett*

*Without their support, understanding, encouragement, and love this work would not have been possible.*

## TABLE OF CONTENTS

<b>LIST OF TABLES .....</b>	<b>x</b>
<b>LIST OF FIGURES .....</b>	<b>xi</b>
<b>Chapter 1 Introduction .....</b>	<b>1</b>
1.1 Motivation .....	1
1.2 Contributions .....	2
1.3 Outline .....	3
<b>Chapter 2 An Overview of Contemporary Smartphone Platforms .....</b>	<b>4</b>
2.1 Google Nexus One Overview .....	8
<b>Chapter 3 Problem Statement .....</b>	<b>12</b>
<b>Chapter 4 Related Work .....</b>	<b>15</b>
4.1 Work on Mobile Usage Studies .....	15
4.2 Work on CPU and Backlight Energy Optimization .....	16
4.3 Work on Energy Optimization for Data and Location Interfaces .....	17
4.4 Usage of Machine Learning Techniques in Mobile Devices .....	18
4.5 Hardware Strategies for Energy Optimization .....	18
<b>Chapter 5 Machine Learning Techniques .....</b>	<b>20</b>
5.1 Markov Decision Processes .....	20
5.2 Bayesian Classification .....	21
5.3 Q-Learning .....	23
5.4 Linear Discriminant Analysis.....	24
5.5 Linear Logistic Regression.....	25
5.5.1 Scaled Conjugate Gradient .....	27
5.6 Non-Linear Logistic Regression with Neural Networks.....	29
5.7 K-Nearest Neighbor .....	31
<b>Chapter 6 Strategy 1: CPU and Backlight Energy Optimization .....</b>	<b>33</b>
6.1 Fundamental User-Device Interaction Mechanisms .....	34
6.2 User-Device Interaction Field Study .....	36
6.3 AURA Middleware Framework Overview .....	39
6.4 Runtime Monitor .....	41
6.5 Bayesian Application Classifier .....	42



6.6 Power Manager .....	44
6.6.1 Markov Decision Process based Manager .....	44
6.6.2 Q-Learning based Manager.....	47
6.6.3 Power Manager Control Parameters .....	48
6.7 Device Power Modeling.....	50
6.7.1 Overview .....	50
6.7.2 CPU DFS Power Model.....	51
6.7.3 Screen Backlight Power Model .....	54
6.8 Experimental Methodology.....	56
6.9 Experimental Results.....	57
6.9.1 Event Emulation Results.....	57
6.9.2 User Study Results .....	64
<b>Chapter 7 Strategy 2: Data and Location Interface Energy Optimization .....</b>	<b>70</b>
7.1 User Interaction Studies .....	71
7.1.1 Context Logger .....	71
7.1.2 Data Preparation.....	72
7.1.3 Real User Profiles .....	74
7.1.4 Synthetic User Profiles .....	75
7.2 Device Power Modeling.....	76
7.3 Experimental Results.....	78
7.3.1 Prediction Accuracy Analysis.....	78
7.3.2 Energy Savings .....	80
7.3.3 Implementation Overhead.....	82
<b>Chapter 8 Summary and Future Work .....</b>	<b>85</b>
8.1 Summary .....	85
8.2 Future Work .....	86
<b>References .....</b>	<b>89</b>
<b>Appendix A .....</b>	<b>94</b>
<b>Source Code.....</b>	<b>94</b>
A.1 AppSettings.java (Strategy 1).....	94
A.2 AppSimulator.java (Strategy 1).....	97
A.3 AppUtil.java (Strategy 1) .....	104
A.4 AURA.java (Strategy 1).....	108
A.5 BatteryModels.java (Strategy 1).....	123

A.6 ControlAlgorithm.java (Strategy 1) .....	123
A.7 CPUInfo.java (Strategy 1).....	125
A.8 DVFSControl.java (Strategy 1).....	128
A.9 DVFSTest.java (Strategy 1) .....	137
A.10 FrequencyGovernors.java (Strategy 1).....	145
A.11 MiscParamInfo.java (Strategy 1) .....	146
A.12 NetInfo.java (Strategy 1).....	147
A.13 PowerModels.java (Strategy 1) .....	151
A.14 SimpleXYSeries.java (Strategy 1) .....	154
A.15 State.java (Strategy 1) .....	155
A.16 TrainingSetEntry.java (Strategy 1).....	155
A.17 AppClassification.java (Strategy 1) .....	155
A.18 AppClassifier.java (Strategy 1) .....	157
A.19 ApplicationDatabase.java (Strategy 1).....	159
A.20 ApplicationInfo.java (Strategy 1).....	166
A.21 AppMonitor.java (Strategy 1) .....	180
A.22 ContextLogger.java (Strategy 2) .....	223
A.23 ContextLoggerService.java (Strategy 2) .....	227
A.24 ContextPrediction.py (Strategy 2).....	238
A.25 kNearestNeighbor.py (Strategy 2).....	255
A.26 mvsom.py (Strategy 2) .....	257
A.27 neuralNet.py (Strategy 2) .....	260
A.28 gradientDescent.py (Strategy 2).....	264
<b>ABBREVIATIONS .....</b>	<b>269</b>

## LIST OF TABLES

Table 6.1: App interaction classification .....	37
Table 6.2: Q-Learning algorithm reinforcements .....	48
Table 6.3: Algorithm control parameters .....	49
Table 6.4: Algorithm control parameter values .....	57
Table 7.1: Recorded data attributes .....	72
Table 7.2: Interface configuration states .....	74
Table 7.3: Idle power consumption for data and location interfaces .....	77
Table 7.4: Average algorithm run times .....	83

## LIST OF FIGURES

Figure 2.1: U.S. smartphone platform market share as of Jan. 2012 .....	5
Figure 2.2: Google Nexus One smartphone power distributions.....	7
Figure 2.3: Nexus One OLED display .....	9
Figure 2.4: Nexus One touchscreen sensor and controller .....	9
Figure 2.5: Nexus One PCB underside .....	10
Figure 2.6: Nexus One PCB topside and audio components .....	11
Figure 2.7: Nexus One wireless transceiver .....	11
Figure 5.1: Example of MDP with 3 states and 2 actions.....	21
Figure 5.2: Comparison of gradient descent (red) and conjugate gradient (green) algorithms ....	28
Figure 5.3: Neural network perceptron model.....	30
Figure 5.4: K-nearest neighbor example.....	32
Figure 6.1: Interaction slack process .....	35
Figure 6.2: Application interaction distributions .....	38
Figure 6.3: Application touch event timelines .....	39
Figure 6.4: <i>AURA</i> energy optimization middleware framework .....	41
Figure 6.5: Control algorithms state flow diagram.....	45
Figure 6.6: Control algorithms timeline illustration .....	47
Figure 6.7 Power monitor setup.....	51
Figure 6.8: DFS-based CPU power model for HTC Dream.....	53
Figure 6.9: DFS-based CPU power model for Google Nexus One .....	53
Figure 6.10: Monsoon power monitor screen power .....	54
Figure 6.11: Screen backlight power model for HTC Dream.....	55

Figure 6.12: Screen backlight power model for Google Nexus One .....	55
Figure 6.13: Minimum acceptable prediction rates for chosen applications .....	59
Figure 6.14: Average energy saved on HTC Dream for emulated patterns.....	61
Figure 6.15: Average energy saved on Google Nexus One for emulated patterns.....	61
Figure 6.16: Average successful prediction rate on HTC Dream for emulated patterns .....	63
Figure 6.17: Average successful prediction rate on Google Nexus One for emulated patterns ...	63
Figure 6.18: Average energy saved on HTC Dream for real users (performance bias) .....	65
Figure 6.19: Average energy saved on Google Nexus One for real users (performance bias).....	66
Figure 6.20: Average successful prediction rate on HTC Dream for real users (performance bias) .....	67
Figure 6.21: Average successful prediction rate on Google Nexus One for real users (performance bias) .....	67
Figure 6.22: Average successful prediction rate on HTC Dream for real users (energy bias) .....	68
Figure 6.23: Average successful prediction rate on Google Nexus One for real users (energybias)	69
Figure 7.1: Unique locations identified for varying GPS precisions .....	73
Figure 7.2: Real user state distributions.....	75
Figure 7.3: Synthetic user profiles .....	76
Figure 7.4: Relative power consumption of configuration states .....	77
Figure 7.5: Real user algorithm prediction accuracy .....	78
Figure 7.6: synthetic user algorithm prediction accuracy .....	80
Figure 7.7: Percent energy saved for real users .....	81
Figure 7.8: Percent energy saved for synthetic users.....	81

## Chapter 1

### Introduction

Within the past decade, mobile computing has morphed into a principal form of human communication, business, and social interaction. As of 2011, there are more than 5.3 billion mobile subscribers worldwide, with smartphone sales showing the strongest growth. Over 300,000 apps have been developed within the past three years across various mobile platforms. Popular mobile activities include web browsing, multimedia, games, e-mail, and social networking [1]. Overall, these trends suggest that mobile devices are now the new development and computing platform for the 21st century.

#### **1.1 Motivation**

Unfortunately, the energy demands of newer technologies (e.g. 4G networking, multicore/GPUs) and applications (e.g. 3D gaming, Apple's FaceTime™) on mobile devices have greatly overwhelmed modern energy storage abilities. The growing disparity between functionality and mobile energy storage has been a strong catalyst in recent years to develop software-centric algorithms and strategies for energy optimization [8]-[26]. These software techniques work in tandem with well-known energy optimizations implemented in hardware including CPU DVFS, power/clock gating, and low power mode configurations for device interfaces and chipsets [32], [33], [52].

The notion of “smart” mobile devices has recently spawned a number of research efforts on developing “smart” energy optimization strategies. Some of these efforts employ strategies that

are context-aware including utilization of device, user, spatial, temporal, and application awareness that attempt to dynamically modify or learn optimal device configurations to maximize energy savings with little or negligible impact on user perception and quality of service (QoS) [18], [45].

## 1.2 Contributions

This thesis presents two “smart” energy optimization strategies.

The first strategy, named *AURA*, focuses on optimizing mobile device CPU and backlight energy while being aware to the running application and interactions from the user. It takes advantage of user idle time between interaction events of the foreground application to reduce CPU and backlight energy consumption. In order to balance energy consumption and quality of service (QoS) requirements that are unique to each individual user, *AURA* makes use of a Bayesian application classifier to dynamically classify applications based on user interaction. Once an application is classified, *AURA* utilizes Markov Decision Process (MDP) or Q-Learning based power management algorithms to adjust processor frequency and screen backlight levels to reduce system energy consumption between user interaction events. Real-world user evaluation studies with the Google Android based HTC Dream and Google Nexus One mobile devices running the *AURA* framework demonstrate promising results, with up to 29% energy savings compared to the baseline device manager; and up to 5× savings over the best known prior work on CPU and backlight energy co-optimization, with negligible impact on user quality of service.

The second strategy focuses on optimizing energy consumed by the mobile device’s data and location interfaces. It proposes and demonstrates the use of four different classes of machine learning algorithms (i) linear discriminant analysis, (ii) linear logistic regression, (iii) k-nearest

neighbor, and (iv) non-linear logistic regression with neural networks, for their effectiveness in learning a user's mobile device usage pattern pertaining to spatiotemporal and device contexts, to predict data and location interface configurations. The resulting predictions manage the network interface states allowing for dynamic adaptation to optimal energy configurations when respective interfaces are not required while maintaining an acceptable level of user satisfaction. These strategies are tested on both synthetic and real-world user usage patterns, which demonstrate that high and consistent prediction rates are possible. The proposed techniques are also compared with prior work on device configuration prediction using self-organizing maps [45] and energy-aware location sensing [17], showing an improvement upon these state-of-the-art techniques

### **1.3 Outline**

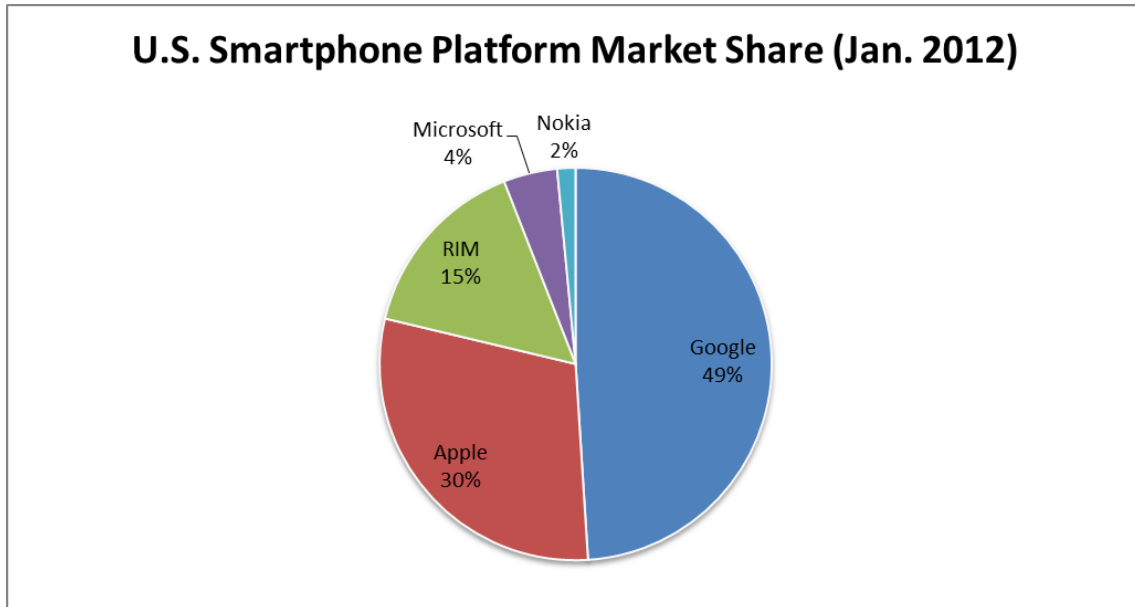
The rest of the thesis is organized as follows. Chapter 2 provides an overview of contemporary smartphone platforms. Chapter 3 describes the problem statement for the thesis. Chapter 4 reviews the current work related to context-aware energy optimizations in mobile devices. Chapter 5 provides a detailed overview of the machine learning techniques used in the thesis. Chapter 6 presents the CPU and screen backlight energy optimization strategy. Chapter 7 presents the strategy involving using machine learning algorithms to predict optimal data and location interface configurations. Chapter 8 concludes the thesis with a summary and future work. The appendix offers the source code of the two strategies presented.



## Chapter 2

### An Overview of Contemporary Smartphone Platforms

*Smartphones* are mobile computing devices with advanced computability and connectivity. Today's smartphones offer many functions in addition to just phone calls – most also serve as media players, handheld gaming devices, digital/video cameras, GPS navigation units, web browsers, and much more. They also offer high-speed data access via WiFi or mobile broadband. These devices use various mobile operating systems, with the most common being Apple's iOS, Google's Android, Nokia's Symbian, Microsoft's Windows Mobile, or RIM's BlackBerry OS, and contain APIs (Application Program Interfaces) for running third-party applications. Figure 2.1 shows the U.S. mobile subscriber market share for each of these operating systems as of January 2012. Google's Android dominates the market share, with nearly half of all smartphone subscribers using it, while Apple's iOS is a fairly close second.



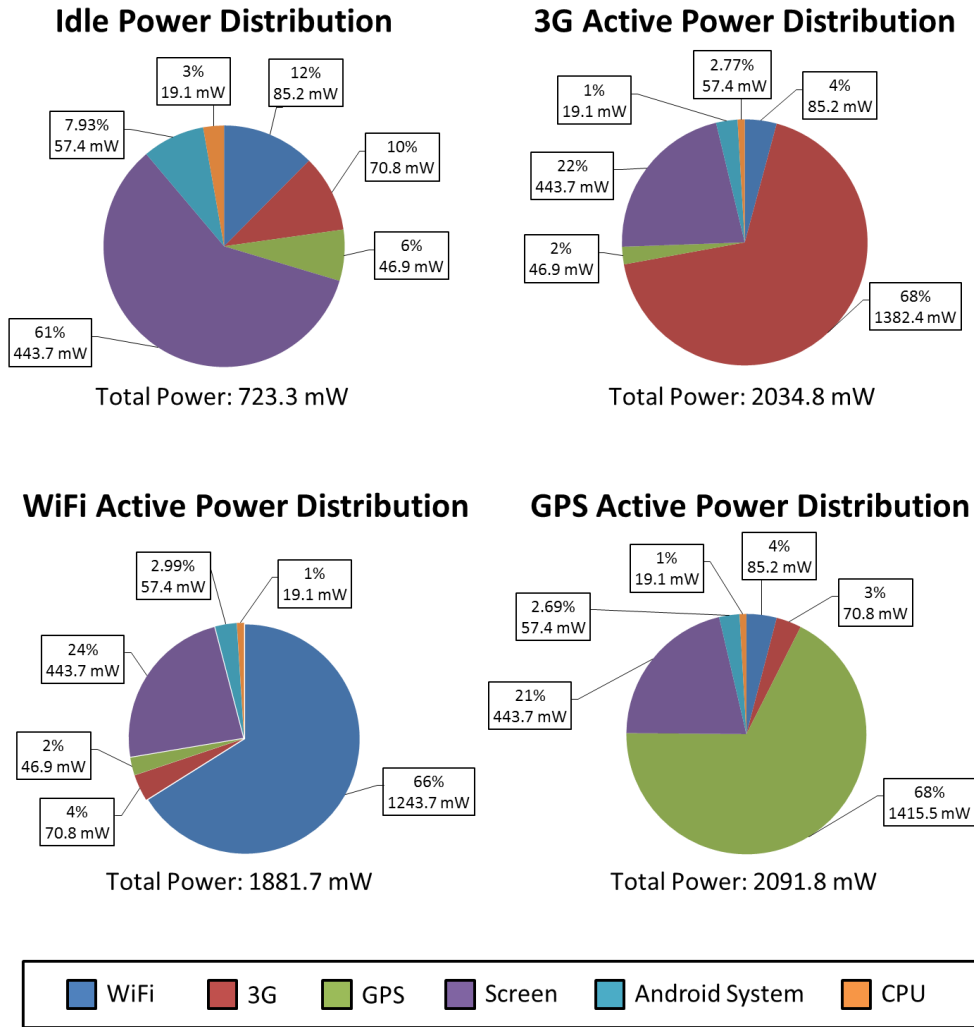
**Figure 2.1: U.S. smartphone platform market share as of Jan. 2012**

Virtually all of the smartphones today use some variant of the lithium ion battery [60]. Lithium ion battery technology is approximately 15 years old, and while there have been significant improvements since its inception, it is further down the development curve than other smartphone technologies. For this reason, improvements in smartphone battery technology are having trouble keeping up with smartphone power consumption demands. The three primary functional components of a lithium-ion battery are the negative electrode, positive electrode, and the electrolyte. Depending on materials choices for these components, the voltage, capacity, life, and safety of a lithium-ion battery can change dramatically. One variant of the Li-Ion battery is the lithium polymer (Li-Poly) battery, which uses a different type of electrolyte – a dry solid polymer electrolyte, which replaces the traditional porous separator, which is soaked with electrolyte. While Li-Poly batteries are more expensive than Li-Ion batteries, they offer several advantages, such as very low profiles, flexible form factors, light weight, and improved safety. Typical full charge time for Li-Ion and Li-Poly batteries is on the order of a few hours, and they

can be expected to have a battery lifetime of between 300 and 500 charging cycles (generally lasting between 2 and 3 years).

The dawn of 4G cellular networks is upon us. Around 2009, it became apparent that 3G networks would be overwhelmed by the growth of bandwidth-intensive applications like streaming media. 4G networks focus on the provision of fast internet, data, and voice services all around the world. 4G capability in smartphones is becoming more and more common, and because it is much more power intensive than previous network technologies, the power problem is only exacerbated.

Apart from cellular network technologies, smartphones contain other data and location interfaces such as WiFi and GPS. These interfaces also consume large amounts of power, both in their active and idle states. This can be seen in Figure 2.2, which shows the power distributions of the Google Nexus One Smartphone. Even when 3G, WiFi, and GPS interfaces are all enabled and idle, they account for more than 25% of total system power dissipation. Furthermore, when only one of the interfaces is active, the other two idle interfaces still consume a non-negligible amount of power.



**Figure 2.2: Google Nexus One smartphone power distributions**

Most smartphones today contain large touchscreen displays that allow easy viewing for full web pages, high-definition videos, e-books, etc. The two existing mainstream technologies for smartphone displays are LCD (Liquid Crystal Display) and OLED (Organic Light-Emitting Diode). OLED is a much newer technology than LCD and works by applying electric current to luminescent organic materials. OLED offers many advantages over LCD technology, including simpler construction, lower power consumption (due to the powered backlighting that LCD displays require), higher contrast ratio, and faster response times [61]. However, LCD is a more

mature technology, and is cheaper and easier to mass-produce. Both OLED and LCD displays are some of the main contributors to the power consumption challenges present in today's mobile devices.

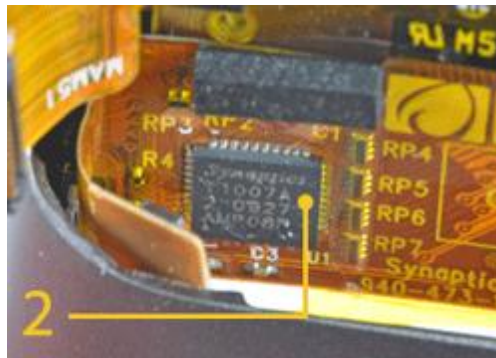
Smartphone CPU architecture is quickly advancing. The most common architecture used is the ARM (Advanced RISC Machine) architecture. ARM's relative simplicity makes it suitable for lower-power applications such as mobile and embedded electronics. However, processors in mobile devices are becoming more and more advanced to satisfy users' demands. For example, the ARM Cortex A9, which is used in Apple's iPad 2, is a dynamically scheduled superscalar processor with speculation. Multicore mobile devices are also becoming quite common. For smartphones, dual-core processors are the norm, and more and more quad-core devices are being developed. Some examples – the Samsung Galaxy Nexus's 1.2 GHz dual-core processor, the LG Optimus's 1.5 dual-core processor, the HTC Amaze 4G's 1.5 GHz dual-core processor, and the recently-released HTC One X's 1.5GHz quad-core Nvidia Tegra 3 processor.

## **2.1 Google Nexus One Overview**

To give an example of current smartphone technology, this section will briefly describe the various hardware elements of the Google Nexus One smartphone [63]. The Nexus One uses (1) a Samsung-manufactured OLED display, as shown in Figure 2.3, with (2) a Synaptics Clearpad 2000 touchscreen sensor and controller, shown in Figure 2.4.

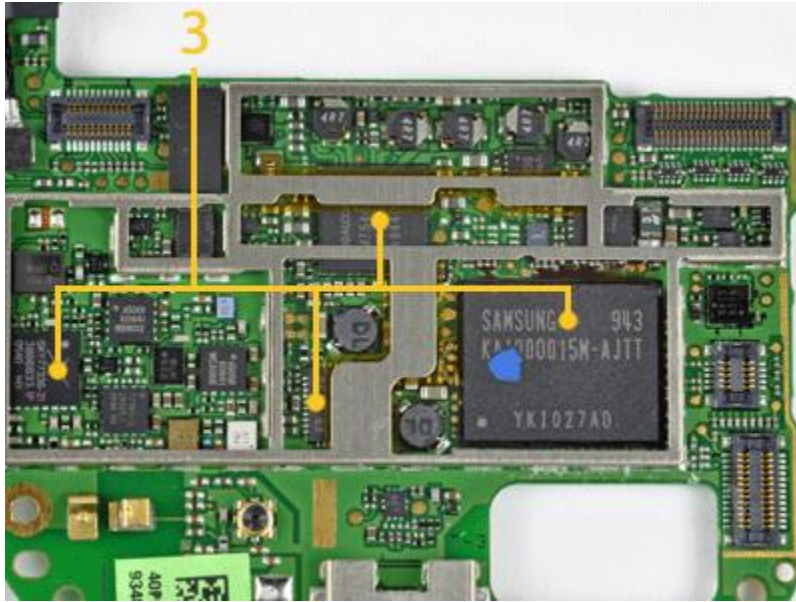


**Figure 2.3: Nexus One OLED display**



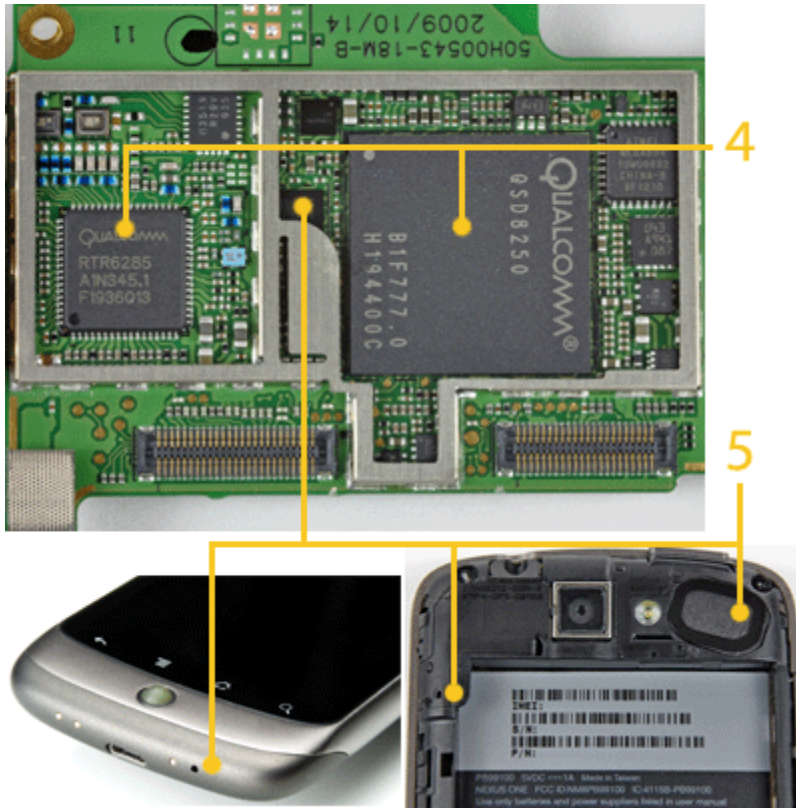
**Figure 2.4: Nexus One touchscreen sensor and controller**

Figure 2.5 shows the underside of the Nexus One's PCB, which contains 512 Mbytes of NAND-flash memory and 512 Mbytes of mobile SDRAM. Power-management ICs include Qualcomm's PM7540 and Texas Instruments' TPS65023, and Skyworks' SKY77336 provides the GSM power-amplifier function.



**Figure 2.5: Nexus One PCB underside**

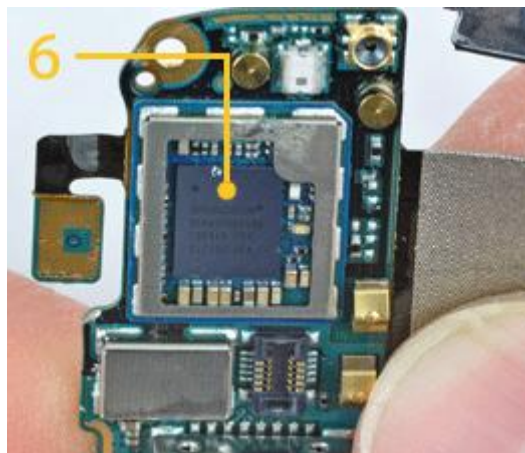
The Nexus One’s PCB topside, shown in Figure 2.6, includes a 1-GHz QSD8250 Snapdragon application processor, which integrates cellular modem and GPS (global-positioning-system) functions, along with a Qualcomm RTR6285 RF-transceiver IC. The QSD8250 is a variant of the ARM Cortex-A8 architecture, thus supporting the ARM Version 7 instruction set. Between the two Qualcomm devices is the Audience A1026 audio processor, also shown in Figure 2.6. This audio processor works in conjunction with two Knowles Electronics MEMS microphones. One microphone resides on the handset’s underside to capture the user’s voice, while the other is located on the back—to the left of the 5M-pixel still camera lens and associated flash—and captures ambient environmental sounds.



**Figure 2.6: Nexus One PCB topside and audio components**

Lastly, the Nexus One uses a Broadcom BCM4329 wireless transceiver, shown in Figure 2.6.

This transceiver supports IEEE 802.11n wireless networking data rates.



**Figure 2.7: Nexus One wireless transceiver**



## Chapter 3

### Problem Statement

As the previous chapter established – the combination of aging battery technology along with consumer demands for faster data acquisition and rigorous multitasking capability facilitates the need for better energy management strategies in mobile devices. The advent of 4G networks, power-hungry data and location interfaces, large backlit displays, and advanced CPU architecture all supplement this need. These energy management strategies often manifest themselves as software-centric algorithms and strategies, which can be implemented alongside more well-known hardware strategies.

When devising a software energy-optimization strategy, a key consideration is the quality of service (QoS) that is provided to the user. While energy management is important, it is also very important to retain the original functionality and responsiveness of the device. One solution to this is to exploit user *contexts* to **conserve energy while still offering acceptable QoS**. *Context-awareness*, or being conscious to data such as user location, time of day/week, application interaction, and device attributes, etc. can help energy optimization strategies minimize device energy consumption more intelligently.

The components that are available for tuning for energy optimization include the following: (1) the processor, (2) the screen backlight, (3) the device sensors (i.e. proximity and ambient light sensors, accelerometer, etc.), (4) the location interfaces (i.e. GPS and WiFi/network), and (5) the wireless communication (data) interfaces. The main method used for **processor** optimization is DVFS. However, when scaling processor frequency and voltage, user perception and cognitive delay must be carefully considered so that the user's experience is not ruined. Application-aware

methods can be used to scale processor frequency and voltage according to what the user is doing at a particular point in time. User perception must also be considered when optimizing **screen backlight** energy. The screen backlight must be bright enough for the user to see easily, which depends on ambient lighting (whether the user is indoors or outdoors) and the application that is currently running. In darker areas, or when the user is using brighter applications, videos, etc., the backlight may be able to be reduced without user perception. For the **device sensors**, energy consumption can be reduced by configuring sampling frequency, or by using more energy-efficient sensors (known as *sensor substitution*). Often, the more energy efficient sensors are less accurate, so there is a tradeoff. Another method is to synchronize the sensing patterns of the various sensors, allowing the device to “wake up” less often (known as *sensor piggybacking*). **Location interface** energy consumption can be reduced by choosing whether to use GPS or the WiFi/network interfaces intelligently. The WiFi and network location interfaces use WiFi AP or cell tower triangulation to determine user location. This method is less accurate, but more energy efficient than using GPS. Another option is to use other device sensors such as the accelerometer, bluetooth, or ambient light and sound sensors to help determine location. Finally, to optimize **wireless communication interface** energy, we can choose when to use which interface, which include WiFi and the cellular network interfaces (GSM, CDMA, UMTS, 4G, etc.). One of the most important considerations when choosing which wireless communication interface to use is the signal strength of the various interfaces.

The goal of this thesis is to optimize mobile device energy consumption via two separate strategies, while considering user perception, ensuring that user QoS is not noticeably impacted. The first strategy minimizes **processor** and **screen backlight** energy consumption by scaling CPU frequency and screen brightness during idle time between user interactions. The second

strategy minimizes **location interface** and **wireless communication interface** energy consumption by dynamically predicting what interfaces will be needed at a point in time, and then disabling the unnecessary ones.

## Chapter 4

### Related Work

A large amount of work has been done in the area of energy optimization for mobile devices in recent years. This work can be categorized into the following: (1) work on mobile usage studies, (2) work on CPU and backlight energy optimization, (3) work on energy optimization of wireless interfaces for data transfers and location sensing, (4) usage of machine learning techniques in mobile devices, and (5) hardware strategies for energy optimization.

#### **4.1 Work on Mobile Usage Studies**

A great deal of research is dedicated to mobile usage studies to understand how users interact with their phones in the real world. In [4] the authors demonstrated from a two-month smartphone usage study that all users have unique device usage patterns. They present a usage pattern analysis using log data collected from smartphones, and claim that their results can be used to access network selection and energy efficient communication. Eberle et al. [5] present the results of an energy measurement campaign that was conducted on different sensors and positioning methods on a smartphone. They attempt to provide a guideline for researchers and developers to design more accurate and energy efficient algorithms for continuous location tracking. Balasubramanian et al. [6] present a measurement study of the energy consumption characteristics of 3G, GSM (2G), and WiFi. The authors find that 3G and GSM networks incur a high “tail energy” overhead because of lingering in high power states after completing a transfer, then, using this model, they develop a protocol to reduce energy consumption of common mobile applications. Finally, MyExperience [7] gathers traces from user phones in the wild by logging more than 140 event types, including: 1) device usage such as communication, application

usage, and media capture, 2) user context such as calendar appointments, and 3) environmental sensing such as Bluetooth and GPS, and then uses the traces to study high-level user actions.

#### **4.2 Work on CPU and Backlight Energy Optimization**

The first strategy presented in this thesis focuses on optimizing CPU and backlight energy consumption. There have been some other efforts to optimize CPU and backlight energy consumption in recent years. Shye et al. [8] in particular make several important observations: (1) the screen and the CPU are the two largest power consuming components, and (2) users are generally more satisfied with a scheme that gradually reduces screen brightness rather than one that changes abruptly. Based on the second observation, the authors implement a scheme that utilizes *change blindness* [41], [42] by slowly reducing screen brightness and CPU frequency over time. However, their approach does not consider application-aware and user-aware scaling like the first strategy in this thesis does, because of which their approach tends to be overly conservative at times, and at other times can often detrimentally impact user QoS. Other researchers have experimented with screen optimizations that would be enabled with OLED display technology by altering the user interface to dim certain parts of the screen to save considerable power, then conducting user acceptance studies [28], [29]. The first strategy in this thesis is based on current mobile device screen technology. Bi et al. propose a user interaction-aware DFS approach in [30] much like the strategy in this thesis. However, their approach is directed towards stationary desktop systems, and not towards the touch-based mobile devices that are becoming more prevalent today. Their proposed approach is also different in that it does not consider backlight optimization, and uses a simple user-aware but application-unaware history predictor to set CPU frequency levels based on demand. PICSEL [31] is another user-aware CPU DFS scheme that is directed towards desktop and laptop machines. Still other DFS

schemes are application-aware and observe application behavior with respect to CPU utilization and memory references to construct statistical models and apply voltage/frequency schedules based on these models [32], [33], or detect regions of low CPU utilization and insert instructions to control the switching of voltage/frequency levels [34]-[37].

### **4.3 Work on Energy Optimization for Data and Location Interfaces**

The second strategy presented in this thesis focuses on reducing energy consumed by the device's data and location interfaces by intelligently selecting which interfaces to enable at certain times. Much of the related work pertaining to mobile device energy optimization concentrates on determining the most energy-efficient data interface (e.g. 3G/EDGE, WiFi). In [9] an energy-aware handoff algorithm is proposed based on the energy consumption of UMTS (3G) and WiFi. The energy costs for transmitting data over different wireless interfaces are explored by Ra et al. [10], with the goal of balancing energy and delay during data transfers.

Other work [11]-[26] focuses on energy-efficient location-sensing schemes aiming to reduce high battery drain caused by location interfaces (e.g. WiFi, GPS). Lee et al. [17] propose a Variable Rate Logging (VRL) mechanism that disables location logging or reduces the GPS logging rate by detecting if the user is standing still or indoors. Nishihara et al. [18] propose a context-aware method to determine the minimum set of resources (processors and peripherals) that result in meeting a given level of performance, much like the second strategy in this thesis. They determine if a user is moving/stationary and indoors/outdoors and control resources using a static lookup table. In contrast, the strategy in this thesis controls resources dynamically by using machine learning algorithms that are trained on real user activity traces. A framework for mobile sensing that efficiently recognizes contextual user states is proposed by Wang et al. in [19].

Zhuang et al. [20] propose an adaptive location-sensing framework that involves substitution, suppression, piggybacking, and adaptation of applications' location-sensing requests to conserve energy. Their work is directed towards LBAs (location-based applications) and only focuses on location interfaces, while the second strategy in this thesis is a system-wide optimization strategy that is capable of saving energy regardless of the foreground application type.

#### **4.4 Usage of Machine Learning Techniques in Mobile Devices**

A substantial amount of research has been dedicated to utilizing machine learning algorithms in mobile devices. Cheung et al. [21] use Markov decision processes to prolong mobile phone battery lifetime by maximizing a user-defined reward function. Jung et al. [43] propose a supervised learning based power management framework that utilizes a method of Bayesian classification. Batyuk et al. [45] extend a traditional self-organizing map to provide a means of handling missing values, then use it to predict mobile phone settings such as screen lock pattern and WiFi enable/disable. Other works attempt to predict the location of mobile users using machine learning algorithms. In [46] the authors propose a model that predicts spatial context through supervised learning, and the authors in [47] take advantage of signal strength and signal quality history data and model user locations using an extreme learning machine algorithm. These works are focused on using user context for device self-configuration and location prediction, whereas the second strategy in this thesis is focused on using user context for optimizing energy consumed by both data transfer and location interfaces.

#### **4.5 Hardware Strategies for Energy Optimization**

Lastly, a large amount of work is dedicated to hardware optimizations to minimize energy consumption. Swanson et al. [52] take advantage of a phenomenon known as *dark silicon*, or the

areas of a chip's silicon that must remain mostly passive in order to stay within the chip's power budget, by filling it with specialized cores in order to save energy in common applications. Duan et al. [64] explore the applicability of RAM optimizations for smartphone platforms, then apply these optimizations to Phase Change Memory (PCM) and study their energy efficiency and performance. They also propose a hybrid approach using both mobile RAM and PCM. Finally, Lin et al. [65] present a suite of compiler and runtime techniques that allow developers to leverage low-power processors that can accomplish frequent, simple tasks with high efficiency.



## Chapter 5

### Machine Learning Techniques

The notion of searching for patterns and regularities in data is the fundamental concept in the field of pattern recognition and data classification. Machine learning is often focused on the development and application of computer algorithms in this field [48]. The focus of machine learning is be able to recognize complex data patterns, and then make good and useful approximations based on those patterns. Some examples of machine learning applications are learning applications, classification, regression, unsupervised learning, and reinforcement learning. Classification and regression are known as *supervised learning* problems, where there is an input and an output, and the task is to learn the mapping from the input to the output. *Unsupervised learning* problems, on the other hand, we only have input data, and the aim is to find regularities in the input. In some cases, the output of the system is a sequence of actions, and the sequence of correct actions required to reach the goal, known as the *policy*, is most important. In these cases, *reinforcement learning* algorithms are used to learn from past good action sequences and generate a good policy. All of these types of machine learning algorithms attempt to perform accurately on new, unseen examples after *training* on a finite data set. The following sections discuss the details of the machine learning techniques used in this thesis.

#### 5.1 Markov Decision Processes

Markov Decision Processes (MDP) [21] are discrete time stochastic control processes that are widely used as decision-making models for systems in which outcomes are partly random and partly controlled. They are often used in decision making, artificial intelligence or control theory. In each time step in an MDP, the process is in some state  $s$ , and a decision maker may choose any

action  $a$  available from state  $s$ . An example of an MDP with 3 states and 2 actions is shown in Figure 5.1. MDPs consist of a four-tuple  $(S, A, P_a, R_{s,a})$ , where  $S$  is a finite number of states,  $s \in S$  is the current state,  $A$  is a finite set of actions available from each state,  $a \in A$  is the action taken based on the current state,  $P_a(s)$  is the probability that action  $a$  in state  $s$  at time  $t$  will lead to state  $s^*$  at time  $t + 1$ , and  $R_{s,a}$  is a reward from transitioning to state  $s^*$  from state  $s$ . In Figure 5.1, the numbers on each line are the state transition probabilities. The objective of solving an MDP is to find the policy that maximizes a measure of long-run expected rewards.

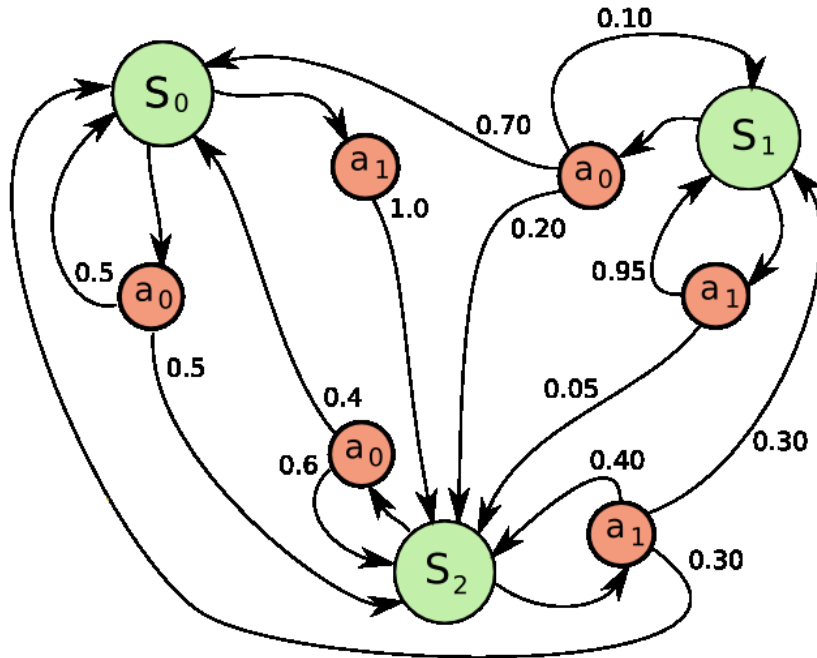


Figure 5.1: Example of MDP with 3 states and 2 actions

## 5.2 Bayesian Classification

Bayesian learning [49] is a form of supervised learning that involves using evidence or observations along with prior outcome probabilities to calculate the probability of an outcome.

More specifically, the probability that a particular hypothesis is true, given some observed evidence (the posterior probability of the hypothesis), comes from a combination of the prior probability of the hypothesis and the compatibility of the observed evidence with the hypothesis (the likelihood of the evidence). The first strategy in this thesis uses a form of Bayesian learning based on the classification method executed by Jung et al. [43]. This method of classification involves mapping a set of input features,  $X = \{x_1, x_2, \dots, x_n\}$ , to a set of output measures,  $Y = \{y_1, y, \dots, y_n\}$ . Learning for the classifier is accomplished through creation of a training set of size  $\gamma_{TS}$ . Finding a consistent mapping from input features to an output measure using this training set enables the classifier to accurately predict the class of a set of input features.

The classification task consists of calculating posterior probabilities for each class, based on the prior probability, class likelihood, and evidence, then choosing the class with the highest posterior probability. For two input features, the posterior probabilities are obtained using the following equation:

$$\text{posterior probability} = \frac{\text{likelihood} \times \text{prior probability}}{\text{evidence}} \quad (5.1)$$

↓

$$\text{Prob}(y = c|x_1, x_2) = \frac{\text{Prob}(x_1, x_2|y = c) \times \text{Prob}(y = c)}{\text{Prob}(x_1, x_2)} \quad (5.2)$$

The denominator  $\text{Prob}(x_1, x_2)$ , or the evidence, is the marginal probability that an observation  $X$  is seen under all possible hypotheses, and is irrelevant for decision making since it is the same for every class assignment [43].  $\text{Prob}(y = c)$  is the prior probability of the hypothesis that the application class  $y$  is  $c$ . This is easily calculated from the training set.  $\text{Prob}(x_1, x_2|y = c)$  is the

per-input-feature conditional probability of seeing the input feature vector  $X$  given that the application class  $y$  is  $c$ . From Jung et al. [43] we have:

$$Prob(x_1, x_2|y = c) = Prob(x_1|y = c) \times Prob(x_2|y = c) \quad (5.3)$$

Thus, the posterior probability of a class may be computed as follows:

$$Prob(y = c) = Prob(x_1, x_2|y = c) \times Prob(y = c) \quad (5.4)$$

There may be some cases in which input features do not occur together with an output measure due to an insufficient number of data points in the training set, resulting in a zero conditional probability. To deal with this, we eliminate them by smoothing:

$$Prob(x_i|y = c) = \frac{freq(x_i, y = c) + \lambda}{freq(y = c) + \lambda n_x} \quad (5.5)$$

where  $\lambda$  is a smoothing constant ( $\lambda > 0$ ), and  $n_x$  is the number of different attributes of  $x_i$  that have been observed.

### 5.3 Q-Learning

Q-Learning is a reinforcement learning technique that does not require a model of the environment. The Q-Learning algorithm creates a model by exploring the environment. For the exploration phase, one possibility is to use the epsilon-greedy ( $\epsilon$ -greedy) search, in which we choose an action randomly out of all possible actions with probability  $\epsilon$ , and choose the best action with probability  $1 - \epsilon$  [51]. The cost of a single action taken from a state is the reinforcement for that action from that state. The value of that state-action is the expected value of the full return, or the sum of the reinforcements that will follow when that action is taken [49]. This state-action value function is called the Q function because it calculates the quality of a state-action combination. Given the current state,  $s_t$ , an action,  $a_t$ , the reinforcement for taking

action at in state  $s_t$ ,  $r_{t+1}$ , the next state,  $s_{t+1}$ , and the next action,  $a_{t+1}$ , new Q function values are calculated for each possible state-action pair using the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (r_{t+1} + \gamma + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (5.6)$$

where  $0 < \alpha \leq 1$  is the learning rate and  $0 \leq \gamma < 1$  is the discount factor. The learning rate determines to what extent the old Q values are overridden. A learning rate of 0 will not update the old Q values at all, while a learning rate of 1 will only consider the most recent information. The discount factor determines the importance of future rewards. A factor of 0 will make the algorithm opportunistic, considering current rewards more importantly, and a factor of 1 will make the algorithm strive for a higher long-term reward.

#### 5.4 Linear Discriminant Analysis

Linear discriminant analysis (LDA) makes use of a Bayesian approach to classification in which parameters are considered as random variables of a prior distribution. This concept is fundamentally different from data-driven linear and non-linear discriminant analyses in which what is learned is a *function* that maps or separates samples to a class. Bayesian estimation and the application of LDA is also known as *generative modeling*, in that what is learned is a probabilistic model of the samples from each class. By considering parameters as random variables of a prior distribution one can make use of prior known information. For example knowing that a mean  $\mu$  is very likely to be between  $a$  and  $b$ , the probability can be determined in such a way that the bulk of the density lies between  $a$  and  $b$  [49]. Given a prior probability distribution and a class likelihood, Bayes' theorem (equation 5.7) can be invoked to get an inferred posterior probability to derive a class prediction ( $C_k$ ) for a new observed sample  $x_n$  using a *maximum a posteriori* (MAP; equation 5.8) [49]:

$$p(C_k|x_n) = \frac{p(C_k)p(x_n|C_k)}{p(x)} \quad (5.7)$$

$$\operatorname{argmax}_c p(C_k|x_n) \quad (5.8)$$

LDA is applicable to a wide range of classification problems of both univariate or multivariate input spaces and binary- or multi-class classification. A number of statistical distribution functions can be applied, but the most common is the *Gaussian* or *Normal* distribution (which we use in our study) as shown in equation 9 below.

$$N(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x - \mu)^2\right\} \quad (5.9)$$

There are a number of reasons the Gaussian distribution is a prominent model used in several fields of study, including natural and social sciences, but a simplistic view is that the shape of the Gaussian distribution, the well-known bell shape, is often an acceptable model for complex and partly random phenomena. This notion is brought about by the fact that the shape of the Gaussian is controlled by the mean  $\mu$  and covariance  $\sigma^2$ . The location of the mean (or the peak of the curve) gives an expectation of where random variables tend to cluster and the covariance (or width of the curve) gives an indication of the variability of the data.

## 5.5 Linear Logistic Regression

Similar to LDA, linear logistic regression (LLR) is a technique used to derive a linear model that directly predicts  $p(C_k|x_n)$ , however it does this by determining linear boundaries that maximize the likelihood of the data from a set of class samples instead of invoking Bayes' theorem and generating probabilistic models from priori information. LLR expresses  $p(C_k|x_n)$  directly by requiring all linear function values to be between 0 and 1 and that they all sum to 1 for any value of  $x_n$ , as shown in equation 5.10:

$$p(C_k|x_n) = \frac{f(x_n, \beta_k)}{\sum_{m=1}^K f(x_n, B_m)} \quad (5.10)$$

With LDA, Bayes' theorem, class priors, and class probability models were used to infer the class posterior probabilities, which were then used to discriminate between the different classes for a given sample  $x_n$ . In contrast, LLR solves for the linear weight parameters,  $\beta_k$ , directly using gradients to maximize the data likelihood. This is done by enumerating the likelihood function,  $L(\beta)$ , using a 1-of-K coding scheme for the target variables, as shown in equations 5, 6 [48], in which every value of  $t_{n,k} \in \{0, 1\}$  and each row only contains a single '1'. The class variable transformations are known as indicator variables and are used in the exponents of the likelihood function to select the correct terms for each sample  $x_n$ .

$$C = \{k_1, k_2, k_3, \dots, k_n\}$$

↓

$$\begin{pmatrix} t_{1,1} & t_{1,2} & \dots & t_{1,k} \\ t_{2,1} & t_{2,2} & \dots & t_{2,k} \\ \vdots & & & \\ t_{n,1} & t_{n,2} & \dots & t_{n,k} \end{pmatrix} \quad (5.11)$$

$$L(\beta) = \prod_{n=1}^N \prod_{k=1}^K p(C_k|x_n)^{t_{n,k}} \quad (5.12)$$

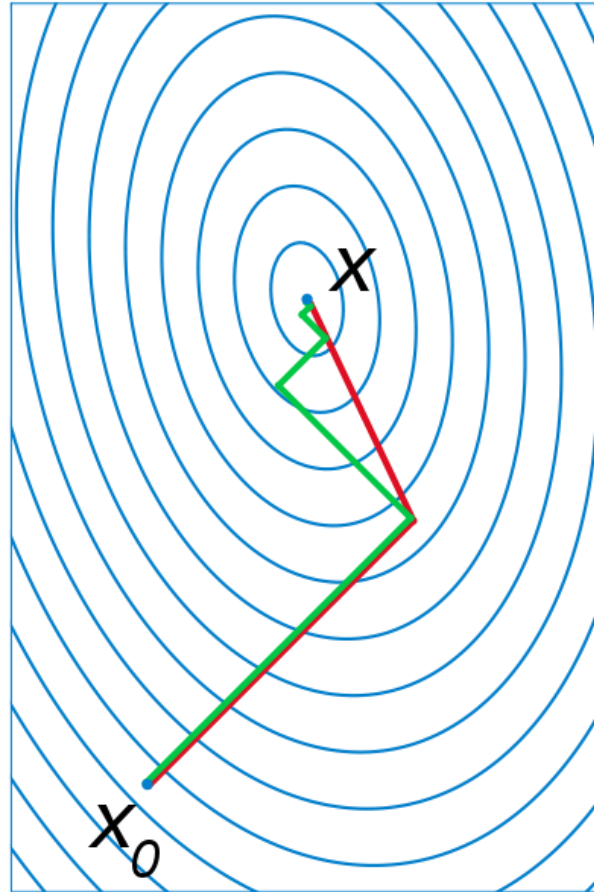
In order to find the  $\beta$  that maximizes the data likelihood, the *product of products* is transformed to a *sum of sums* using the natural logarithm to simplify the gradient calculation with respect to  $\beta$ . Since equation 5.13 is non-linear, iterative methods like gradient descent or scaled conjugate gradient [49] can be used to solve for the gradient of the log likelihood,  $\nabla_{\beta} LL(\beta)$ , and obtain the respective  $\beta$  weights.

$$LL(\beta) = \sum_{n=1}^N \sum_{k=1}^K t_{n,k} \log p(C_k|x_n) \quad (5.13)$$

### 5.5.1 Scaled Conjugate Gradient

Simple gradient descent algorithms use a fixed step size  $\delta$  when following a gradient. However, when fixed it is difficult to choose an optimal value for  $\delta$ , which may result in slow convergence times. Instead, it is better to perform a series of one-dimensional iterative searches known as *line searches* in the direction of the gradient to choose  $\delta$  in each iteration. Although using line searches to choose  $\delta$  is better than using a fixed step size, there are a few problems associated with the resulting gradient descent algorithm. For example, because the gradient descent directions interfere, a minimization along the gradient in one direction may spoil past minimizations in other directions. This problem is solved using *conjugate gradient* methods, which compute non-interfering conjugate directions. Figure 5.2 shows an example of the two algorithms, gradient descent (red line) and conjugate gradient (green line) beginning at point  $x_0$ , then moving along the gradient to find a minimum at point  $x$ .





**Figure 5.2: Comparison of gradient descent (red) and conjugate gradient (green) algorithms**

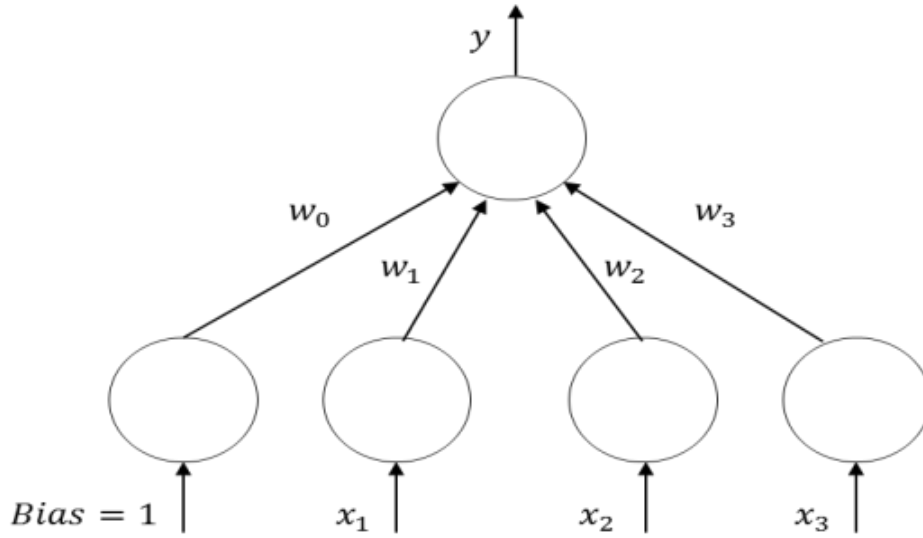
Standard conjugate gradient algorithms still use line searches along the conjugate directions to determine step size. However, there are several drawbacks to doing line searches that can be detrimental to the performance of the algorithm, such as the error calculations involved each iteration. *Scaled conjugate gradient* (SCG) algorithms substitutes the line search by *scaling* the step size  $\delta$  depending on success in error reduction and goodness of a quadratic approximation of the likelihood [50].

## 5.6 Non-Linear Logistic Regression with Neural Networks

Neural network models, also known as *Artificial Neural Networks*, are inspired by the way the human brain is believed to function. Many of the normal basic everyday information processing requirements handled by the brain, for example sensory processing, cognition, and learning, surpass any capable computing system out there today. Although a human brain is quite different than today's computing hardware, it is believed that the basic concepts still apply in that there is a computational unit, known as a *neuron*, and connections to memory stored in *synapses*. The main difference being that the human brain consists of billions of these simple parallel processing units, neurons, which are interconnected in a massive multi-layered distributive network of synapses and neurons [49].

In machine learning, these concepts are modeled as what is called a *perceptron*, the basic processing element, connected by other *perceptrons* through weighted connections, as illustrated in Figure 5.3. The output of a perceptron is simply a weighted sum of its inputs including a weighted bias, as shown in equation 5.14.

$$y = \sum_{i=1}^n w_i x_i + w_0 \quad (5.14)$$



**Figure 5.3: Neural network perceptron model**

To compute the output  $y$  given a sample  $x_i$ , *backpropagation* using the gradient with respect to the weights is performed using a training dataset to find the weight parameters,  $w_i$ , that minimize the mean squared error between the neural network outputs,  $y_i$ , and the target outputs,  $t_i$ . By default, the neural network consists of a hyperplane (for multiple perceptrons) that can be used as a linear discriminant to linearly separate the classes. To improve prediction accuracy, we make it non-linear, by applying a sigmoidal or hyperbolic tangent to hidden unit layer perceptrons, as denoted in equation 5.15. This allows for non-linear boundaries with the output of the neural network being linear in the weights, but non-linear in the inputs.

$$y'_i = \text{sigmoid}(y_i) = \frac{1}{1 + \exp(\mathbf{w}^T \mathbf{x})} \quad (5.15)$$

For classification with a neural network (non-linear logistic regression), the number of parallel output perceptrons is equal to the number of classes. The output from each perceptron,  $y_i$ , is then sent to post processing as in equation 5.16 to determine the respective class by taking the maximum of the post-processed outputs:

$$C_i \text{ if } y_i = \max_i \frac{\exp(y_i)}{\sum_i \exp(y_i)} \quad (5.16)$$

One of the biggest criticisms about the use of neural networks is the time required for training. Although this can be a major issue if using a simple gradient descent approach, newer training techniques, such as the scaled conjugate gradient (SCG) [50], can greatly minimize the time required for training. SCG, a method for efficiently training feed-forward neural networks, was used for training the neural networks in this thesis.

### 5.7 K-Nearest Neighbor

The k-nearest neighbor (KNN) algorithm is a fairly simple non-parametric unsupervised approach for the data classification problem. A key assumption of non-parametric estimation is that similar inputs have similar outputs [48]. In KNN, new samples are classified by assigning them the class that is the most common among the  $k$  closest samples in the attribute space. This method requires some form of distance measure for which Euclidean distance is typically used. The Euclidean distance between two points  $a$  and  $b$ , each containing  $i$  attributes, is defined in equation 5.17.

$$d(a, b) = \sqrt{\sum_{i=1}^n (b_i - a_i)^2} \quad (5.17)$$

Let us look at an example. Figure 5.4 shows a data set, characterized as blue squares and red triangles. The green circle is a new sample that needs to be classified as either a blue square or as a red triangle. If  $k = 3$  (represented by the smaller inner circle with radius 3), the new sample is classified as a red triangle because there are more red triangles within the considered area. Similarly, if  $k =$

5 the new sample is classified as a blue square.

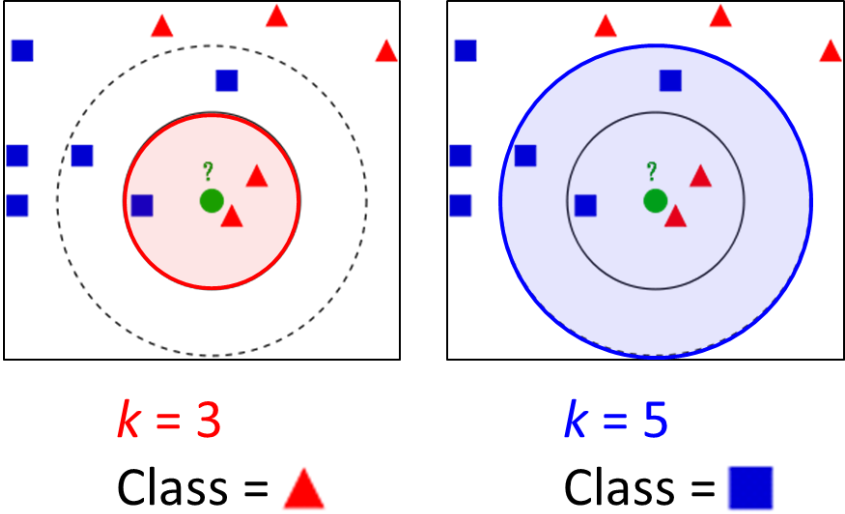


Figure 5.4: K-nearest neighbor example

## Chapter 6

### Strategy 1: CPU and Backlight Energy Optimization

This chapter presents the first energy-optimization strategy for pervasive mobile devices in this thesis – an application and user-interaction aware energy management framework, named *AURA*. *AURA* takes advantage of user idle time between user interaction events with the foreground application to optimize CPU and backlight energy consumption. Overall, this strategy makes the following novel contributions:

- Usage studies with real users are conducted and a Bayesian application classifier tool is developed to categorize mobile applications based on user interaction activity
- An integrated MDP/Q-Learning-based application and user interaction-aware energy management framework is developed that adapts CPU and backlight levels in a mobile device to balance energy consumption and user QoS
- Backlight and CPU power dissipation are characterized on Android OS based HTC Dream and Google Nexus One mobile architectures
- The framework is implemented as middleware running on the HTC Dream and Google Nexus One smartphones, and real energy savings are demonstrated on commercial apps running on the devices

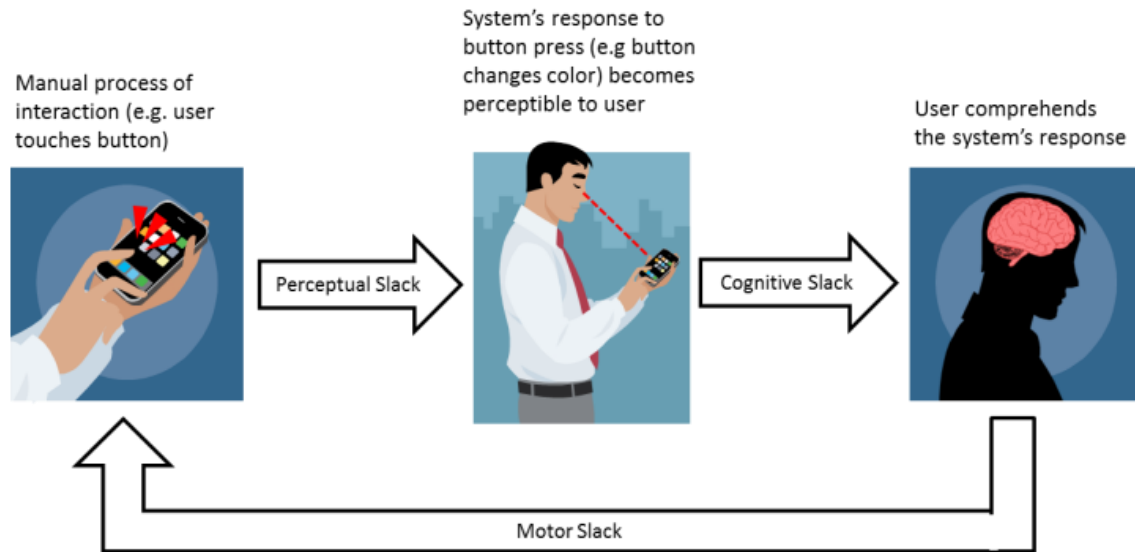
Section 6.1 first describes some fundamental observations that lay the foundation for energy savings in mobile devices. Section 6.2 presents the results of field studies involving users interacting with apps on real mobile devices. Section 6.3 gives a high level overview of the *AURA* energy management framework. Sections 6.4-6.6 elaborate on the major components of the framework. Section 6.7 presents the power models used to quantify the energy-effectiveness

of the proposed framework. Section 6.8 presents the experimental methodology. Finally, Section 6.9 presents the energy savings and performance results.

## 6.1 Fundamental User-Device Interaction Mechanisms

Here the underlying concepts stemming from the psychology of user-device interactions that drive the CPU and backlight energy optimizations in the *AURA* framework are explained.

There are three basic processes involved in a user's response to any interactive system [40], such as a smartphone or personal computer. During the *perceptual* process, the user senses input from the physical world. During the *cognitive* process, the user decides on a course of action based on the input. Finally, during the *motor* process, the user executes the action by mechanical movements. These three processes are consecutive in time and can be characterized by an *interaction slack*. For instance, when interacting with an app on a mobile device, the time between when a user interacts with an app (e.g., touching a button) and when a response to that interaction is perceptible to the user (e.g., the button changes color) is the perceptual slack; the period after the system response during which the user comprehends the response represents the cognitive slack; and finally the manual process of the next interaction (e.g. moving finger to touch the screen) involves a motor slack. Figure 6.1 illustrates this process. Before the user physically touches the input peripherals, the system is idle during the cumulative slack period, for hundreds to possibly many thousands of milliseconds. During this time, if the CPU frequency can be reduced, energy may be saved without affecting user QoS. **CPU energy** optimizations in *AURA* exploit this inherent slack that arises whenever a user interacts with a mobile device.



**Figure 6.1: Interaction slack process**

Another interesting phenomenon that can open up new possibilities for energy optimizations is the notion of *change blindness* [41] as revealed by researchers in human psychology and perception. Change blindness refers to the inability of humans to notice large changes in their environments, especially if changes occur in small increments. Many studies have confirmed this limitation of human perception [42] – a majority of observers in one study failed to observe when a building in a photograph gradually disappeared over the course of 12 seconds; in another study, gradual color changes over time to an oil painting went undetected by a majority of subjects but disruptive changes such as the sudden addition of an object were easily detected. By gradually reducing screen brightness over time under certain scenarios, energy consumption in mobile devices may thus be reduced without causing user dissatisfaction. This principle is exploited by *AURA* to optimize screen **backlight energy**.



## 6.2 User-Device Interaction Field Study



















To understand how users interact with mobile applications in the real world, a field study with users running a variety of apps on a popular smartphone was conducted. It is well-known that apps running on smart mobile devices can invoke vastly different interaction behaviors from users – whereas some applications require constant interaction with the user, others may experience bursts of user interaction followed by long idle periods. These unique interaction patterns make conventional general-purpose energy management strategies ineffective on mobile devices. We hoped to uncover trends from the field studies that would potentially guide scenario-specific energy optimizations.

18 Android apps were selected from the following categories: games, communication, multimedia, shopping, personal/ business, travel/local, and social networking. A custom background interaction logger app was developed to record interaction traces for users running each of these apps. The study involved five different users interacting with the apps over the course of a day on an Android OS based Google Nexus One smartphone. The users selected for the study spanned the proficiency spectrum: user 3 for instance mainly used his smartphone for phone calls, and was not familiar with many of the selected applications, whereas user 2 relied on her smartphone for a variety of tasks, and was quite comfortable while interacting with the chosen applications.

Table 6.1 shows a summary of the interaction logged for the users for each of the selected apps. The patterns in the table are classified based on interaction frequency (very-low, low, low-medium, medium, medium-high, high, very-high). An obvious observation we can immediately make from these results is that although some apps belong to the same category (e.g., *Sudoku* and *Jewels* are both games) they possess vastly different interaction characteristics. It is also

interesting to see that user-device interactions for some apps are very **application-specific**. For example, *Jewels*, a fast-paced gaming application, is always classified as very-high-interaction, and *Music* is almost always classified as very-low-interaction because users usually select a song then cease interaction. On the other hand, some classifications are noticeably **user-specific**, meaning that they may be classified quite differently based on the type of user interacting with them. Interaction results for *Minesweeper*, *Gallery*, and *News & Weather* all illustrate this idea.

**Table 6.1: App interaction classification**

APPLICATION		CLASSIFICATION				
		User 1	User 2	User 3	User 4	User 5
	1: Market (com.android.vending)	Med High	Med	Med	Low Med	Med
	2: Gallery (com.cooliris.media)	Med High	Low	Med High	Low Med	Med
	3: Jewels (org.mhgames.jewels)	Very High	Very High	Very High	Very High	Very High
	4: Wordfeud FREE (com.hbwares.wordfeud.free)	Very Low	Very Low	Low	Low Med	Very Low
	5: Facebook (com.facebook.katana)	High	Med High	High	Med High	Med
	6: Gmail (com.google.android.gm)	Med	Low	Low Med	Low Med	Low Med
	7: SMS (com.android.mms)	Very High	Very High	Very High	Very High	Very High
	8: Browser (com.android.browser)	High	Very High	Med	Med High	High
	9: News & Weather (com.google.android.apps.genie.geniewidget)	Low Med	Low	Low Med	Med High	Med
	10: YouTube (com.google.android.youtube)	Very Low	Very Low	Very Low	Very Low	Very Low
	11: Calculator (com.android.calculator2)	Very High	Very High	Very High	Very High	Very High
	12: Twitter (com.android.twitter)	Med High	Med	Med High	Med High	Med
	13: Calendar (com.android.calendar)	Med	Med High	Low Med	Low Med	Med
	14: Google Maps (com.google.android.apps.maps)	Med	Very High	High	High	High
	15: Sudoku (cz.romario.opensudoku)	High	Low	Very High	High	Low Med
	16: Music (com.android.music)	Very Low	Very Low	Very Low	Low	Very Low
	17: Solitaire (com.softick.android.solitaire.klondike)	Very High	High	Very High	High	Very High
	18: Minesweeper (artfulbits.aiMinesweeper)	Med High	Med High	High	Very High	Med

Next, a study was conducted to examine the variability in interaction patterns for different applications. By averaging the times between touch events for each user's interaction sessions from the previous study, we obtained interaction distributions for eight common applications, shown in Figure 6.2. In the figure, the circle sizes represent how frequently the between-touch times occurred. For example: WordFeud's large circles at 1 and 7+ indicate that most of the touch events in its interaction pattern had either 1 second or greater than 7 seconds separating them. Note that the times between touch events are rounded to the nearest second. Looking at the figure, it is clear that the interaction patterns for the *Market* and *Gallery* applications are quite

variable. Alternatively, the interaction patterns for the *SMS* and *Jewels* applications are fairly invariable, exhibiting a large percentage of shortly spaced touch events.

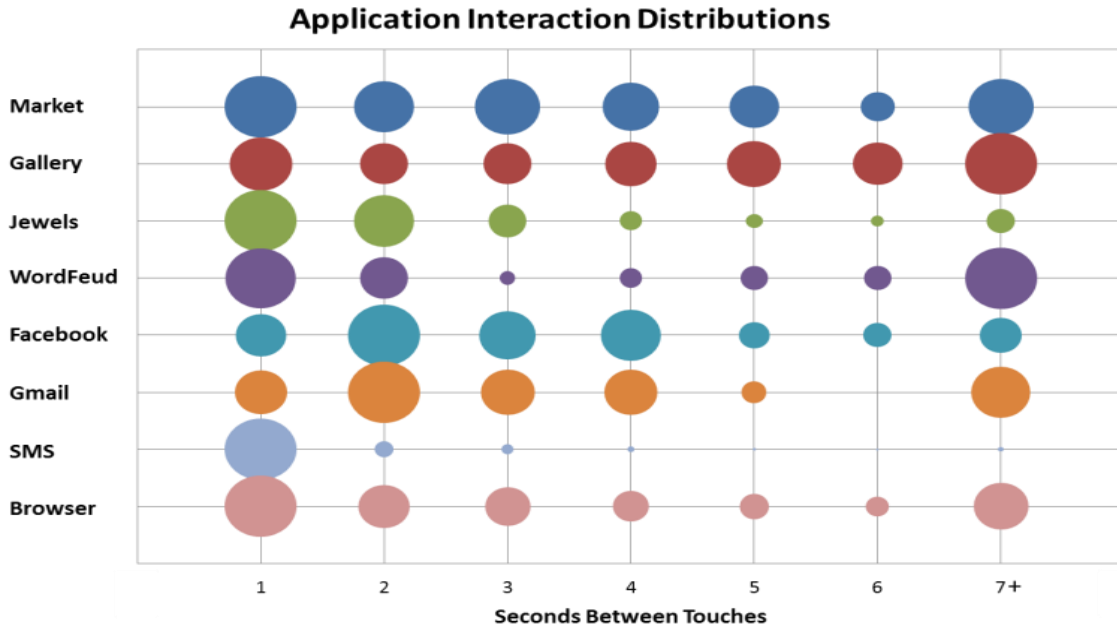


Figure 6.2: Application interaction distributions

To further the analysis of the variability of the interaction patterns for different applications, the occurrences of actual touch events over two minute intervals for the eight common applications we plotted, shown in Figure 6.3. This allows visualization of the “bursty” interaction that some applications display, e.g. *WordFeud*, *SMS*, and *Gmail*. In addition, Figure 6.3 also shows the variability of *Market*, *Gallery*, *Facebook*, and *Browser*, and strengthens the claims that *Jewels* and *SMS* are very-high-interaction applications, while *WordFeud* is a very-low-interaction application.

## Application Touch Event Timelines

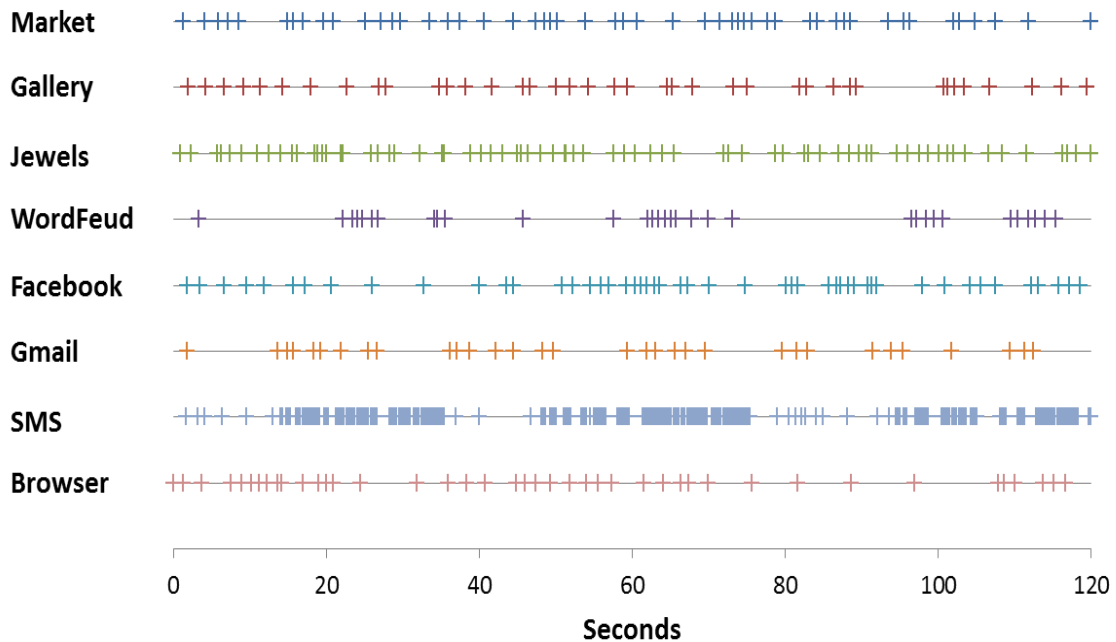


Figure 6.3: Application touch event timelines

We can thus conclude that users interact with devices in a user-specific and application-specific manner. Any framework that would attempt to exploit interaction slack and change blindness to save energy (as discussed in the previous section) must tailor its behavior uniquely across users and applications.

### 6.3 *AURA* Middleware Framework Overview

A high level overview of the *AURA* energy optimization framework for mobile devices is presented in this section. Figure 6.4 shows the *AURA* framework that is implemented at the middleware level in the software stack of the Android OS [53], and runs in the background as an Android Service. As the user switches between applications, the *AURA runtime monitor* will receive a “Global Focus Event,” and update any relevant session information of the previous

foreground application to an **app profile database**, stored in memory. The runtime monitor will then search the database for the current foreground application. If the application exists in the database and is classified, the **power manager** will make use of the application's learned classification and user interaction pattern statistics (e.g., mean and standard deviation of touch events) to invoke a user-specific power management strategy that is optimized to the application. The power manager directly accesses the mobile device's hardware components to change screen backlight and CPU frequency. If the application does not exist in the database, or the application exists in the database but is not classified, *AURA* will make use of the user interaction events ("Global Touch" and "Global Key" events) to dynamically classify the application using a **Bayesian application classifier**. Following classification, the power manager will run an appropriate power management strategy on the next invocation of the app. An **event emulator** was also integrated into *AURA* to simulate user interaction events for validation purposes.

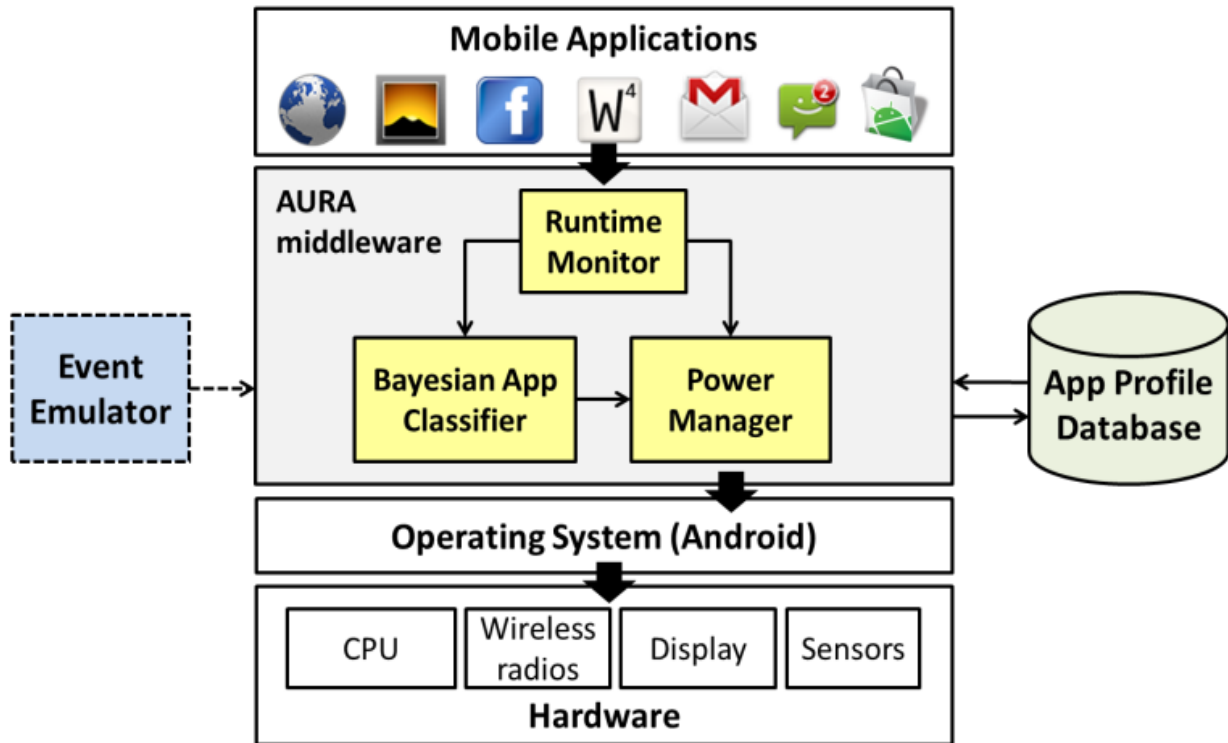


Figure 6.4: AURA energy optimization middleware framework

## 6.4 Runtime Monitor

The runtime monitor is an event-driven module responsible for monitoring user interactions with the mobile device. The Android OS makes use of public callback methods, known as event listeners, which allow applications to receive and handle user interactions (e.g., touch and key events) with items displayed on the user interface (UI) [53]. Unfortunately, there is no API to allow applications to receive “global” UI events, which are often consumed by application code or the current foreground application. In order for the runtime monitor to receive global UI events, a set of custom event broadcasts were created that are sent when any touch or key event occurs, when an application gains focus, and when the phone configuration changes (e.g., hard keyboard is opened or closed). The runtime monitor is only invoked whenever such custom global UI events are broadcast. The module keeps tracking variables for each current foreground application, the class of user interaction events, and the phone state, which are updated on

receiving global UI events. When a new application gains foreground focus, the runtime monitor will save session information from the previous foreground application (e.g. classification, user interaction stats) to the app profile database and retrieve any stored application information from the database for the new foreground application. If the new foreground application is classified, then the runtime monitor will invoke the power manager, otherwise, it will invoke the Bayesian classifier to dynamically classify the application as it receives user interaction events.

### 6.5 Bayesian Application Classifier

*AURA* uses the form of Bayesian learning described in Section 5.2 of this thesis. For this strategy, the input features are the mean ( $x_1$ ) and standard deviation ( $x_2$ ) of the times between user interaction (e.g., touch) events. A 7-part categorization of input features into *very-low*, *low*, *low-medium*, *medium*, *medium-high*, *high* and *very-high* classes based on corresponding values of mean and standard deviation is used. Such a fine-grained classification allows categorization of applications so that the power management algorithms (explained in the following section) can be adjusted to each application more effectively. A single output measure is defined: the resulting application classification ( $y$ ). The pre-defined application classes are *very-low-interaction*, *low-interaction*, *low-medium-interaction*, *medium-interaction*, *medium-high-interaction*, *high-interaction*, or *very-high-interaction*.

Consider an example of dynamic classification based on the training set shown in Table 6.2. When looking at this table, it should be noted that  $x_1=high$  denotes a *high-interaction* mean value, thus a smaller mean gap between events. Also, for simplicity, in this example we only consider three application classes: *low-interaction*, *medium-interaction*, and *high-interaction*. Let the bottom row be a new input feature ( $x_1=med$ ,  $x_2=high$ ) which was not in the training set,

that is presented to the classifier. If  $\lambda = 1$ , this new input is classified as follows. For the hypothesis  $y=high$ -interaction, the posterior probability is:

$$Prob(x_1 = med, x_2 = high | y = high) = \frac{1}{6} \times \frac{1}{6} \times \frac{1}{5} = \frac{1}{180} \quad (6.1)$$

because

$$Prob(x_1 = med | y = high) = \frac{1}{6} \quad (6.2)$$

and

$$Prob(x_2 = high | y = high) = \frac{1}{6} \quad (6.3)$$

and

$$Prob(y = high) = \frac{1}{5} \quad (6.4)$$

Similarly, for the hypothesis  $y=med$ -interaction, posterior probability is:

$$Prob(x_1 = med, x_2 = high | y = med) = \frac{3}{8} \times \frac{1}{4} \times \frac{3}{5} = \frac{9}{160} \quad (6.5)$$

and for hypothesis  $y=low$ -interaction, posterior probability is:

$$Prob(x_1 = med, x_2 = high | y = low) = \frac{1}{6} \times \frac{1}{3} \times \frac{1}{5} = \frac{1}{90} \quad (6.5)$$

The output measure of the new input feature is thus *med*-interaction. An application is classified in this manner on each interaction event. Upon being assigned the same classification  $m$  (a user-defined parameter) number of times, the application is marked as classified, and an appropriate power management strategy will be run on its next invocation.



## 6.6 Power Manager

As previously stated, the power manager makes use of each applications learned classification and user interaction pattern statistics to run its energy management algorithms. Using these algorithms, the power manager directly controls screen backlight levels and CPU frequency, scaling them intelligently to balance energy efficiency and performance. Four different algorithms were developed for use within the power manager – three MDP based algorithms and one Q-Learning based algorithm.

### 6.6.1 Markov Decision Process based Manager

As part of the *AURA* framework, three MDP based power management algorithms were developed to transition between a nominal state and below nominal state at discrete time intervals based on probabilities of user interaction, processor demands, and application classification. Markov Decision Processes are explained in Section 5.1. All three of the MDP based algorithms make use of the state flow diagram illustrated in Figure 6.5. When an application is classified and gains foreground focus, the classification and learned user event statistics are used to set algorithm control parameters (e.g., evaluation interval  $\tau$ , screen backlight adjustment levels  $\Delta_S$ , probability thresholds  $P_T$ ) and speculate on the probability of a user event within the next evaluation interval. When the probability of an event is below a given high-to-low threshold ( $P_{HLT}$  in Figure 6.5) and processor utilization is below a given utilization threshold ( $U_{HLT}$  in Figure 6.5), the state transitions to *Below Nominal* and the processor frequency level is adjusted to the lowest supported frequency. Note that the frequency levels shown in the figure are the values used for the Google Nexus One device. While the state is *Below Nominal*, the screen brightness is gradually adjusted down at each evaluation interval, taking advantage of the concept of *change blindness*. A transition to the mid-level frequency while in the *Below Nominal*

state will also occur if processor utilization reaches a low-med utilization threshold ( $U_{LMT}$  in Figure 6.5) or an immediate transition back to *Nominal* state will occur if processor utilization exceeds an even higher utilization threshold ( $U_{HLT}$  in Figure 6.5). If processor utilization is not a dominating factor then a transition back to *Nominal* will occur as the probability of a user event approaches a given low-to-high threshold ( $P_{LHT}$  in Figure 6.5) or any user event occurs, in which the nominal processor frequency and screen backlight level are set back to user-specified defaults.

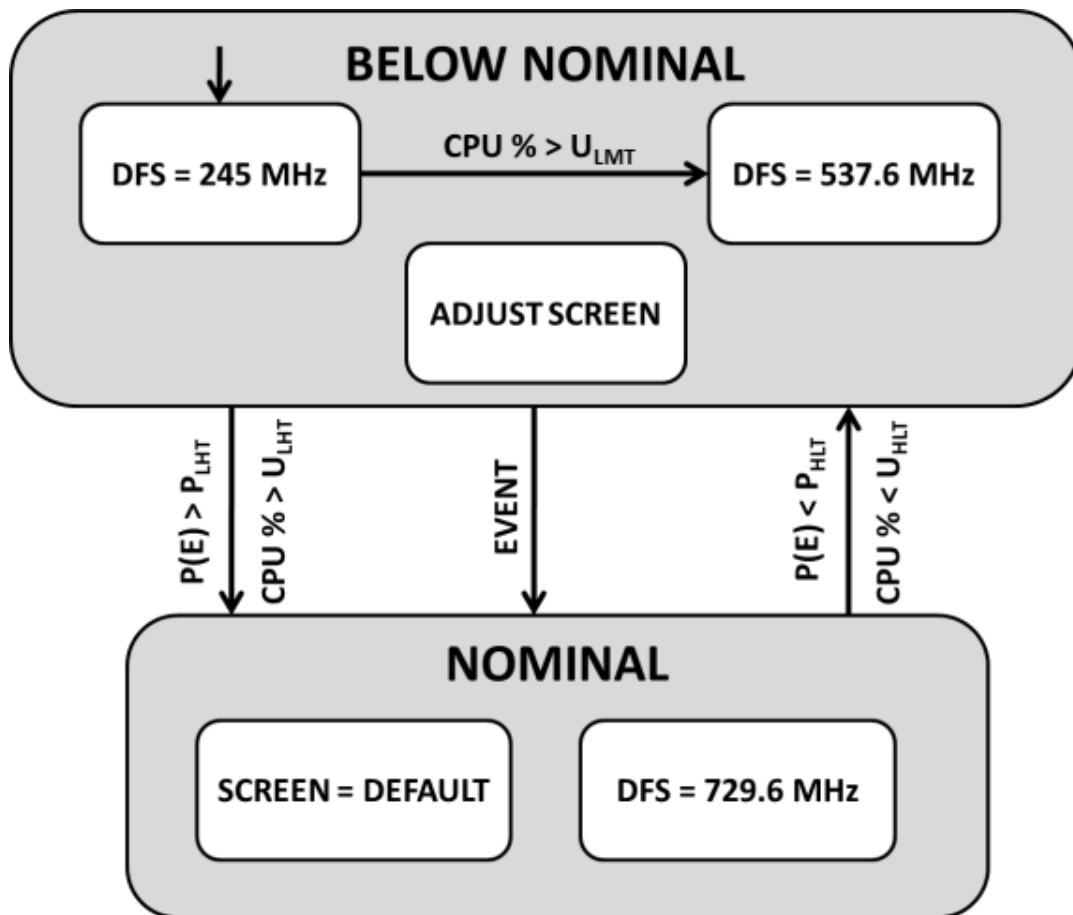


Figure 6.5: Control algorithms state flow diagram

The baseline MDP algorithm, *NORM*, makes use of a Gaussian distribution to predict user events based on learned classification and stored user-interaction statistics. Two derivatives of *NORM*

were created to dynamically adapt to real-time user-interaction during each classified application invocation. The *E-ADAPT* variant is event-driven and uses the most recent  $W_E$  user events to predict future interaction events. The *T-ADAPT* variant makes use of a moving average window of size  $W_T$ , to dynamically track recent user interaction within a temporal context.

Figure 6.6 illustrates the basic working of the MDP based power management algorithms. At Point 1, the probability of a user event reaches the low-to-high probability threshold and the state transitions back to *Nominal*. After an event occurs at Point 2, the probability of a user event falls below the high-to-low threshold, Point 3, and the state transitions to the *Below Nominal* state. While in the below nominal state, the screen backlight gradually changes during this predicted user idle time. At Point 4, the probability of a user event rises above the low-to-med threshold and the processor frequency is adjusted accordingly. At Point 5, the probability of a user-event reaches the low-to-high threshold and the state transitions back to *Nominal*. Point 6 shows a misprediction, in which a user event occurs while in the *Below Nominal* state. This causes the state to transition back to *Nominal* again. Mispredictions are used as a coarse QoS metric to evaluate the effectiveness of the algorithms and ensure energy savings are not diminishing overall QoS.

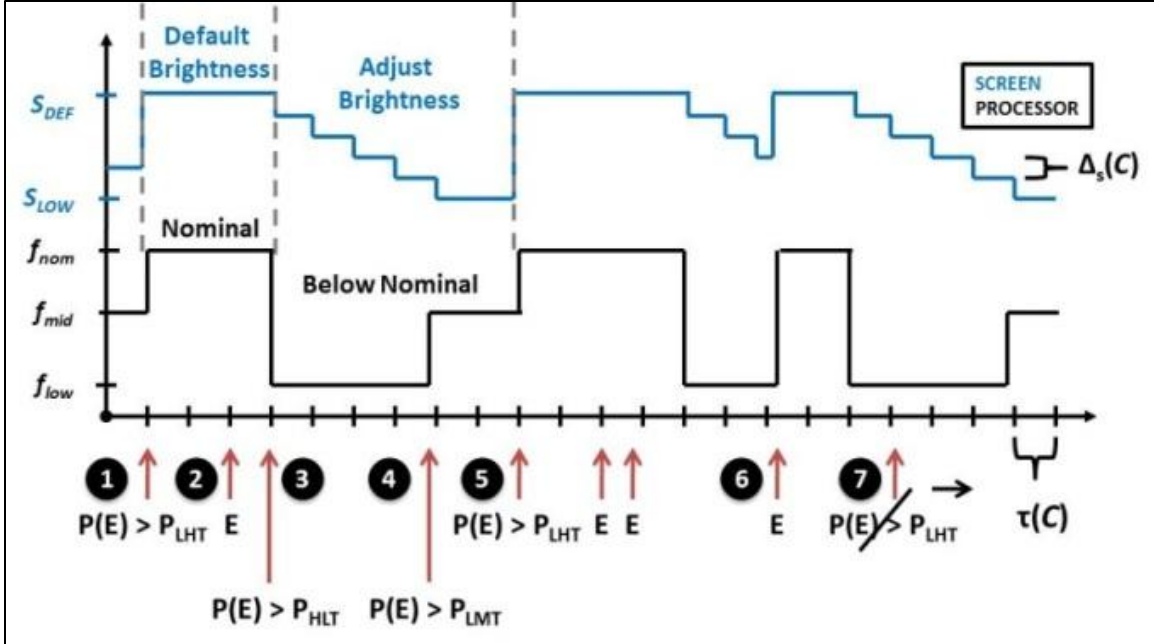


Figure 6.6: Control algorithms timeline illustration

### 6.6.2 Q-Learning based Manager

A Q-Learning based power management algorithm was also developed to include in the *AURA* framework. The Q-Learning technique is explained in detail in Section 5.3. In the *AURA* power manager, the Q-Learning algorithm **state** is made up of two parts – (1) whether the phone is *Nominal* (nominal processor frequency and default screen brightness) or *Below Nominal* (processor frequency is set to the lowest level and the screen brightness is gradually adjusted down), and (2) the evaluation interval number. The evaluation intervals in the *AURA* power manager are discrete time intervals in which to re-evaluate the system state and decide appropriate actions, and the evaluation interval number keeps track of how many evaluation intervals have passed since the last touch event. It can be one of 11 values: 1-10 or >10. The two possible **actions** are whether the state should change or not. The **reinforcements** for each state action pair are shown in Table 6.3. The values for the rewards bias the algorithm towards QoS -

higher rewards are given for when touch events are predicted correctly, and higher penalties are given when they are predicted incorrectly. Each application has its own table of Q values, which is stored in the app profile database when the app loses foreground focus.

**Table 6.2: Q-Learning algorithm reinforcements**

<b>State</b>	<b>Touch Received?</b>	<b>Action</b>	<b>Reinforcement</b>
<i>Nominal</i>	yes	Do NOT change to <i>Below Nominal</i>	2
<i>Nominal</i>	no	Do NOT change to <i>Below Nominal</i>	-1
<i>Nominal</i>	yes	Change to <i>Below Nominal</i>	-2
<i>Nominal</i>	no	Change to <i>Below Nominal</i>	1
<i>Below Nominal</i>	yes	Do NOT change to <i>Nominal</i>	-2
<i>Below Nominal</i>	no	Do NOT change to <i>Nominal</i>	1
<i>Below Nominal</i>	yes	Change to <i>Nominal</i>	2
<i>Below Nominal</i>	no	Change to <i>Nominal</i>	-1

### 6.6.3 Power Manager Control Parameters

Table 6.4 lists several key algorithm control parameters that can be used to customize the algorithms to each individual user and application. For example, the evaluation interval is determined based on the application classification. High interaction applications will require more frequent evaluation, while low interaction applications can use a larger evaluation interval. Event probability thresholds are also set based on application classification and coefficient of variance, which gives a measure of the variability of the learned user interaction pattern. For example, low interaction applications with low variability will use high event probability thresholds to bias towards energy savings. For applications with high variability in user-interaction patterns, event probability thresholds will be slightly biased towards QoS. Processor

utilization thresholds are statically set at design time; however, could also be set dynamically if finer-grained control is required. Screen backlight levels are set based on application classification, where high interaction applications are again biased towards QoS and low interaction applications are biased towards energy savings. It is important to note that screen backlight levels are also limited by a QoS-driven lower bound of 40%.

For the *E-ADAPT* variant, a larger  $W_E$  value can be used to get a global view of the user-interaction pattern or a smaller  $W_E$  value can be used for a more local view of the user-interaction pattern. A similar argument can be made for the *T-ADAPT* variant which dynamically adapts the user-event statistics based on a moving average window of size  $W_T$

**Table 6.3: Algorithm control parameters**

Parameter	Dependencies	Description
Evaluation Interval ( $\tau$ )	Application Classification	Discrete time interval in which to re-evaluate the system state and decide appropriate action(s)
Event Probability Threshold ( $P_T$ )	Application Classification; Not used in Q-LEARNING	Threshold to determine state transitions based on user-event prediction
Processor Utilization Threshold ( $U_T$ )	Static; Not used in Q-LEARNING	Conditional threshold to determine state transitions based on CPU demands
Backlight Adjustment Levels ( $\Delta_S$ )	COV <sup>†</sup>	Screen backlight adjustment levels while in state <i>Below Nominal</i>
Event Window ( $W_E$ )	E-ADAPT Only	Number of most recent user events to use for dynamic adaptation
Time Window ( $W_T$ )	T-ADAPT Only	Size of moving average window to use for dynamic adaptation
Epsilon ( $\epsilon$ )	Q-LEARNING Only	Probability that an action will be chosen randomly, as opposed to choosing the best action
Learning Rate ( $\alpha$ )	Q-LEARNING Only	Determines to what extend old Q values are overridden
Discount Factor ( $\gamma$ )	Q-LEARNING Only	Determines the importance of future rewards

<sup>†</sup>COV  $\equiv$  Coefficient of Variance

## 6.7 Device Power Modeling

### 6.7.1 Overview

In order to quantify the energy-effectiveness of the proposed framework, power analysis was performed on two Android based smartphones: HTC Dream and Google Nexus One, with the goal of creating power models for the CPU and backlight. We use a variant of Android OS 2.2 (Froyo), on the HTC Dream and a variant of Android OS 2.3.3 (Gingerbread) on the Google Nexus One. We use Android SDK Revision 11 on both devices. The Google Android platform is comprised of a slightly modified 2.6.35.9 Linux kernel, and incorporates the Dalvik Virtual Machine (VM), a variant of Java implemented by Google. All user-space applications are Dalvik executables that run in an instance of the Dalvik VM.

The power estimation models were built using real power measurements on the HTC Dream and Google Nexus One devices. The contact between the smartphone and the battery was instrumented, and current was measured using the Monsoon Solutions power monitor [55]. The monitor connects to a PC running the Monsoon Solutions power tool software that allows real-time current/power measurements over time. The power monitor setup is shown in Figure 6.7. A custom test app was developed that was capable of switching dynamic frequency scaling (DFS) levels and screen backlight levels over time. The corresponding power traces from the Monsoon Power Tool were then used to obtain power measurements for each DFS frequency and screen brightness level.

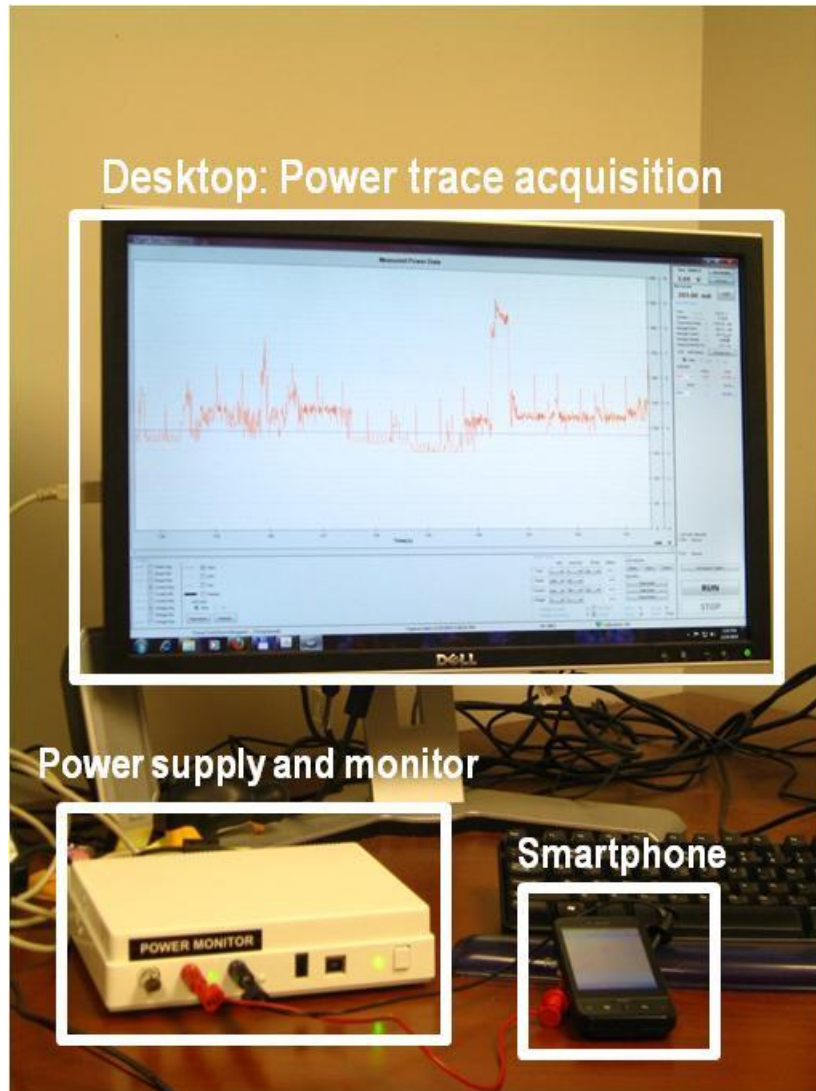


Figure 6.7 Power monitor setup

### 6.7.2 CPU DFS Power Model

Dynamic frequency scaling (DFS) is a standard technique in computing systems to conserve power by dynamically changing clock frequency of a digital component (e.g., processor) at runtime when the system is idle or under-utilized. Linux-based systems, such as Android, provide a DFS kernel subsystem (*cpufreq*) that is a generic framework for managing the processor operation. The *cpufreq* subsystem supports a variety of DFS drivers, many of which



are configurable [56]. The *userspace* driver, which is used in this work, allows a user-space program to manually control DFS levels based on a user-defined set of rules.

The HTC Dream smartphone consists of a Qualcomm® MSM7201A processor [58]. This processor is nominally clocked at 528 MHz; however, it supports 11 different frequency levels ranging from 128 MHz up to 614 MHz. The Google Nexus One consists of a Qualcomm QSD8250 Snapdragon processor [59]. It supports 21 frequency levels ranging from 245 MHz to 1.1 GHz, and is nominally clocked at 729.6 MHz. In order to get power models for the DFS levels, the test app was used to manually switch frequency domains. Given the difficulty of controlling CPU utilization explicitly, an intensive computation loop, followed by a wait function was used to toggle processor utilization between 0 and 100% for each frequency level, and the power consumption was recorded respectively. Based on the observed measurements, linearity was assumed for CPU utilization that lies between 0 and 100 % for each frequency. Linear regression analysis [57] was used to determine appropriate coefficients and develop our CPU power estimation models with units of *mW/% utilization*. The DFS power models for the two devices are shown in Figures 6.8 and 6.9.

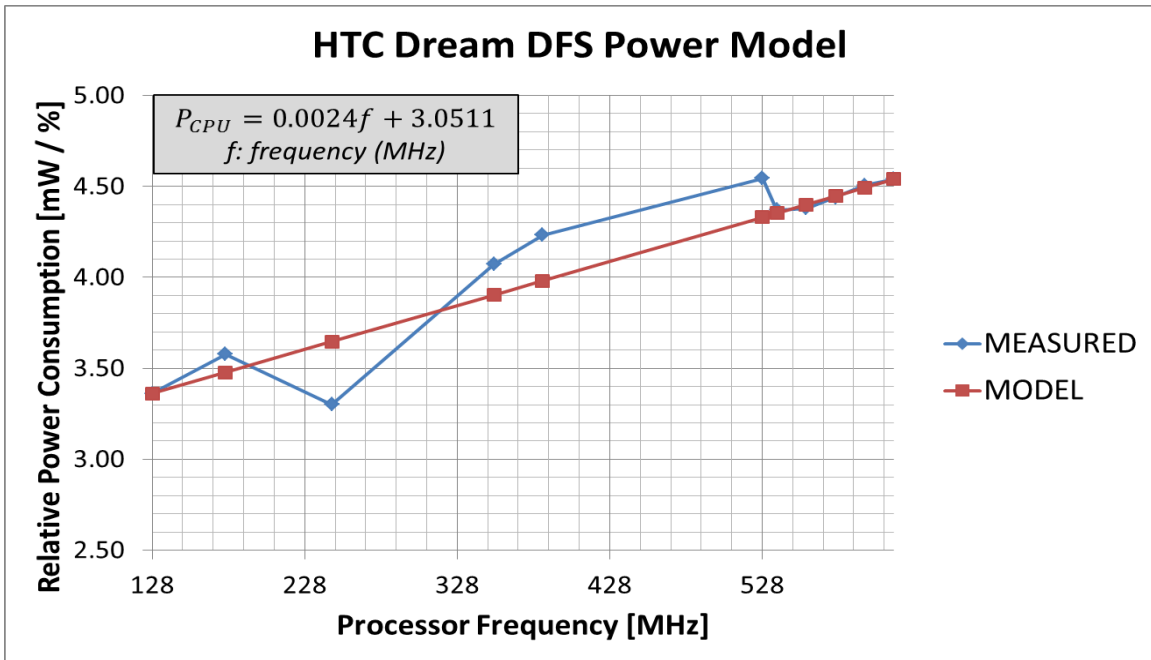


Figure 6.8: DFS-based CPU power model for HTC Dream

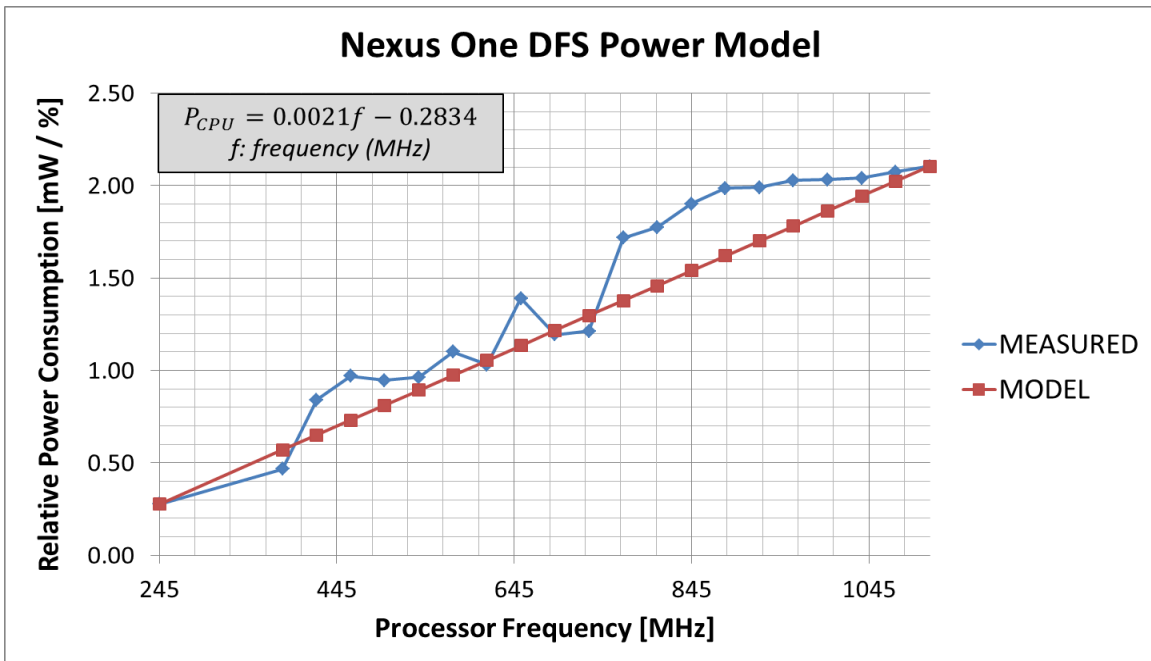


Figure 6.9: DFS-based CPU power model for Google Nexus One

### 6.7.3 Screen Backlight Power Model

Screen power consumption was also measured using the Monsoon Power Tool with the help of our custom test app which allowed static and dynamic control of screen backlight levels. Screen backlight levels were incremented in steps of 10% and relative average power measurements were recorded with the Monsoon Power Tool. Figure 6.10 shows the instantaneous power measured using the tool as the backlight level is ramped up. Linear regression analysis was again used to obtain relative power models with units of  $mW$ . The screen backlight power models for the two devices are shown in Figures 6.11 and 6.12.

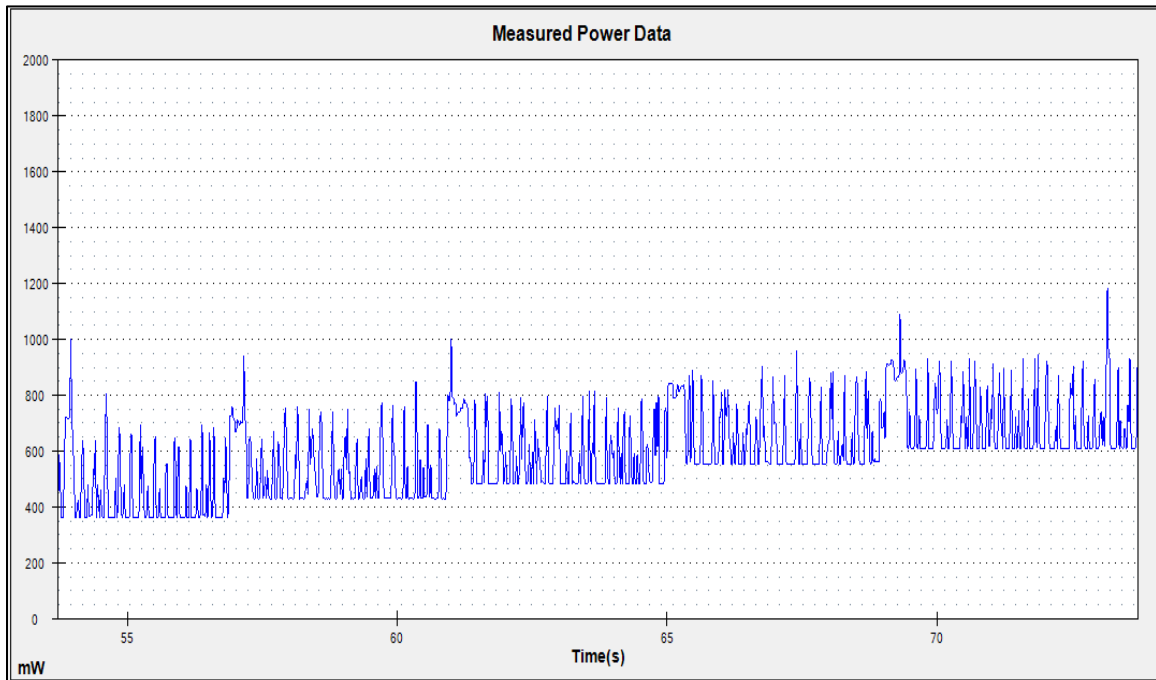


Figure 6.10: Monsoon power monitor screen power

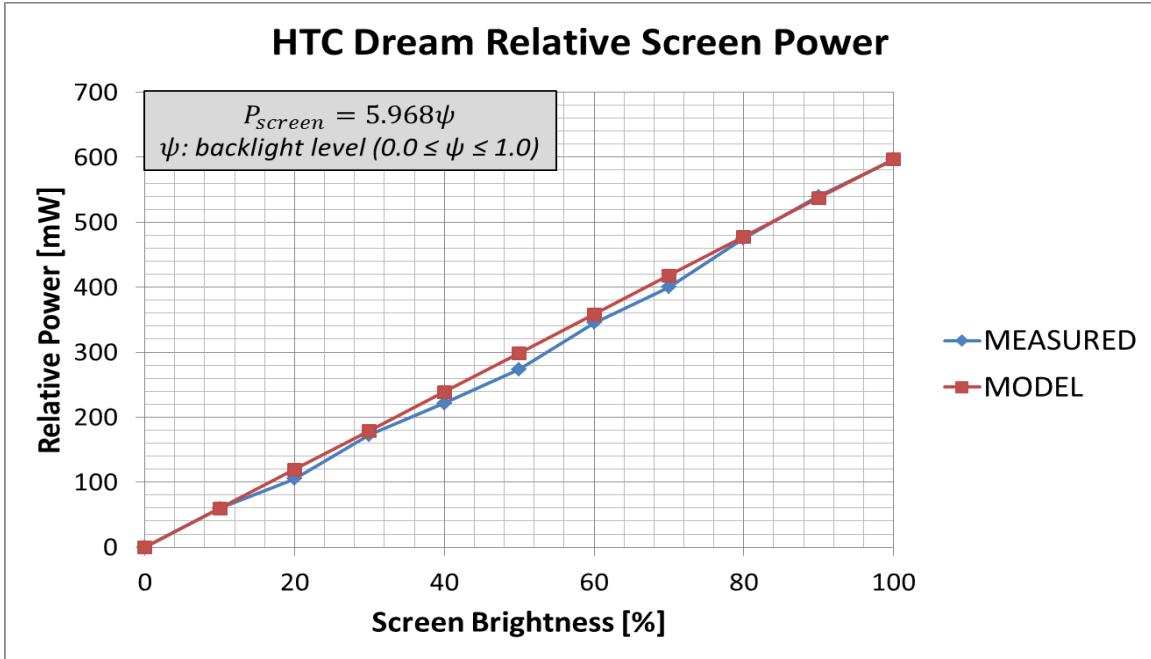


Figure 6.11: Screen backlight power model for HTC Dream

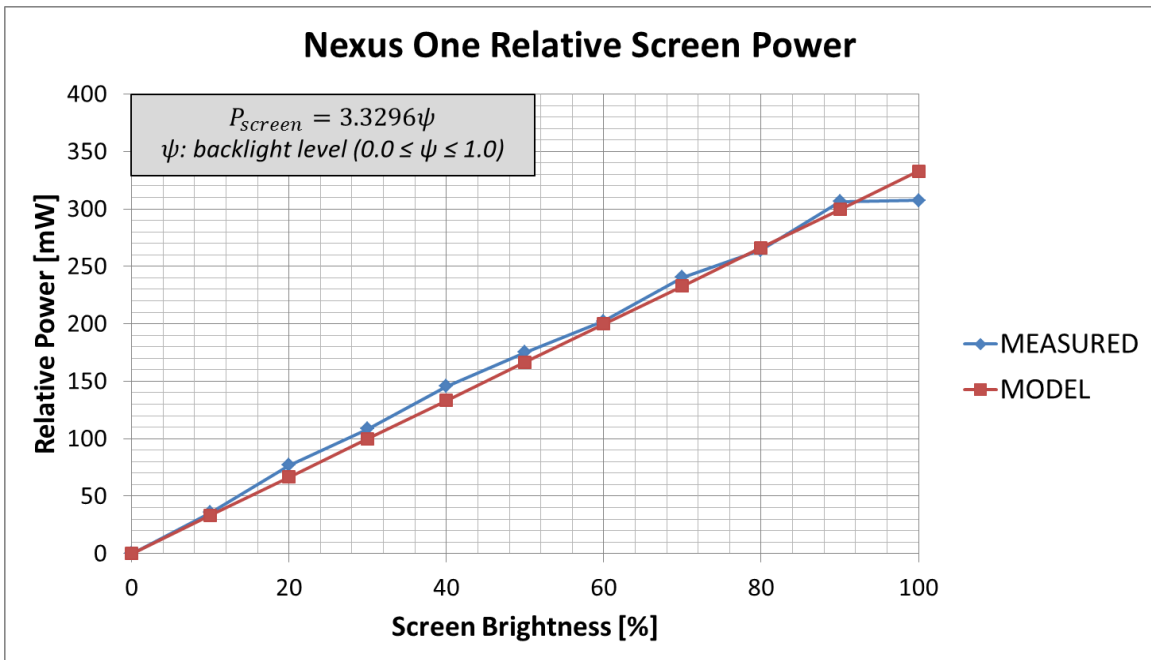


Figure 6.12: Screen backlight power model for Google Nexus One

## 6.8 Experimental Methodology

The *AURA* framework was implemented on real HTC Dream and Google Nexus One smartphones. The CPU and backlight power models described above were integrated into the *AURA* framework, to guide the MDP and Q-Learning based power management strategies and also to quantify any resulting energy savings. To test the effectiveness of the *AURA* framework, stimuli were obtained from two sources:

- **Event emulator:** This module emulates various key and touch based user interaction event patterns, including: (i) *Constant* - sends events at a user-defined constant rate; (ii) *Stochastic* - sends events using a Gaussian distribution probability with a user-defined mean and standard deviation; (iii) *Random* - sends events using a pseudo-random uniform distribution; and (iv) *High Burst Long Pause (HBLP)* - sends a quick burst of 20 events, followed by a long 10 second pause.
- **Real users:** Five users were involved who interacted with applications on their Android smartphones as they would normally, generating user-specific interaction events in the process.

The algorithm control parameter values assumed in the *AURA* implementation are shown in Table 6.5.

**Table 6.4: Algorithm control parameter values**

Module	Parameter	Value(s)
Bayesian Classifier	Smoothing Constant ( $\lambda$ )	1
	Training Set Size ( $\gamma_{TS}$ )	50
All Algorithms	Evaluation Interval ( $\tau$ )	1000/850/700/550/400/250/100 (VH/H/MH/M/LM/L Classification <sup>†</sup> )
MDP Algorithms	High-to-Low Event Probability Threshold ( $P_{HLT}$ )	0.55/0.5/0.45/0.4/0.35/0.3/0.25 (VH/H/MH/M/LM/L Classification <sup>†</sup> )
	Low-to-High Event Probability Threshold ( $P_{LHT}$ )	0.75/0.7/0.65/0.6/0.55/0.55/0.45 (VH/H/MH/M/LM/L Classification <sup>†</sup> )
	High-to-Low Processor Utilization Threshold ( $U_{HLT}$ )	0.5
	Low-to-Med Processor Utilization Threshold ( $U_{LMT}$ )	0.6
	Low-to-High Processor Utilization Threshold ( $U_{LHT}$ )	0.8
	Backlight Adjustment Levels ( $\Delta_S$ )	0.05/0.1/0.15 (H/M/L COV)
	Event Window Size ( $W_E$ )	6
	Time Window Size ( $W_T$ )	6
Q-Learning Algorithm	Epsilon ( $\epsilon$ )	0.1
	Learning Rate ( $\alpha$ )	1
	Discount Factor ( $\gamma$ )	0.1

<sup>†</sup>VH  $\equiv$  very-high-interaction, H  $\equiv$  high-interaction, MH  $\equiv$  medium-high-interaction, M  $\equiv$  medium-interaction, LM  $\equiv$  low-medium-interaction, L  $\equiv$  low-interaction, VL  $\equiv$  very-low-interaction

## 6.9 Experimental Results

### 6.9.1 Event Emulation Results

In the first study, the event emulator was used to imitate four different user-interaction patterns, to contrast the effectiveness of the four power management algorithms. The primarily interest

was in the energy savings from our algorithms compared to nominal frequency and default screen backlight settings on the HTC Dream and Google Nexus One devices. To evaluate the level of QoS being offered, a *successful prediction rate* metric was defined. The successful prediction rate metric keeps track of the number of events that occurred while in *Below Nominal* state. A low successful prediction rate is indicative of diminished QoS. QoS is subjective and difficult to quantify at a user level, and this is well known – two users may perceive a common event very differently. Therefore, the successful prediction rate is merely an attempt at measuring user satisfaction.

To discover how the prediction rate metric was correlated with actual user satisfaction, the control parameters of the *NORM* algorithm were modified to offer different successful prediction rates. Users were then asked to interact with eight common Android apps with the algorithm running and record their level of satisfaction on a scale of 1 to 5 according to the following criteria:

1. *Very Satisfied*: app functions normally and screen brightness changes are not intrusive
2. *Somewhat Satisfied*: very little lag noticeable; screen brightness changes noticeable but not overly distracting
3. *Mediocre*: app functions slower than usual; screen brightness is intrusive at times; app is still enjoyable to use
4. *Somewhat Dissatisfied*: lag and screen brightness changes very noticeable, but not to the point of ruining app functionality
5. *Very Dissatisfied*: app runs frustratingly slow; screen brightness changes make viewing

difficult; app is rendered useless

The *minimum acceptable prediction rate* for an application is defined to be the prediction rate that causes **noticeable degradation** in the performance of an app, or that offers a user satisfaction level of anything better than 3 on the above scale. Figure 6.13 shows the minimum acceptable prediction rates for different applications. Evidently, different applications can tolerate different minimum acceptable prediction rates. Thus, in an attempt to offer acceptable QoS across all applications, we define a *performance threshold* of 80%, which is a conservative average of the eight apps' minimum acceptable prediction rates. The goal is to tune our power management algorithms so that they never offer prediction rates below the performance threshold.

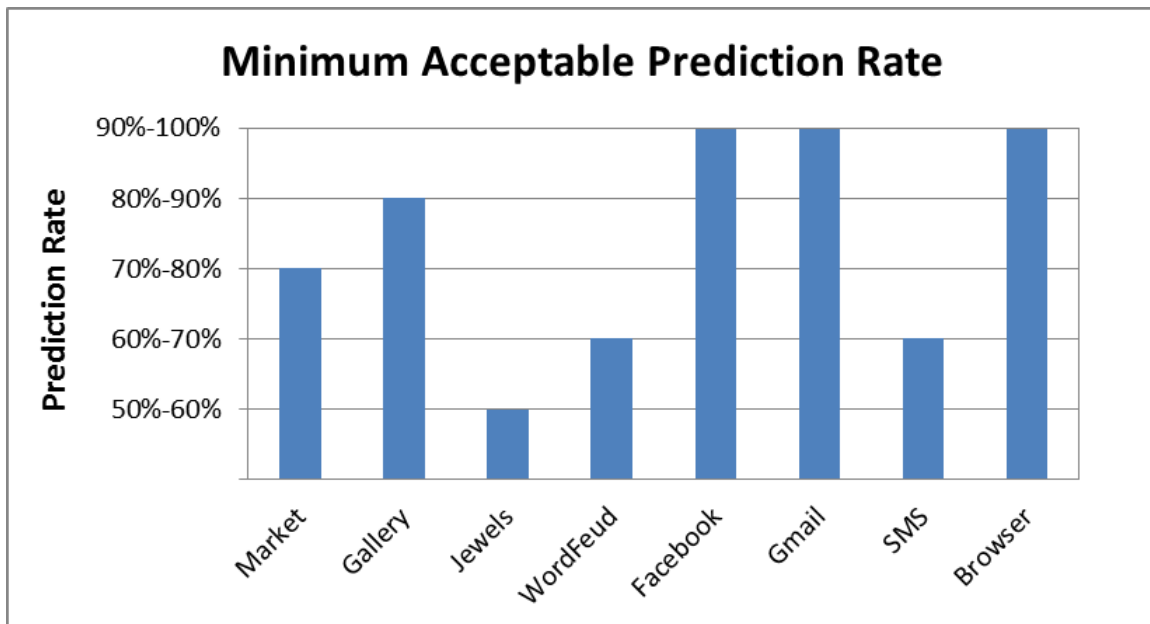


Figure 6.13: Minimum acceptable prediction rates for chosen applications

Figures 6.14 and 6.15 show the energy savings achieved by the four power management algorithms we proposed in this work, for the four emulated interaction patterns on the HTC Dream and Google Nexus One devices. It can be seen that *E-ADAPT* is capable of offering the



best energy savings out of the MDP algorithms (up to 20%) on the HTC Dream due to the responsive event-triggered nature of its dynamic adaptation. The algorithm only adapts on instances of user events and does not require constant adaptation during each evaluation interval. This also reveals why *E-ADAPT* performs poorly in conditions like *HBLP* - the quick events during the burst are stored in the *E-ADAPT* window as they occur and used to calculate the mean for prediction, but the long pause is not taken into account until the event *after* the long pause occurs. On the other hand, *T-ADAPT* offers lower energy savings due to its time-based nature. Whereas *E-ADAPT* must wait for an event occurrence in order to adjust its prediction, *T-ADAPT* adjusts on each evaluation interval. This form of adaptation is slightly more computationally intensive. Additionally, if the state is *Nominal*, it will not drop to *Below Nominal* under *T-ADAPT* unless CPU utilization is below  $U_{HLT}$  (even if the probability of an event is very low). Because *T-ADAPT* shifts its window and does computations on a regular interval instead of waiting for an event to occur, the CPU utilization is high more often, thus the state transitions to *Below Nominal* less often. *NORM* offers the best overall energy savings. *NORM* performs well in conditions like *HBLP* because it takes the average time between all touch events into account, giving the algorithm a more global view than *E-ADAPT* or *T-ADAPT*. When considering energy savings across all four emulated patterns, *Q-LEARNING* is a close second to *NORM*. However, one reason for this is that the algorithm was tested as if the emulated patterns were seen on the initial invocation of an application (i.e., good Q values have not been learned yet), Thus, the higher energy savings may have a negative impact on user QoS.

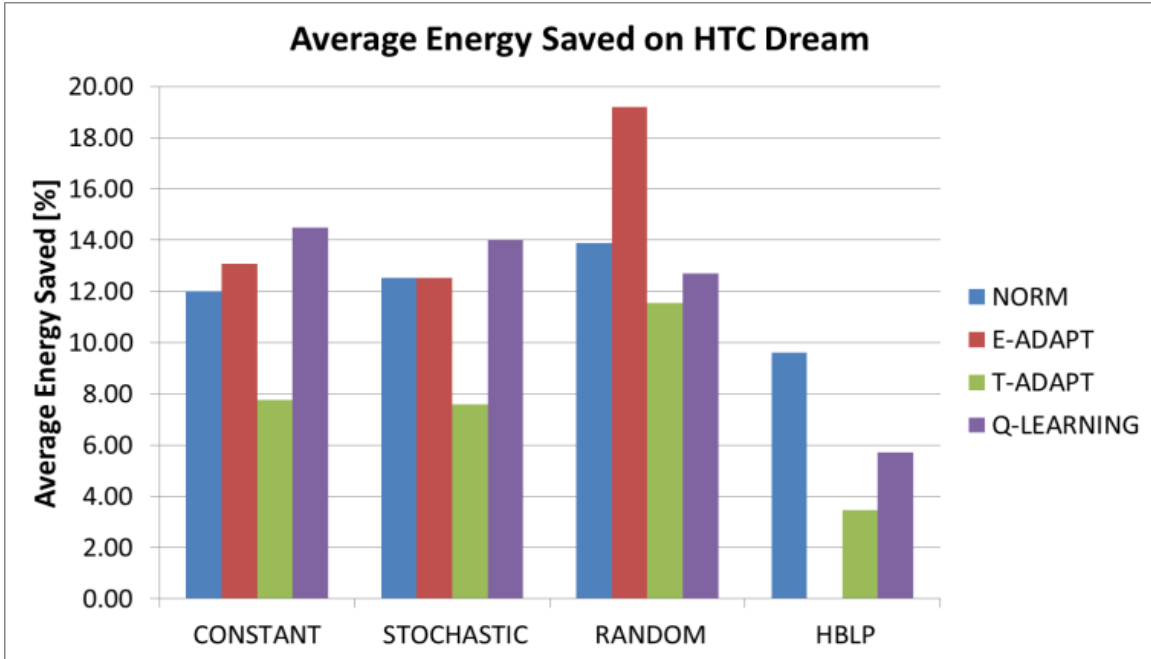


Figure 6.14: Average energy saved on HTC Dream for emulated patterns

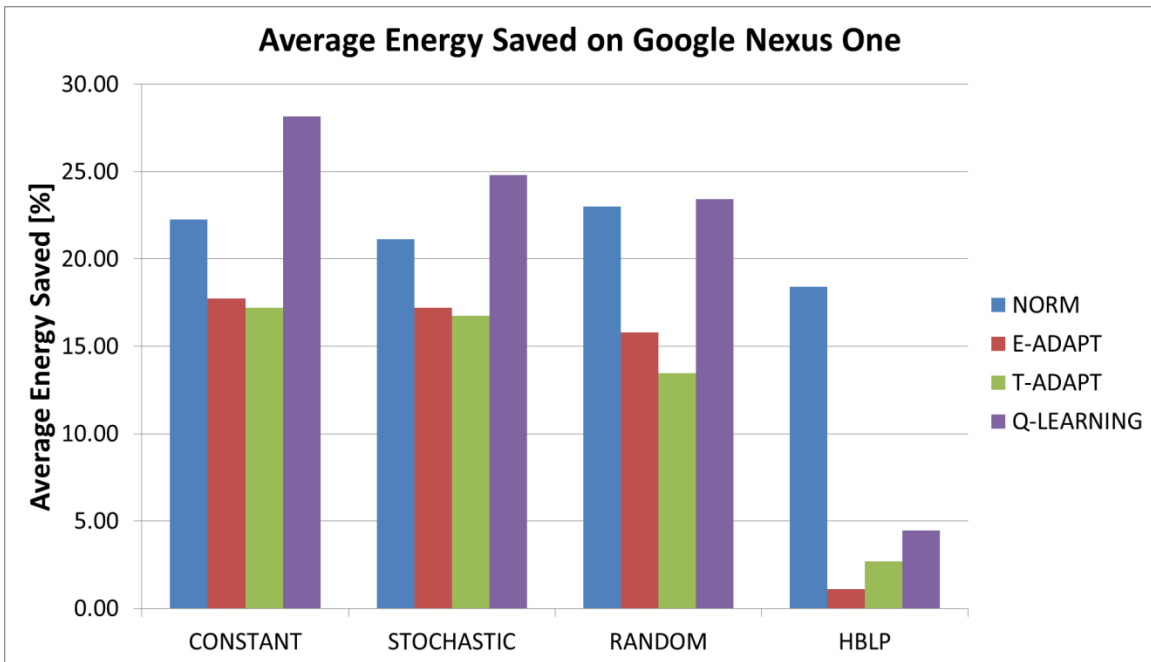
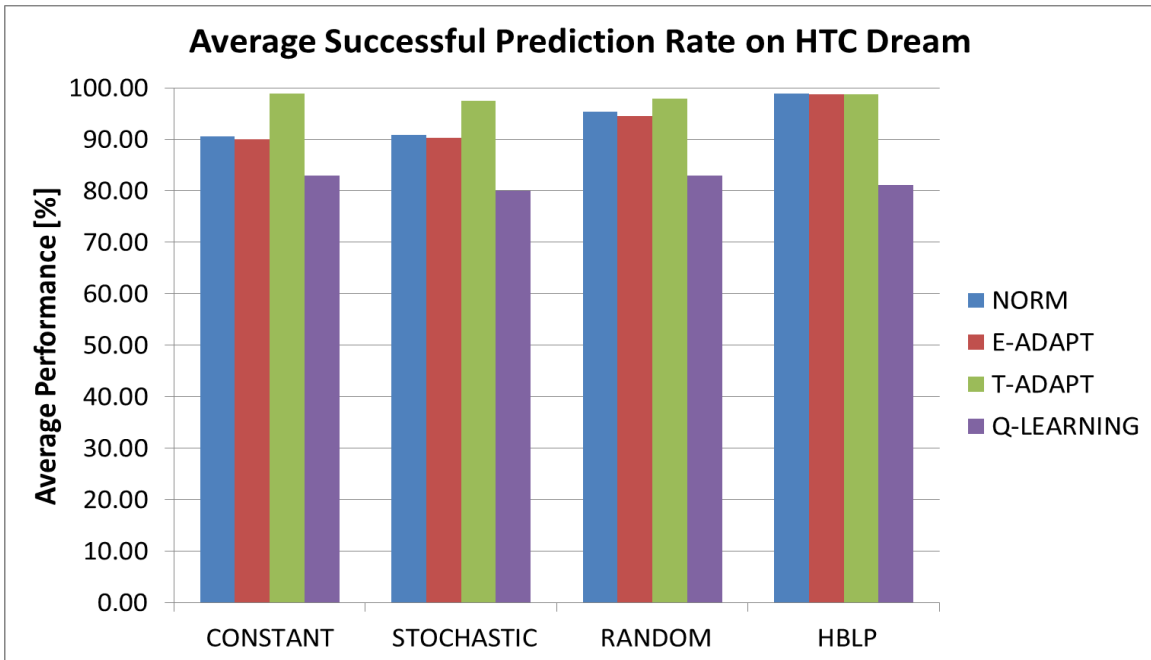
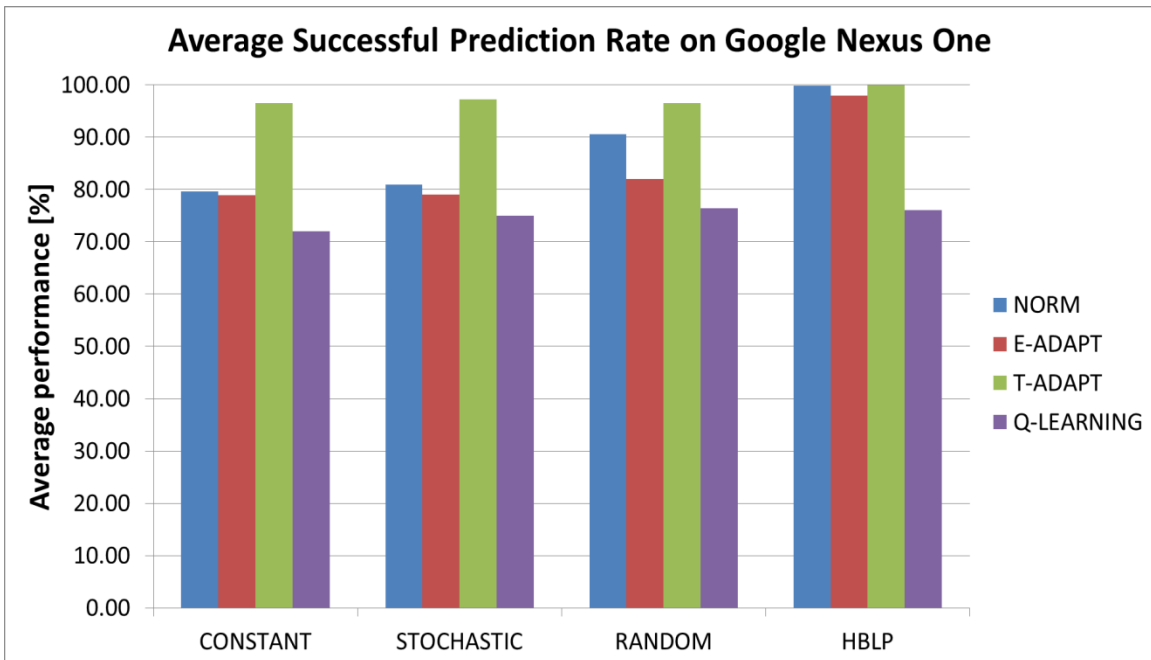


Figure 6.15: Average energy saved on Google Nexus One for emulated patterns

Figures 6.16 and 6.17 show average successful prediction rates of each algorithm for the event emulator patterns. *Q-LEARNING*'s successful prediction rate is the lowest of the four algorithms. This can be attributed to the fact that the algorithm has not had a chance to learn a good model for the emulated patterns. If the algorithm had more of a chance to learn good models for each pattern better prediction rates could have been achieved (possibly at the cost of lower energy savings). It is clear that all three MDP algorithms have high successful prediction rates, validating the robustness of our chosen classification and power management algorithms. Interestingly, a higher successful prediction rate does not necessarily imply higher energy savings for the following reason: an event may be predicted to occur much earlier than the actual time of its occurrence and the prediction will still be considered successful. For example, if the state is *Below Nominal* and the probability of an event occurrence exceeds  $P_{LHT}$ , the state will switch to *Nominal* and remain there until an event occurs and the probability of an event occurrence is low again. Therefore, if an event does not occur quickly, less energy will be saved because the *Nominal* state will be prolonged (but the prediction will still be considered successful). This is the reason why although *T-ADAPT* offers the best successful prediction rates due to its superior temporal locality characteristics, it offers lower energy savings than *NORM* and *E-ADAPT*. The fact is also applicable only to the MDP algorithms, and not to *Q-LEARNING* as it does not wait for an event to occur before switching to *Below Nominal*. It should be noted that in our experiments, the MDP algorithm control parameters were biased towards high successful prediction rates, thus maintaining a high minimum level of user QoS for all three algorithms. The control parameters could potentially be adjusted to accept lower successful prediction rates in order to offer even higher energy savings



**Figure 6.16:** Average successful prediction rate on HTC Dream for emulated patterns



**Figure 6.17:** Average successful prediction rate on Google Nexus One for emulated patterns

## 6.9.2 User Study Results

A second study was conducted, this time focusing on eight common Android apps and five real users using these apps on the HTC Dream and Google Nexus One devices. The users were asked to interact with the different applications over the course of several sessions during a day, with different energy saving algorithms enabled in each session. In addition to the MDP based and Q-Learning based power management algorithms running as part of the *AURA* middleware framework, the framework proposed by Shye et al. [8] was implemented, which also aims to improve CPU and backlight energy consumption for mobile devices by using change blindness to gradually ramp down both CPU frequency and screen backlight levels over time.

Figures 6.18 and 6.19 compare the average energy savings of the technique proposed by Shye et al. (*CHBL*) with the four *AURA* power management techniques for the eight different applications, across all users. It can be seen that *CHBL* offers fairly consistent (but low) energy savings across all applications. The consistency can be explained by noting that the change blindness optimization employed in *CHBL* is independent of user interaction patterns and application type, instead using constant time triggered scaling of the CPU frequency and backlight levels. In contrast, *AURA*'s user-aware and application-aware algorithms (specifically *E-ADAPT* and *Q-LEARNING*) offer higher energy savings because they can dynamically adapt to the user interaction patterns and take full advantage of user idle time. For instance, the *E-ADAPT* and *Q-LEARNING* algorithms save up to 4× and 5× more energy compared to *CHBL* for the *Gmail* and *WordFeud* applications on the HTC Dream, respectively. On the Nexus One, *E-ADAPT* and *Q-LEARNING* together save an average of approximately 2.5× more energy than *CHBL* across all applications. In some cases, for instance the high interaction *Jewels* application, interaction slack is too small to be exploited by our algorithms. While *CHBL* offers higher

energy savings for *Jewels* due to its lack of user-awareness and application-awareness during scaling, it comes at a cost: noticeable QoS degradation issues for the user.

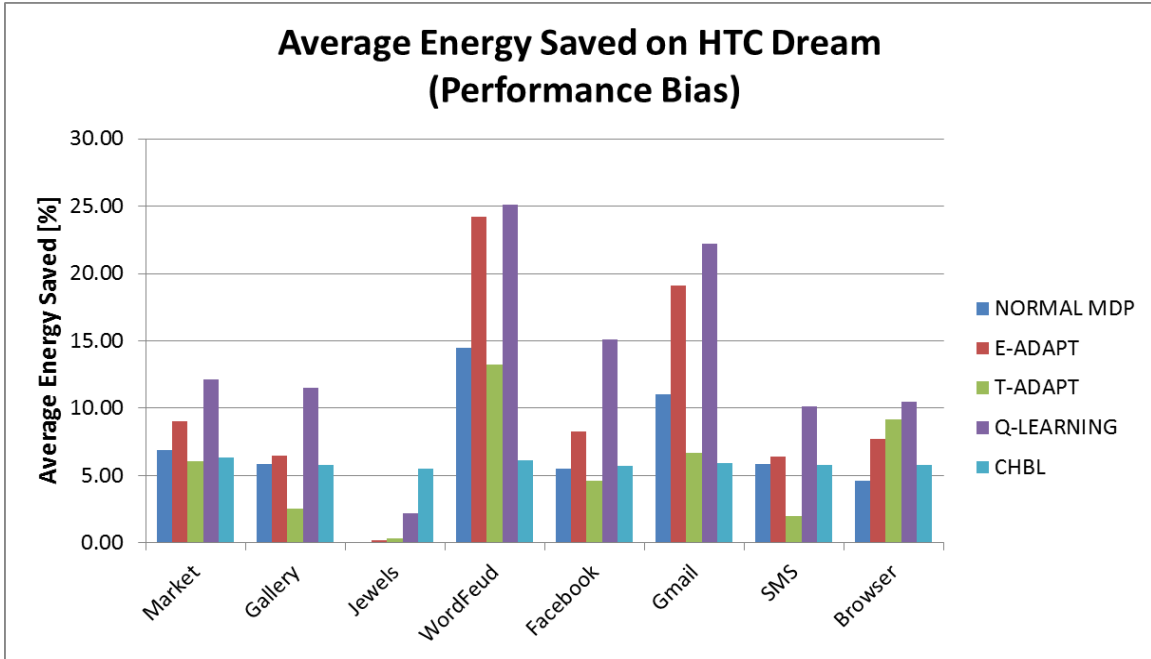


Figure 6.18: Average energy saved on HTC Dream for real users (performance bias)

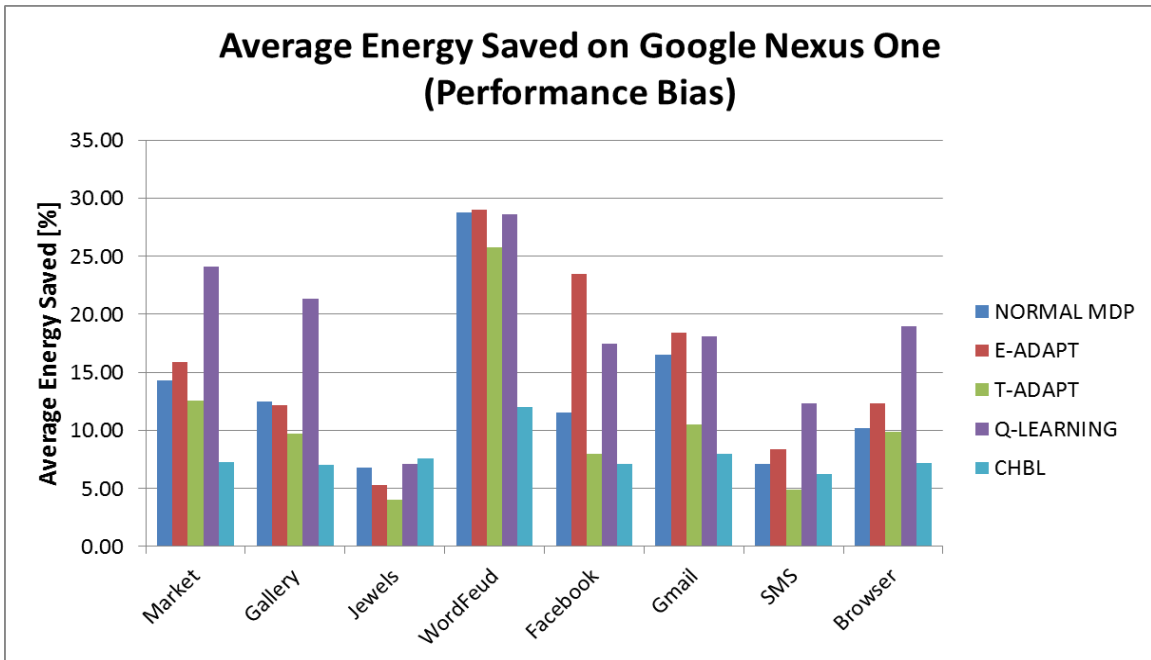


Figure 6.19: Average energy saved on Google Nexus One for real users (performance bias)

Figures 6.20 and 6.21 show the average successful prediction rates for the real user interaction patterns. Shye et al.’s algorithm, *CHBL*, was not included because it does not contain defined states or prediction mechanisms, making determining mispredictions impossible. The figures show high successful prediction rates similar to those that were seen in the event emulation results. The reason for this is that the algorithm control parameters used for the results in Figures 6.18-6.21 were biased towards maintaining fewer mispredictions.

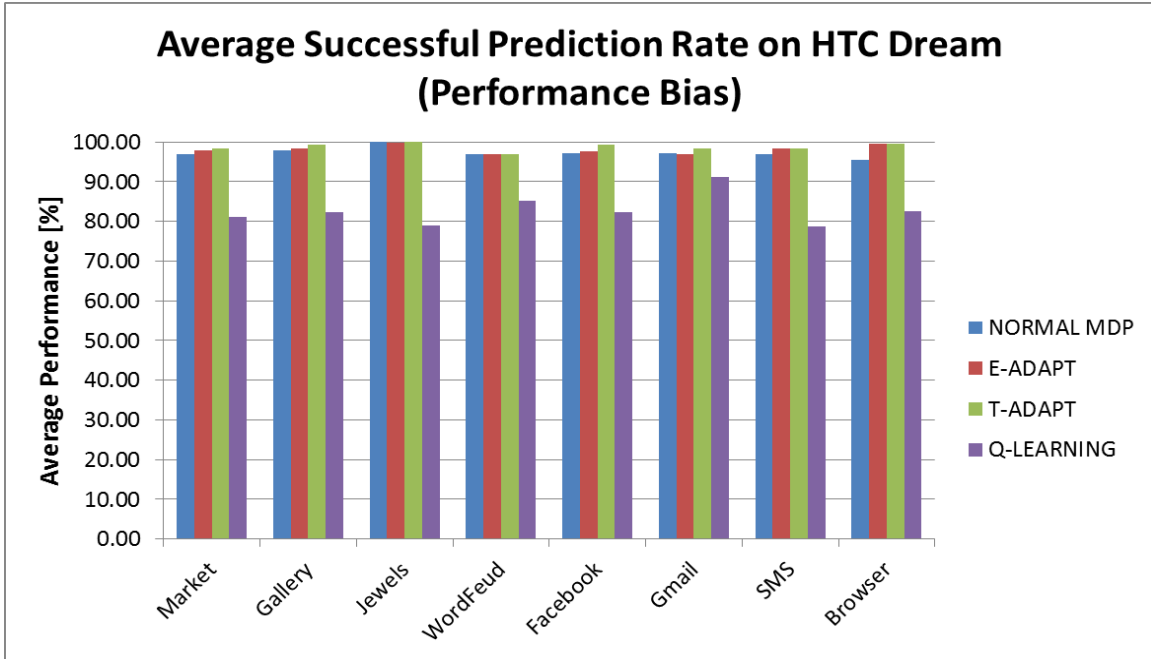


Figure 6.20: Average successful prediction rate on HTC Dream for real users (performance bias)

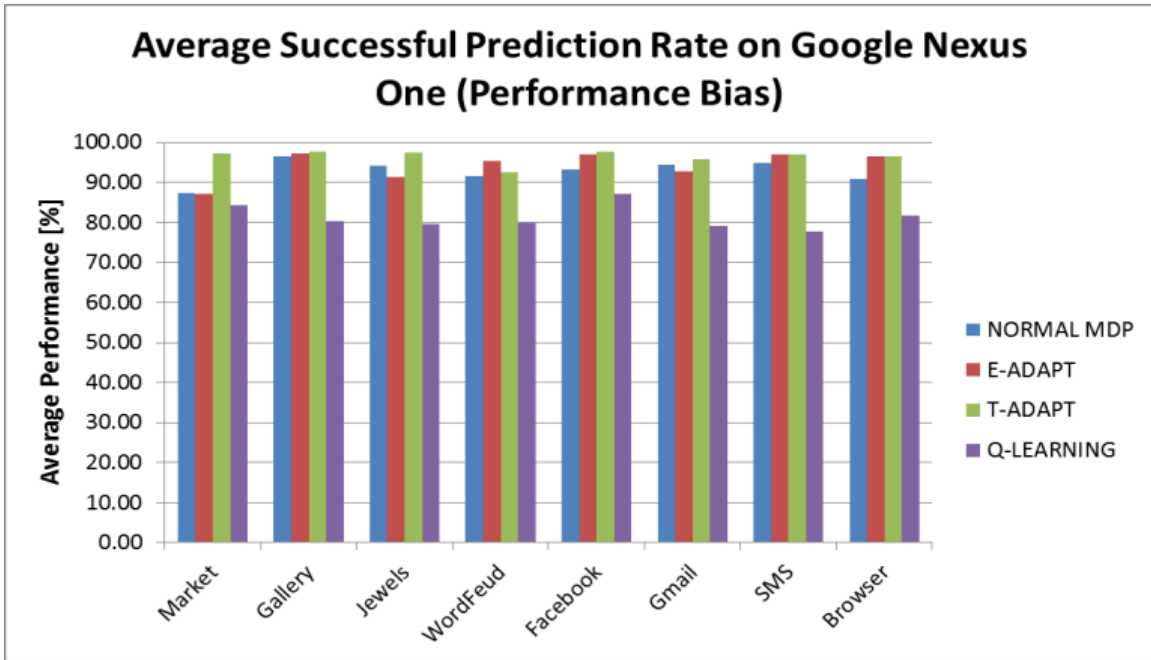


Figure 6.21: Average successful prediction rate on Google Nexus One for real users (performance bias)



For the next study, the algorithm control parameters for the MDP algorithms were adjusted to allow even higher energy savings to see how the number of mispredictions was affected. The event probability thresholds  $P_{HLT}$  and  $P_{LHT}$  were increased by 20%, and the processor utilization thresholds  $U_{HLT}$ ,  $U_{LMT}$ , and  $U_{LHT}$  were increased by 10% from their original values in Table 6.5. Figure 6.22 shows the energy savings of our four algorithms with the adjusted algorithm control parameters on the Google Nexus One compared with *CHBL*, which is unchanged. The energy savings are clearly higher with the adjusted parameters, with *E-ADAPT* and *Q-LEARNING* offering energy savings of up to 34% over default device settings. However, the successful prediction rates for the MDP algorithms in Figure 6.23 have dropped significantly. In fact, most of the prediction rates are below the acceptable performance threshold of 80%, making this configuration unacceptable.

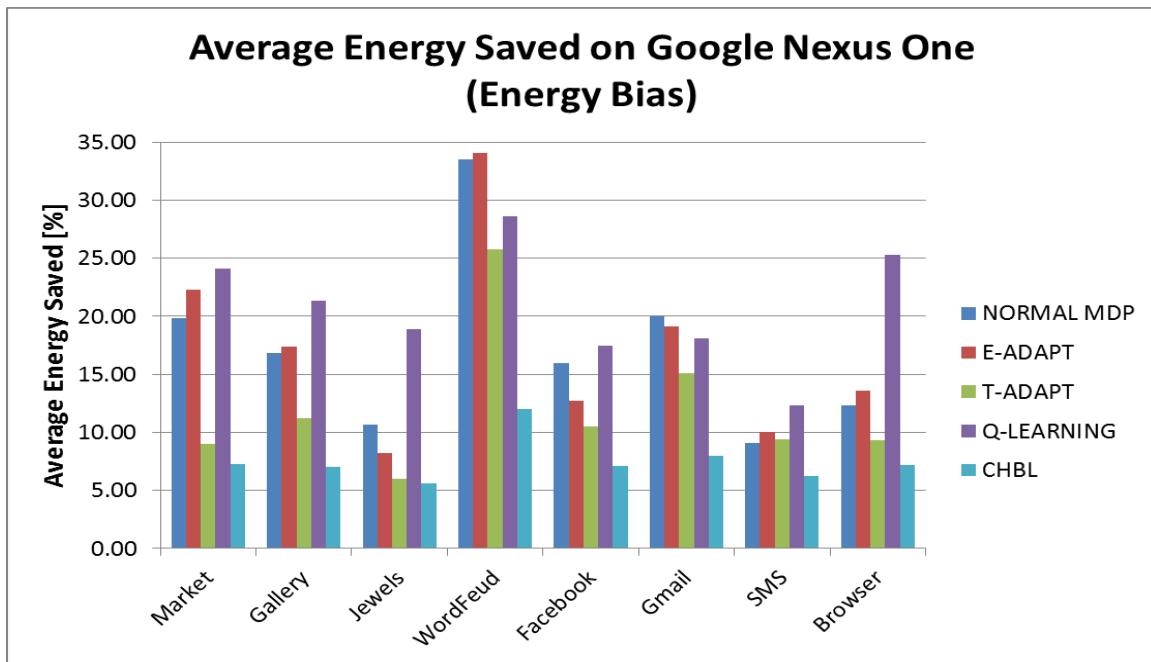


Figure 6.22: Average successful prediction rate on HTC Dream for real users (energy bias)

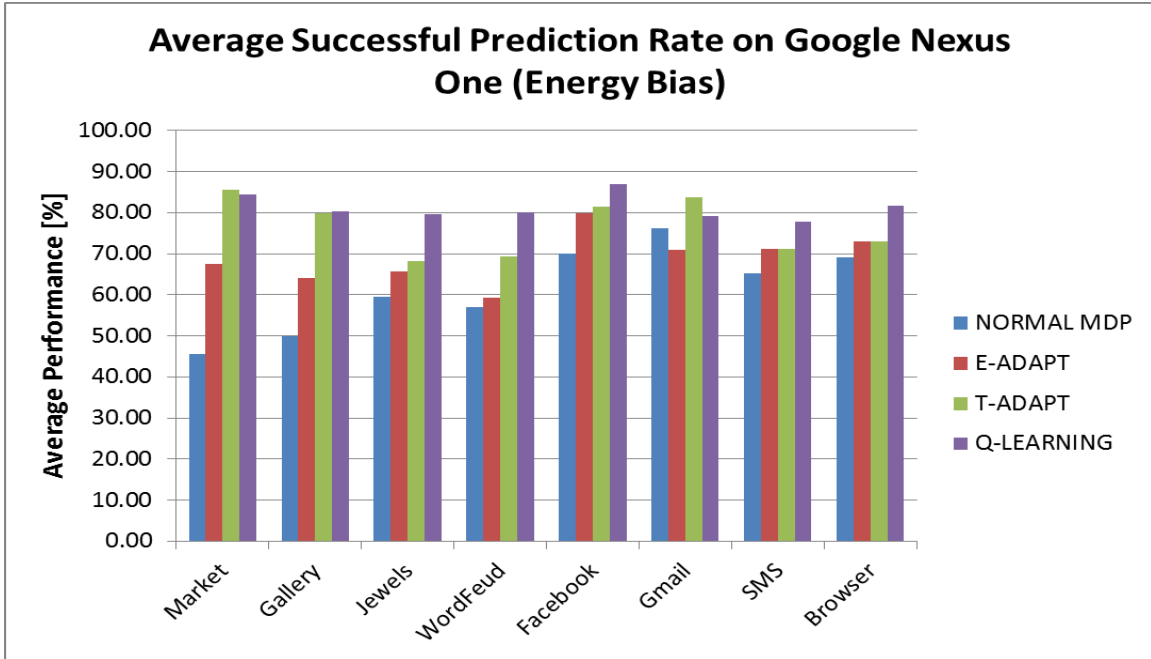


Figure 6.23: Average successful prediction rate on Google Nexus One for real users (energybias)

Overall, the proposed *AURA* framework with the *E-ADAPT* or *Q-LEARNING* schemes enables as much as 29% energy savings over the default device settings for frequency and screen backlight without noticeably degrading user QoS, and improves upon prior work (*CHBL* [8]) by as much as 5× for some applications and devices.

## Chapter 7

### Strategy 2: Data and Location Interface Energy Optimization

This strategy's focus is on exploiting machine learning algorithms to discover opportunities for energy saving in mobile embedded systems through the dynamic adaptation of data transfer and location network interface configurations without any explicit user input. Consequently, energy-performance tradeoffs are enabled that are unique to each user. This is accomplished by learning both the spatiotemporal and device contexts of a user through the application of a given algorithm and using these to control device network configuration to save energy. In other words, given a set of input contextual cues, the algorithms will exploit learned user context to dynamically classify the cues into a system state that precisely governs how data and location interfaces are utilized. The goal is to achieve a state classification that saves energy while maintaining user satisfaction. In this chapter, the use of four different classes of machine learning algorithms (*i*) *linear discriminant analysis*, (*ii*) *linear logistic regression*, (*iii*) *k-nearest neighbor*, and (*iv*) *non-linear logistic regression with neural networks* (explained in detail in Chapter 5), is proposed and demonstrated on predicting user data/location usage requirements using spatiotemporal and device contexts. These algorithms are tested on both synthetic and real-world user usage patterns, which demonstrate that high and consistent prediction rates are possible. The proposed techniques are also compared with prior work on device configuration prediction using self-organizing maps [45] and energy-aware location sensing [17], showing an improvement upon these state-of-the-art techniques. Section 7.1 discusses the acquisition and creation of real and synthetic user profiles that are used in the analysis studies, Section 7.2 describes the device power modeling effort, and Section 7.3 presents the results of the experimental studies.

## 7.1 User Interaction Studies

In order to compare the relative effectiveness of the different algorithms at predicting a user's data/location usage requirements, five real user usage profiles for four different Android smartphones (HTC myTouch 3G, Google Nexus One, Motorola Droid X, HTC G2, and Samsung Intercept) were collected over a one week period with a custom *Context Logger* application. The application logged user context data on external storage, which was acquired at the end of the one week session and used in the algorithm analysis.

### 7.1.1 Context Logger

A custom *Context Logger* application was created that ran in the background as an Android service and gathered both spatiotemporal and device usage attributes at a one minute interval. Table 7.1 lists the attributes recorded by the logger and used for algorithm analysis and indicates whether the attribute was a continuous variable (floating point), discrete variable (integer), or logical variable (true/false). The *GPS Satellites* attribute is used as an indirect correlation to GPS signal strength and *WiFi RSSI* is a measure of WiFi signal strength. In addition to more common device attributes such as *Battery Level* and *CPU Utilization*, we gathered several uncommon OS attributes: *Context Switches*, *Processes Created*, *Processes Running*, and *Processes Blocked*. Using these as inputs to the machine learning algorithms was an attempt to aid prediction. The three target variables (*Data Needed*, *Coarse Location Needed*, and *Fine Location Needed*) were obtained by examining the requested Android permissions of all of the device's current running foreground applications and services. The *Device Moving* attribute was determined by using the accelerometer sensor and a metric for movement that is the sum of the unbiased variance of X, Y, and Z acceleration [15], given as:

$$Var(m_1 \dots m_n) = \frac{\sum_{i=1}^N m_i^2 - \frac{1}{N} (\sum_{i=1}^N m_i)^2}{N - 1} \quad (7.1)$$

$$Metric = Var(x_1 \dots x_n) + Var(y_1 \dots y_n) + Var(z_1 \dots z_n) \quad (7.2)$$

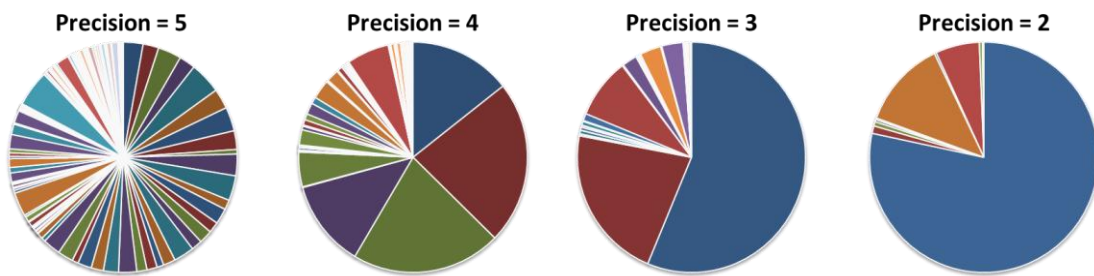
**Table 7.1: Recorded data attributes**

<b>Context</b>	<b>Attribute</b>	<b>Type</b>
Temporal	Day of week	<i>Discrete</i>
	Time of day	<i>Discrete</i>
Spatial	Latitude	<i>Continuous</i>
	Longitude	<i>Continuous</i>
	GPS Satellite Count	<i>Discrete</i>
	WiFi RSSI	<i>Discrete</i>
	Number of WiFi APs Available	<i>Discrete</i>
	3G Network Signal Strength	<i>Discrete</i>
	Device Moving	<i>Logical</i>
	Ambient Light	<i>Discrete</i>
Device	Call State	<i>Discrete</i>
	Battery Level	<i>Discrete</i>
	Battery Status	<i>Discrete</i>
	CPU Utilization	<i>Continuous</i>
	Context Switches	<i>Discrete</i>
	Processes Created	<i>Discrete</i>
	Processes Running	<i>Discrete</i>
	Processes Blocked	<i>Discrete</i>
	Screen On	<i>Logical</i>
Targets	Data Needed	<i>Logical</i>
	Coarse Location Needed	<i>Logical</i>
	Fine Location Needed	<i>Logical</i>

### 7.1.2 Data Preparation

As mentioned earlier, GPS location coordinates were recorded along with the other attributes at one minute intervals. The Android SDK GPS location data returns the user's longitude and latitude coordinates in decimal degrees as reported by the onboard GPS chipset [54]. Although exact accuracy is dependent on the actual GPS hardware, the returned values were truncated to a given precision and each longitude and latitude coordinate pair was mapped to a unique location

identifier. The truncated location resolution generalized the number of unique locations in which a user spends his/her time, given that for example, a user's home may consist of several different samples of different longitude and latitude pairs. In addition, temporal conditions can be applied to further reduce the number of unique locations (e.g. disregard locations where user spent less than  $x$  minutes). Figure 7.1 shows the effect of truncation and application of temporal conditions for a real user and how the primary locations where the user spent most of his/her time are revealed. For our study we used a location precision of 4 decimal places.



**Figure 7.1: Unique locations identified for varying GPS precisions**

The desired data/location interface configurations were partitioned into eight different states based on the desired target variables (*Data Required*, *Coarse Location Required*, and *Fine Location Required*). Table 7.2 maps the logical values of the three target variables to a state. The states define the device's current required resources. *Efficiently predicting one of these 8 states using temporal, spatial, and device context input variables in Table 1 may ultimately allow opportunistic shutdown of location/wireless radios.* If all interfaces are enabled, they would consume a significant amount of energy in their idle states without this dynamic control.

**Table 7.2: Interface configuration states**

<b>State</b>	<b>Data Required</b>	<b>Coarse Location Required</b>	<b>Fine Location Required</b>
1	No	No	No
2	No	No	Yes
3	No	Yes	No
4	No	Yes	Yes
5	Yes	No	No
6	Yes	No	Yes
7	Yes	Yes	No
8	Yes	Yes	Yes

### **7.1.3 Real User Profiles**

The Context Logger application was distributed to five different mobile device users, and logged their context data over the course of one week. Figure 7.2 demonstrates the relative state distributions as described in Table 2 for the five different real users. As can be seen, states 2 – 4 are never realized as data was always required when location was required. In addition, the distributions highlight that each user had a considerably different usage pattern. User 1 can be categorized as a minimal user, where the user rarely utilized their phone with only brief periods of interaction, while users 2 and 5 used their phones rather frequently and for longer periods of time. Users 3 and 4 can be categorized as moderate users, primarily utilizing their devices for only certain times of the day.

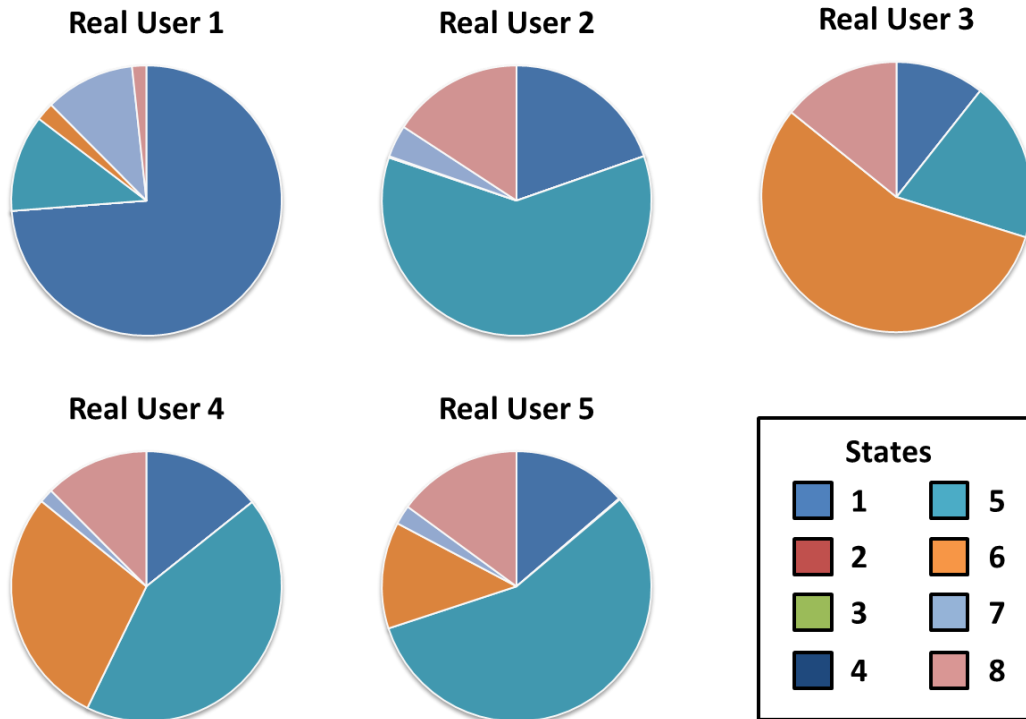


Figure 7.2: Real user state distributions

#### 7.1.4 Synthetic User Profiles

Prior to the algorithm evaluation studies, a subset of *synthetic* user profiles were created for five different *idealized* and *generalized* models of average user usage patterns including the following: (i) *8 – 5 Business Worker*, (ii) *College Student*, (iii) *Social Teenager*, (iv) *Stay At Home Parent*, and (v) *Busy Traveler*. Both an indoor/outdoor location timeline and an interface state profile were created for each synthetic user, as shown in Figure 7.3. Given the difficulty of generating realistic device system data, such as context switches, CPU utilization, and processes created, only a subset of the attribute space was considered. The remaining attributes were based on both the desired state and/or location. For example, if a user was at an outdoor location, larger GPS satellite values and weak WiFi RSSI values were used as opposed to when the user was indoors.



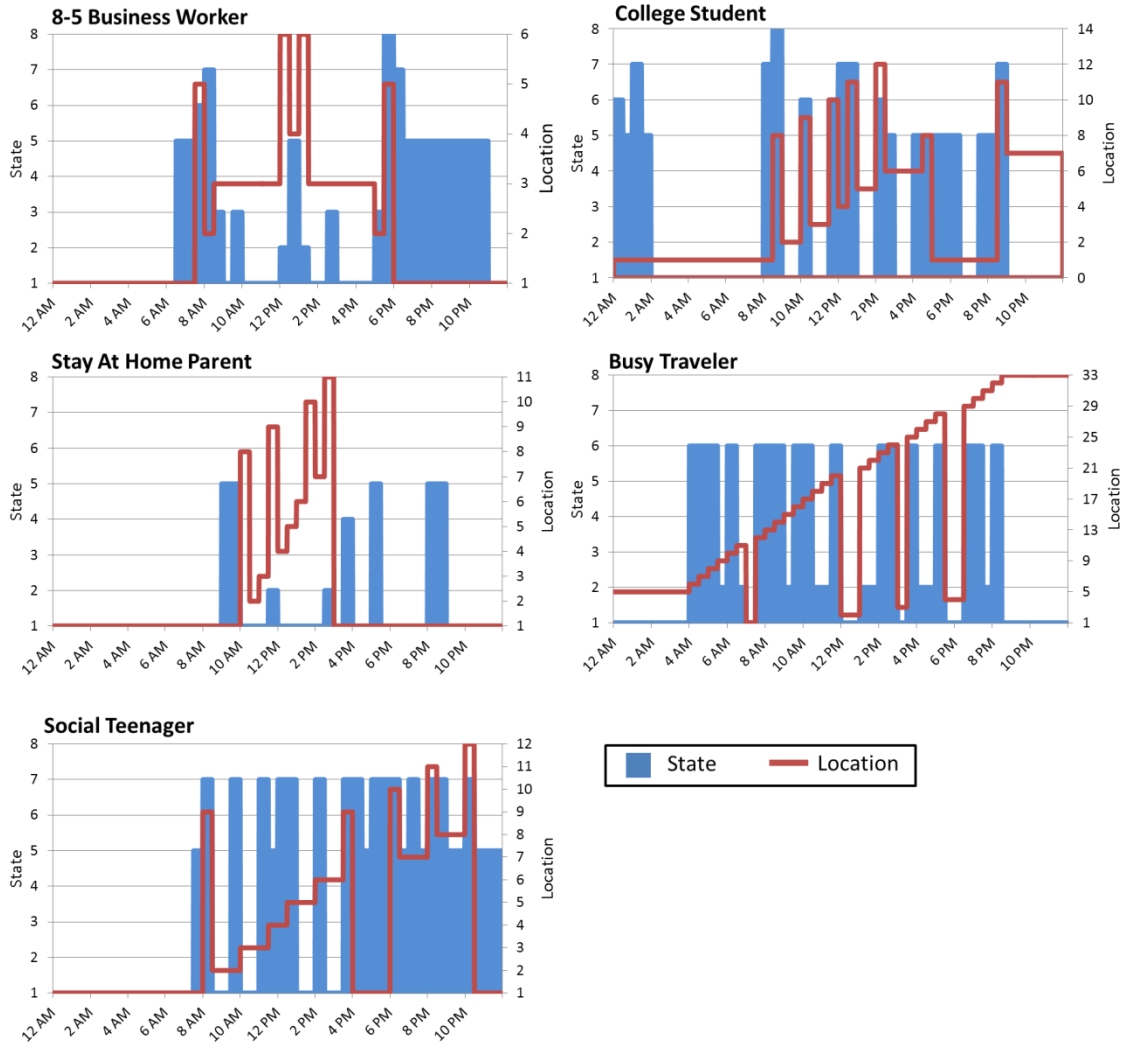


Figure 7.3: Synthetic user profiles

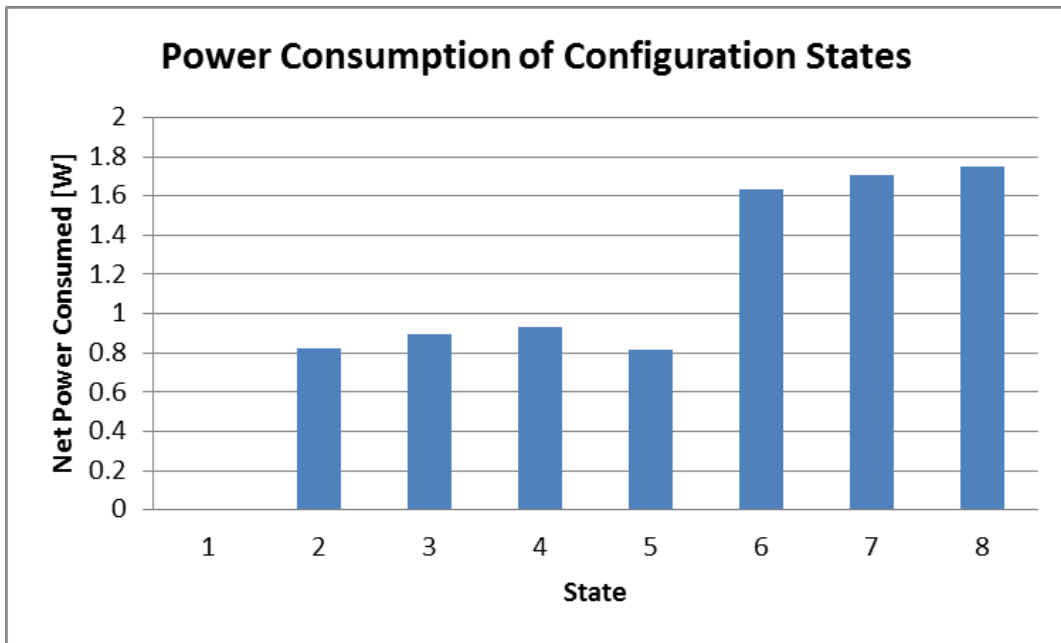
## 7.2 Device Power Modeling

In order to quantify the energy-effectiveness of using machine learning algorithms to predict energy-optimal device states, power analysis was performed on the real Android based smartphones used, with the goal of creating power models for the data and location interfaces. We use a variant of Android OS 2.3.3, (Gingerbread) and the Android SDK Revision 11 as our baseline OS. We built our power estimation models using real power measurements, by instrumenting the contact between the smartphone and the battery, and measuring current using the Monsoon Solutions power monitor [55] – the same method as was used for first strategy in

this thesis. The data/location interfaces were manually enabled one by one and power traces were gathered for each interface in their active and idle states. The power traces from the Monsoon Power Tool were then used to obtain average power consumption measurements for each interface. The power consumption for each of the interfaces in their idle states is shown in Table 7.3, and the power consumption for each of the eight configurations states relative to having none of the interfaces enabled is shown in Figure 7.4.

**Table 7.3: Idle power consumption for data and location interfaces**

WiFi Idle	3G Idle	GPS Idle
0.0852 W	0.0708 W	0.0469 W



**Figure 7.4: Relative power consumption of configuration states**

## 7.3 Experimental Results

### 7.3.1 Prediction Accuracy Analysis

Recall that the input attributes for the learning algorithms come from the gathered spatiotemporal and device context data (Table 7.1), and the predicted output is one of the eight interface configuration states (Table 7.2). To evaluate the prediction accuracy of the different algorithms, the data for each real user was randomly partitioned into training and test sets using a common 80/20 partitioning scheme. The algorithms were then trained on the training data and evaluated on the test data. This was repeated five times for each implementation and the net prediction accuracy is presented in Figure 7.4.

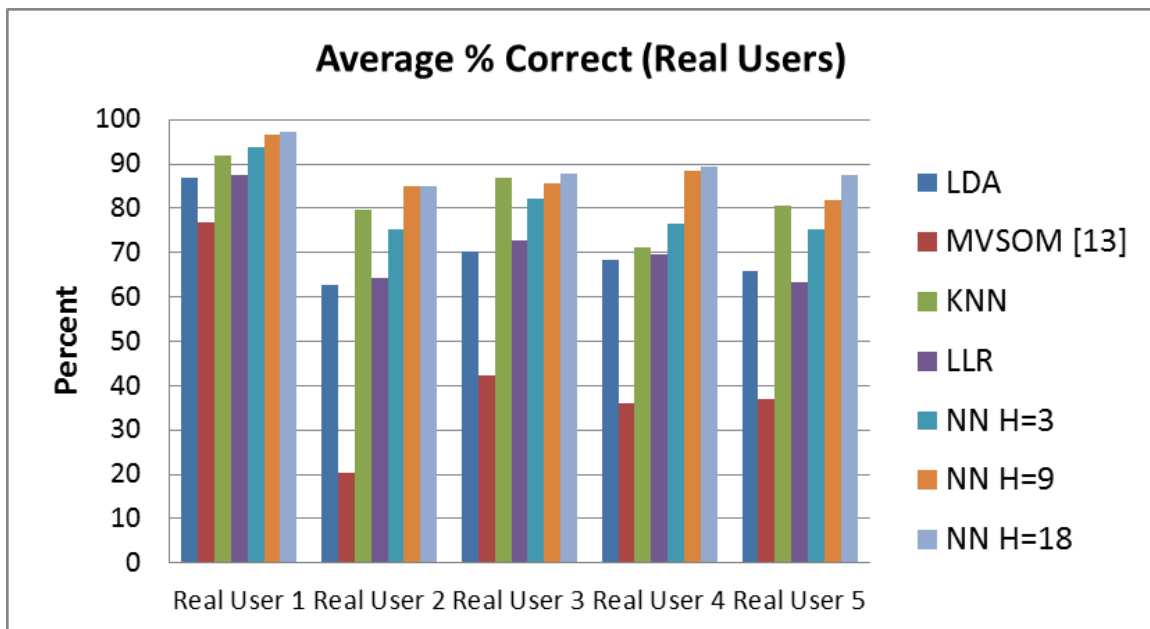


Figure 7.5: Real user algorithm prediction accuracy

Three different neural network (NN) implementations with a varying number of hidden units, equal to the total ( $H=18$ ), half ( $H=9$ ), and one-sixth ( $H=3$ ) the size of the attribute space were evaluated. We compare the prediction accuracy of our algorithms with the configuration

prediction strategy presented in [45] (*MVSOM – Missing Values Self-Organizing Map*). As illustrated in Figure 7.5, the application of neural networks with a number of hidden units *of at least* half the size of the attribute space resulted in the highest prediction rates. K-nearest neighbor (KNN), linear logistic regression (LLR), and linear discriminant analysis (LDA) also performed fairly well, with prediction accuracies in the range of 60 – 90 %. However these approaches were much more sensitive to the usage pattern. MVSOM performed the worst and had a high degree of variance in both the usage pattern and random training data selection.

The same algorithms were applied to the synthetic user profiles; however, the attribute space was reduced to only *Day, Time, Location, GPS Satellites, WiFi RSSI, Network Signal Strength, Data Needed, Coarse Location Needed, and Fine Location Needed*. The same strategy for selecting numbers of hidden units for the neural network implementations was applied for the reduced attribute space. Figure 7.6 illustrates the algorithm prediction rates for the synthetic users showing similar trends as in the real user data.

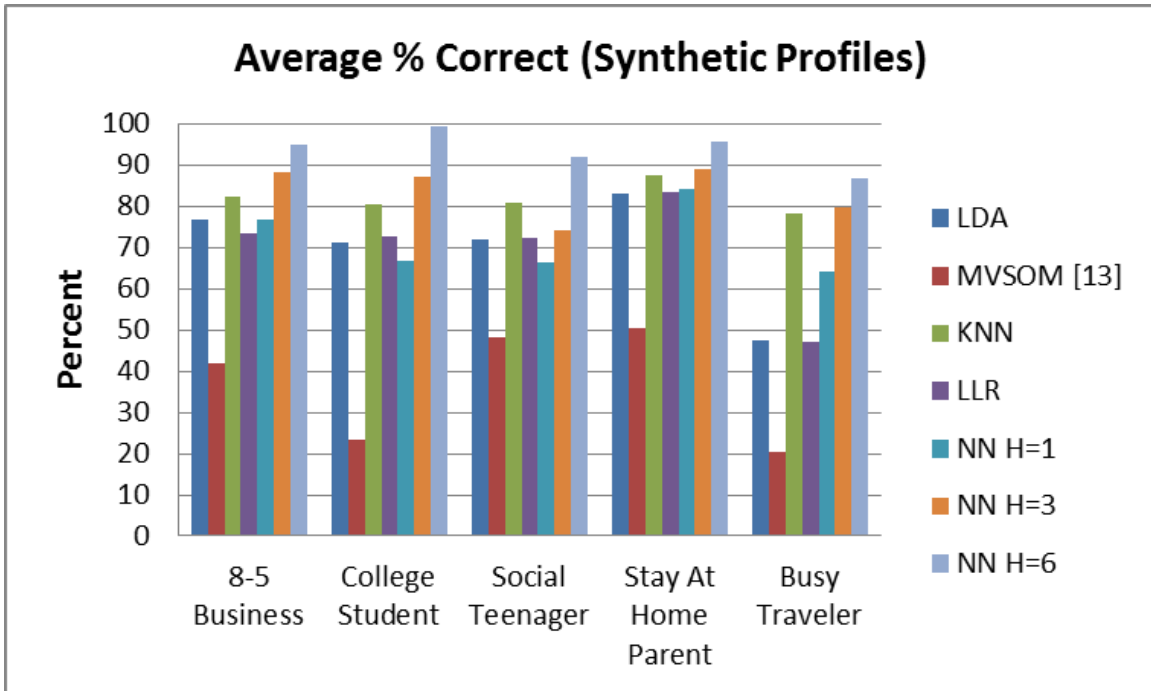


Figure 7.6: synthetic user algorithm prediction accuracy

### 7.3.2 Energy Savings

It is important to note that despite high prediction accuracy, the amount of potential energy savings is still highly dependent on the user usage pattern and if the algorithms are positively or negatively predicting states where energy can be conserved. Figures 7.7 and 7.8 illustrate the energy savings achieved by the individual algorithms when the algorithm's prediction target states are applied to the real and synthetic user profiles, and compare them against the *VRL technique (Variable Rate Logging [17])*. Note that as *VRL* does not predict system state, results for its prediction accuracy are not shown in Figures 7.5 or 7.6. Although simpler linear models can achieve high energy savings, it is important to note that energy savings themselves are not good discriminants of an algorithm's *goodness* because user satisfaction must also be considered. For example, if a user spends a significant amount of time in the energy consuming state 8 and the algorithms are predicting a less energy-consuming state during these instances, then more

energy can be conserved at the cost of user-satisfaction. Directly correlating the prediction accuracy of the algorithms is especially important for highly interactive users such as users 2 and 5, as opposed to minimally interactive users, e.g., user 1.

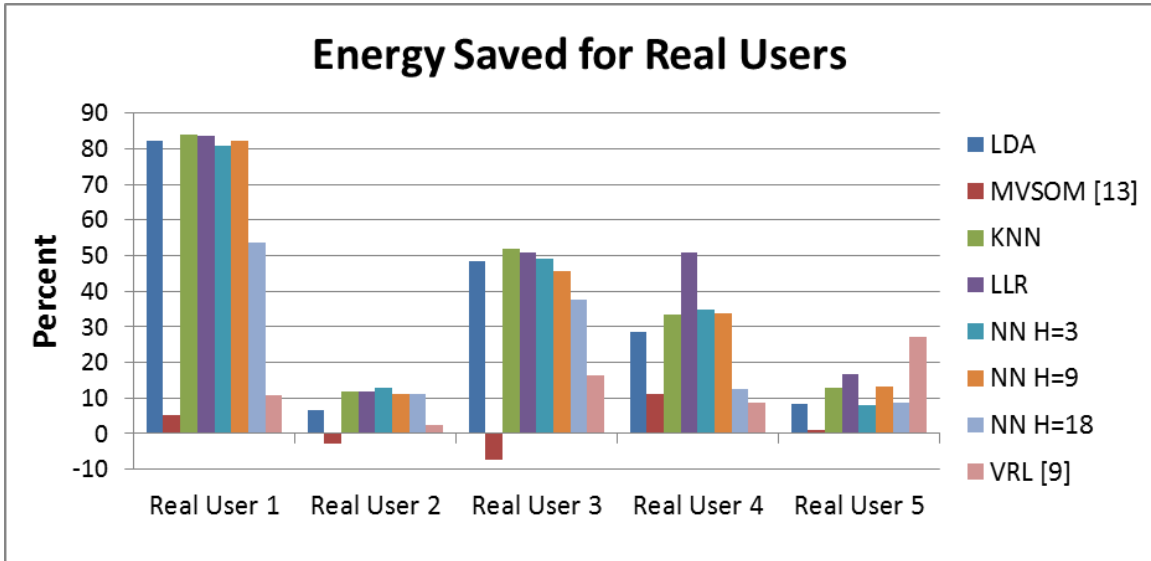


Figure 7.7: Percent energy saved for real users

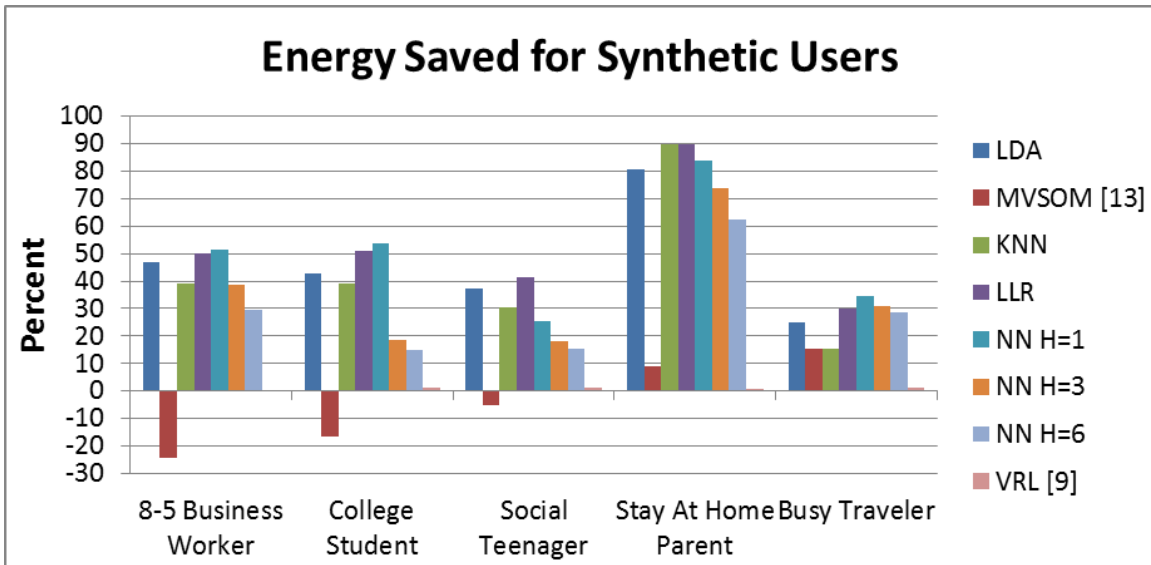


Figure 7.8: Percent energy saved for synthetic users

All energy savings are relative to the baseline case for Android systems without proactive multi-

network interface management. Lower prediction and more generalized models result in the highest energy savings as in the case of LDA and LLR. However, again, these higher savings come at the cost of degraded user satisfaction. *KNN overall performs fairly well in terms of both prediction accuracy and energy savings potential, as does the nonlinear logistic regression with NN approach.* With the latter, an important point to note is that prediction accuracy is proportional to the complexity of the neural network and indirectly proportional to the net energy savings. This outcome is expected as less complex neural networks will result in more generalized models relaxing the constraint for inaccurate predictions that result in higher energy savings. MVSOM, with its low prediction rates, also led to instances of negative energy savings, as it often predicted higher energy states when the true target state was one of less energy consumption. Thus we believe that the MVSOM approach is not very viable for use in mobile embedded systems. *VRL's energy saving capability is constrained because it does not disable device interfaces (only deactivates location logging or reduces logging rate), ignoring idle energy consumption. Overall, compared to VRL, the average energy savings of the KNN algorithm is 25.6%, and that of the NN approaches is 11.7% (H=18), 24.1% (H=9), and 24% (H=3) for real user patterns.*

### **7.3.3 Implementation Overhead**

The prediction and energy savings results presented in the previous sections were obtained using a Python implementation of the algorithms on a 2.6 GHz Intel® Core i5™ processor. When considering real-world implementation, it is important to consider the implementation overhead of the individual algorithms. Current hardware in mobile devices on the market today is quickly catching up to the abilities of modern stationary workstations (e.g. Google's Galaxy Nexus – 1.2 GHz duo-core processor). We determined the maximum implementation overhead for the

Google Nexus One 1 GHz Qualcomm QSD 8250 Snapdragon ARM processor [59], as shown in Table 7.3 below. The values shown in the table are average prediction times for each algorithm at runtime, and do not include initial training time. KNN’s run time is several orders of magnitude larger than any of the other algorithms, because all computations are deferred until classification. Therefore, although KNN is as good as or better than the neural network (NN) based approach in terms of energy savings and prediction, *the non-linear logistic regression with NN approach is preferable because of its fast execution time.*

**Table 7.4: Average algorithm run times**

Algorithm	Average Run time (seconds)
LDA	0.00361
MVSOM [13]	2.15023
KNN	254.131
LLR	0.00307
NN (all 3 variants)	0.02501
VRL [9]	0.05140

*In summary*, the LDA and LLR approaches have the lowest implementation overhead and can result in high energy savings, but often at the cost of user satisfaction. Although the KNN approach is very effective in terms of prediction accuracy and energy savings, its unreasonable implementation overhead renders it unacceptable for real-world applications. The prior work with MVSOM [45] provides low energy savings as a result of its poor prediction accuracy, and takes a long time to run; whereas VRL [17] has low run time but also very low energy savings. *The non-linear logistic regression with neural network approach that uses the fast scaled conjugate gradient training method and with the number of hidden unites equal to half the attribute space* provides good accuracy, good energy savings, and demonstrates the best



adaptation to various unique user usage patterns, while maintaining a low implementation overhead.

## Chapter 8

### Summary and Future Work

Today's smartphones and other mobile battery operated devices are indispensable tools that serve a wide array of purposes, including, but not limited to: communication, entertainment, business, social networking, and travel. Consumers are using mobile devices for more power-intensive tasks and for longer periods than ever before, and demands for performance and battery lifetime are constantly increasing. However, because advances in lithium ion battery technology (used predominately in current mobile devices) are occurring much more slowly than advances in other smartphone technologies, intelligent energy-optimization methods are required to keep up with these demands. These methods must be able to reduce energy consumption while still offering acceptable QoS to the user. Machine learning, a scientific discipline focused on the development and use of algorithms that allow a computer to evolve and learn optimal choices based on empirical data, can be used for this purpose.

#### **8.1 Summary**

This thesis introduced two novel “smart” energy optimization strategies for mobile devices that balance energy consumption and user QoS using machine learning algorithms.

The first strategy optimizes CPU and backlight energy by taking advantage of user idle time between events. This is accomplished through use of a middleware framework with MDP/Q-Learning based power management algorithms. A Bayesian application classifier is also included in the framework to categorize applications based on user interaction, ultimately allowing the power management algorithms to be adjusted to each application more effectively. The strategy is evaluated on both real and emulated user interaction patterns on two different

smartphones, obtaining average energy savings of 29% compared to the stock power manager and up to 5× improvement in savings over prior work without perceptible impact on user QoS.

The second strategy optimizes data and location interface energy consumption by predicting optimal device configurations based on user history data. Predicting optimal device configuring effectively allows opportunistic shutdown of location/wireless radios, thus saving energy. Several machine learning techniques are evaluated on their prediction ability, including linear discriminant analysis (LDA), linear logistic regression (LLR), non-linear logistic regression with neural networks (NN), and k-nearest neighbor (KNN). The techniques are evaluated on both synthetic and real-world user usage patterns, and it is shown that non-linear logistic regression with neural networks is capable of offering good prediction accuracy, good energy savings, superior adaptation to various unique user usage patterns, and low usage overhead. Specifically, the neural network and k-nearest neighbor algorithms demonstrated up to a 90% successful prediction, showing approximately 50% improvement over a previous state-of-the-art prediction approach, and were capable of achieving approximately 85% energy savings for minimally active users, improving the energy savings of a previous state-of-the-art location logging algorithm by 24%.

## **8.2 Future Work**

Both of strategies the described in this thesis show promising energy saving results with negligible impact on user QoS. However, much work can be done to improve the strategies. This section discusses some of the future work that will enable these smart machine learning energy optimization strategies to reach their full potential.

Ongoing work for the first strategy (described in Chapter 6) is focusing on a more in-depth analysis of the overhead resulting from the background power management algorithms. Because the energy savings results were acquired using linear screen backlight and CPU power models, the energy consumed by the background algorithms was not taken into consideration.

For the second strategy, plans for future work include implementation of the energy-saving machine learning techniques on a real mobile device. The results in Chapter 7 were obtained using a Python implementation of the algorithms on a 2.6 GHz Intel® Core i5™ processor. Learning algorithm training is an important consideration when considering a real-world implementation. There are several approaches to when and how the algorithms can be trained quickly and in an energy-efficient manner without affecting user QoS on a real mobile platform. One possibility is to train only while the mobile device is plugged in and charging. Another solution would be to offload the training computations to a network server or a cloud computing service.

There are also other plans for future work for the second strategy. One such plan is to perform Principal Component Analysis (PCA) on the input variables. PCA helps to reduce the attribute space by converting a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called *principal components*. Another plan is to perform a study that correlates successful prediction with actual user satisfaction, much like in the first strategy in this thesis. Finally, we would like to implement and test more machine learning algorithms, such as a Support Vector Machine (SVM).

Although there always improvements to be made in the field of software energy optimization for mobile embedded systems, the work presented in this thesis brings us one step closer to being able to meet consumer demands of performance and battery lifetime.

## References

- [1] MobiThinking, “Global mobile statistics 2011,” <http://mobithinking.com>, April 2011.
- [2] Micro Power White Paper, “Using Lithium Polymer Batteries in Mobile Devices,” [http://www.micro-power.com/userfiles/file/wp\\_polymer\\_final-1274743697.pdf](http://www.micro-power.com/userfiles/file/wp_polymer_final-1274743697.pdf) [Online].
- [3] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, A. Rice, “Exhausting Battery Statistics: Understanding the energy demands on mobile handsets,” in MOBIHELD ’10, pp. 9-14, Aug. 2010.
- [4] J. Kang, S. Seo, J. W. Hong, “Usage Pattern Analysis of Smartphones,” in APNOMS ’11, pp. 1-8, Nov. 2011.
- [5] J. Eberle, G. P. Perrucci, “Energy Measurements Campaign for Positioning Methods on State-of-the-Art Smartphones,” in CCNC ’11, pp. 937-941, May 2011.
- [6] N. Balasubramanian, A. Balasubramanian, A. Venkataramani, “Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications,” in IMC ’09, pp. 280-293, Nov. 2009.
- [7] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, J. A. Landay, “MyExperience: A System for In Situ Tracing and Capturing of User Feedback on Mobile Phones,” in MOBISYS, pp. 57-70, 2007.
- [8] A. Shye, B. Scholbrock, G. Memik, “Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures,” in MICRO-42 ’09, pp. 168-178, Jan. 2009.
- [9] H. Petander, “Energy-Aware Network Selection Using Traffic Estimation,” in MICNET ’09, pp. 55-60, Sept. 2009.
- [10] M. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, M. J. Neely, “Energy-Delay Tradeoffs in Smartphone Applications,” in MOBISYS ’10, pp. 255-270, Jun. 2010.
- [11] I. Constandache, S. Gaonkar, M. Saylor, R. R. Choudhury, L. Cox, “EnLoc: Energy-Efficient Localization for Mobile Phones,” in INFOCOM ’09, pp. 19-25, Jun. 2009.
- [12] K. Lin, A. Kansal, D. Lyberopoulos, F. Zhao, “Energy-accuracy trade-off for continuous mobile device location,” in MOBISYS, pp. 285-298, Jun 2010.
- [13] F. B. Abdesslem, A. Phillips, T. Henderson, “Less is more: energy-efficient mobile sensing with SenseLess,” in MOBIHELD ’09, pp. 61-62, Aug. 2009.

- [14] J. Paek, J. Kim, R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," in MOBISYS '10, pp. 299-314, Jun. 2010.
- [15] I. Shafer, M. L. Chang, "Movement detection for power-efficient smartphone WLAN localization," in MSWIM '10, pp. 81-90, Oct. 2010.
- [16] M. Youssef, M. A. Yosef, M. El-Derini, "GAC: energy-efficient hybrid GPS-accelerometer-compass GSM localization," in GLOBECOM '10, pp. 1-5, Dec. 2010.
- [17] C. Lee, M. Lee, D. Han, "Energy efficient location logging for mobile device," in SAINT '11, pp. 84, Oct. 2010.
- [18] K. Nishihara, K. Ishizaka, J. Sakai, "Power saving in mobile devices using context-aware resource control," in ICNC '10, pp. 220-226, 2010.
- [19] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, N. Sadeh, "A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition," in MOBISYS '09, pp. 179-192, Jun. 2009.
- [20] Z. Zhuang, K. Kim, J. P. Singh, "Improving Energy Efficiency of Location Sensing on Smartphones," in MOBISYS, pp. 315-330, 2010.
- [21] T. L. Cheung, K. Okomoto, F. Maker, X. Liu, V. Akella, "Markov Decision Process (MDP) Framework for Optimizing Software on Mobile Phones," in EMSOFT '09, pp. 11-20, Oct. 2009.
- [22] W. Liang, P. Lai, "Design and Implementation of a Critical Speed-based DFS Mechanism for the Android Operating System," in EMS '10, pp. 1-6, Sept. 2010.
- [23] R. Jurdak, P. Corke, D. Dharman, G. Salagnac, "Adaptive GPS Duty Cycling and Radio Ranging for Energy-Efficient Localization," in SENSYS '10, pp. 57-70, Nov. 2010.
- [24] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, A. T. Campbell, "The Jigsaw Continuous Sensing Engine for Mobile Phone Applications," in SENSYS '10, pp. 71-84, Nov. 2010.
- [25] M. B. Kjaergaard, S. Bhattacharya, H. Blunck, P. Nurmi, "Energy-efficient trajectory tracking for mobile devices," in MOBISYS '11, pp. 307-320, Jun. 2011.
- [26] Y. Liu, S. Lu, Y. Liu, "COAL: Context Aware Localization for High Energy Efficiency in Wireless Networks," in WCNC '11, pp. 2030-2035, May 2011.
- [27] S. P. Tarzia, P. A. Dinda, R. P. Dick, G. Memik, "Display Power Management Policies in Practice," in ICAC '10, pp. 51-60, Jun. 2010.
- [28] L. Bloom, R. Eardley, E. Geelhoed, M. Manahan, P. Ranganathan, "Investigating the Relationship Between Battery Life and User Acceptance of Dynamic, Energy-Aware Interfaces on Handhelds," in HCI '09, pp. 13-24, Sept. 2004.

- [29] T. Harter, S. Vroegindeweij, E. Geelhoed, M. Manahan, P. Ranganathan, “Energy-Aware User interfaces: An Evaluation of User Acceptance,” in CHI ‘04, pp. 199-206, Apr. 2004.
- [30] M. Bi, I. Crk, C. Gniady, “IADVS: On-Demand Performance for Interactive Applications,” in HPCA ’10, pp. 1-10, Apr. 2010.
- [31] A. Mallik, J. Cosgrove, R. Dick, G. Memik, P. Dinda, “PICSEL: Measuring User-Perceived Performance to Control Dynamic Frequency Scaling,” in ASPLOS ‘08, Mar. 2008.
- [32] S. Choi, et al., “A selective DVS technique based on battery residual mi2processors and microsystems,” Elsevier Sc., 30(1):33–42, 2006.
- [33] F. Qian, et al., “TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation,” in ICNP, 2010.
- [34] K. Choi, R. Soma, M. Pedram, “Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times,” in TCAD ‘04, 2004.
- [35] A. Weissel, F. Bellosa, “Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management,” in CASES ‘02, 2002.
- [36] C. H. Hsu, U. Kremer, “Single Region vs. Multiple Regions: A Comparison of Different Compiler-Directed Dynamic Voltage Scheduling Approaches,” in Workshop on PACS, 2002.
- [37] S. Lee, T. Sakurai, “Run-Time Voltage Hopping for Low-Power Real-Time Systems,” in DAC ‘00, 2000.
- [38] D. Shin, J. Kim, S. Lee, “Intra-Task Voltage Scheduling for Low-Energy, Hard Real-Time Applications,” IEEE Design and Testing, 2001.
- [39] F. Xie, M. Martonosi, S. Malik, “Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits,” in PLDI ‘03, 2003.
- [40] S. K. Card, T. P. Moran, A. Newell, “The Psychology of Human-Computer Interaction,” Hillsdale, NJ: Lawrence Erlbaum Assoc., 1983.
- [41] D. J. Simons, S. L. Franconeri, R. L. Reimer, “Change blindness in the absence of a visual disruption”, in Perception, 29:1143–1154, 2000.
- [42] D. J. Simons, R. A. Rensink, “Change blindness: Past, present, and future”, Trends in Cognitive Sciences, Vol.9 No.1 January 2005.
- [43] H. Jung, M. Pedram, “Improving the Efficiency of Power Management Techniques by Using Bayesian Classification,” in ISQED ’08, pp. 178-183, Mar. 2008.



- [44] Y. Tan, W. Liu, Q. Qiu, “Adaptive Power Management Using Reinforcement Learning,” in ICCAD ’09, pp. 461-467, Nov. 2009.
- [45] L. Batyuk, et al., “Context-aware device self-configuration using self-organizing maps” in OC ’11, pp. 13-22, June 2011.
- [46] T. Anagnostopoulos, C. Anagnostopoulos, S. Hadjiefthymiades, M. Kyriakakos, A. Kalousis, “Predicting the location of mobile users: a machine learning approach,” in ICPS ’09, pp. 65-72, July 2009.
- [47] T. Mantoro, et al., “Mobile user location determination using extreme learning machine,” in ICT4M, pp. D25-D30, 2011.
- [48] C. M. Bishop, “Pattern Recognition and Machine Learning,” 1st ed. New York: Springer Science+Business Media, 2006
- [49] E. Alpaydin, “Introduction to Machine Learning,” 2nd ed. Massachusetts: The MIT Press, 2010.
- [50] M. Moller, “Efficient Training of Feed-Forward Neural Networks,” Ph.D. dissertation, CS Dept., Aarhus Univ., Arhus, Denmark, 1997.
- [51] Chuck Anderson. Introduction to Reinforcement Learning. Website, October 2011. <http://www.cs.colostate.edu/~anderson/cs545/index.html/doku.php?id=notes:notes10b>.
- [52] S. Swanson. M.B. Taylor. “Greendroid: Exploring the next evolution of smartphone application processors,” Communications Magazine, IEEE. Vol 49. Issue 4. April 2011.
- [53] Google Android, official website, <http://www.android.com>
- [54] Android Developers, official website, <http://developer.android.com/index.html>.
- [55] Monsoon Solutions Inc., official website, <http://www.msoon.com/LabEquipment/PowerMonitor>, 2008.
- [56] D. Brodowski et al., “Linux CPUFreq – CPUFreq Governors. Linux Documentation,” <http://mjmwired.net/kernel/Documentation/cpu-freq>, March 2011.
- [57] J.J. Faraway, “Linear Models with R”, CRC Press, 2004
- [58] HTC, “HTC G1 Overview,” <http://www.htc.com/www/product/g1/specification.html>
- [59] HTC, “Google Nexus One Tech Specs,” <http://www.htc.com/us/support/nexus-one-google/tech-specs>
- [60] M. Guess, PCWorld, “Why Your Smartphone Battery Sucks,” May 18, 2011, Available: <http://www.htc.com/us/support/nexus-one-google/tech-specs>

- [61] Daniel P., PhoneArena, "Smartphone Displays – AMOLED vs LCD," Oct. 13, 2010, Available: [http://www.phonearena.com/news/Smartphone-Displays---AMOLED-vs-LCD\\_id13824](http://www.phonearena.com/news/Smartphone-Displays---AMOLED-vs-LCD_id13824)
- [62] S. Flosi, comScore, "comScore Reports January 2012 U.S. Mobile Subscriber Market Share," March 6, 2012, Available: [http://www.comscore.com/Press\\_Events/Press\\_Releases/2012/3/comScore\\_Reports\\_January\\_2012\\_U.S.\\_Mobile\\_Subscriber\\_Market\\_Share](http://www.comscore.com/Press_Events/Press_Releases/2012/3/comScore_Reports_January_2012_U.S._Mobile_Subscriber_Market_Share)
- [63] B. Dipert, EDN: Electronics Design, Strategy, News, "The Nexus One: Google hits a smartphone home run," Feb. 17, 2011, Available: [http://www.edn.com/article/512742-The\\_Nexus\\_One\\_Google\\_hits\\_a\\_smartphone\\_home\\_run.php](http://www.edn.com/article/512742-The_Nexus_One_Google_hits_a_smartphone_home_run.php)
- [64] R. Duan, B. Mingsong, C. Gniady, "Exploring memory energy optimizations in smartphones," in IGCC '11, pp. 1-8, Jul. 2011.
- [65] F. X. Lin, Z. Wang, R. LiKamWa, L. Zhong, "Reflex: Using Low-Power Processors in Processors without Knowing Them," in ASPLOS '12, pp. 13-24, Mar. 2012.

## Appendix A

### Source Code

This section presents the majority of the source code for the implementation of the two strategies. Sections A.1 through A.20 provide the source code files for the first strategy, and Sections A.20 through A.27 provide the source code files for the second strategy.

Section A.4 shows the source code for the main *AURA* Android application, and Section A.20 shows the code for the *AURA* background service that provides all the functionality of the framework, including the power management algorithms and application classification. Section A.2 shows the code that implements the *AURA* event emulator. All of the other source code files for strategy 1 are responsible for obtaining device attributes, DFS control, calculating energy savings, etc.

Section A.21 shows the main application code for the Context Logger Android application, and Section A.22 shows the code that provides continuous background logging of user and device attributes. A.23 shows the Python code used to simulate each of the machine learning algorithms. The rest of the source code files for strategy 2 show the Python classes that implement each of the machine learning algorithms.

#### **A.1 AppSettings.java (Strategy 1)**

```
package csu.research.AURA;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.widget.AdapterView;
import android.widget.CheckBox;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;
```

```

public class AppSettings extends Activity
{
    public static final String APPMON_APP_SETTINGS = "APPMON_APP_SETTINGS";
    private TextView mAppTitleTextView;
    private Spinner mAppClassSpinner;
    private CheckBox mStaticCheckBox;
    private EditText mTouchMeanTextBox;
    private EditText mKeyMeanTextBox;
    private EditText mTouchDevTextBox;
    private EditText mKeyDevTextBox;
    private String appName = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.settings);
        Intent intent = getIntent();
        if (intent.getAction().equals(APPMON_APP_SETTINGS))
        {
            initializeView(intent);
        }
        else
        {
            Toast.makeText(getApplicationContext(), "Error occurred opening
App Settings!", Toast.LENGTH_LONG).show();
            setResult(RESULT_CANCELED);

            finish();
        }
    }

    private void initializeView(Intent intent)
    {
        //Initialize main view
        mAppTitleTextView = (TextView)findViewById(R.id.AppTitleTextView);
        mAppClassSpinner = (Spinner)findViewById(R.id.app_settings_classification_spinner);
        mTouchMeanTextBox = (EditText)findViewById(R.id.app_settings_touch_mean_textbox);
        mTouchDevTextBox = (EditText)findViewById(R.id.app_settings_touch_dev_textbox);
        mKeyMeanTextBox = (EditText)findViewById(R.id.app_settings_key_mean_textbox);
        mKeyDevTextBox = (EditText)findViewById(R.id.app_settings_key_dev_textbox);
        mStaticCheckBox = (CheckBox)findViewById(R.id.app_settings_force_static_checkbox);

        appName = intent.getStringExtra("name");
        AppClassification classification = AppClassification.fromInt(intent.getIntExtra("classification", 0));

        if (appName != null)
        {
            String temp = appName;
            if (appName.contains("."))
            {
                temp = appName.substring(appName.lastIndexOf(".") + 1,
appName.length());
            }
        }
    }
}

```

```

        mAppTitleTextView.setText(String.format("%s (%s)",
temp.toUpperCase(), AppClassification.toString(classification)));
    }
    else
    {
        setResult(RESULT_CANCELED, null);
        finish();
    }
    //Spinner
    mAppClassSpinner.setAdapter(new ArrayAdapter<String>(AppSettings.this,
R.layout.spinner_item_view, AppClassification.CLASSIFIED_DESCRIPTIONS));
    if (classification == AppClassification.VeryLowInteraction)
    {
        mAppClassSpinner.setSelection(0);
    }
    else if (classification == AppClassification.LowInteraction)
    {
        mAppClassSpinner.setSelection(1);
    }
    else if (classification == AppClassification.LowMedInteraction)
    {
        mAppClassSpinner.setSelection(2);
    }
    else if (classification == AppClassification.MedInteraction)
    {
        mAppClassSpinner.setSelection(3);
    }
    else if (classification == AppClassification.MedHighInteraction)
    {
        mAppClassSpinner.setSelection(4);
    }
    else if (classification == AppClassification.HighInteraction)
    {
        mAppClassSpinner.setSelection(5);
    }
    else
    {
        mAppClassSpinner.setSelection(6);
    }

    //CheckBox
    mStaticCheckBox.setChecked(!intent.getBooleanExtra("isdynamic", true));

    //Edit Text
    mTouchMeanTextBox.setText(Integer.toString(intent.getIntExtra("touchmean", -
1)));
    mTouchDevTextBox.setText(Integer.toString(intent.getIntExtra("touchdev",
-1)));
    mKeyMeanTextBox.setText(Integer.toString(intent.getIntExtra("keymean", -
1)));
    mKeyDevTextBox.setText(Integer.toString(intent.getIntExtra("keydev", -
1)));
}

@Override
public void onBackPressed()
{
    Intent intent = new Intent();
    intent.putExtra("name", appName);
    intent.putExtra("isdynamic", !mStaticCheckBox.isChecked());
    intent.putExtra("classification",
mAppClassSpinner.getSelectedItemPosition() + 1);

```

```

        try
        {
            Integer tm = Integer.parseInt(mTouchMeanTextBox.getText().toString());
            Integer td = Integer.parseInt(mTouchDevTextBox.getText().toString());
            Integer km = Integer.parseInt(mKeyMeanTextBox.getText().toString());
            Integer kd = Integer.parseInt(mKeyDevTextBox.getText().toString());
            if (tm != null && td != null && km != null && kd != null)
            {
                int touchMean = tm.intValue();
                int touchDev = td.intValue();
                int keyMean = km.intValue();
                int keyDev = kd.intValue();
                intent.putExtra("touchmean", touchMean);
                intent.putExtra("touchdev", touchDev);
                intent.putExtra("keymean", keyMean);
                intent.putExtra("keydev", keyDev);
                setResult(RESULT_OK, intent);
            }
            else
            {
                setResult(RESULT_CANCELED, null);
            }
        }
        catch (NumberFormatException e)
        {
            Toast.makeText(getApplicationContext(), "Error parsing numbers!",
            Toast.LENGTH_LONG).show();
            setResult(RESULT_CANCELED, null);
        }
        finish();
    }
}

```

## A.2 AppSimulator.java (Strategy 1)

```

package csu.research.AURA;

import java.util.ArrayList;
import java.util.Random;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.os.SystemClock;
import android.provider.Settings.SettingNotFoundException;
import android.view.KeyEvent;
import android.view.MotionEvent;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.EditText;

```

```

import android.widget.Spinner;
import android.widget.TextView;
import android.widget.ToggleButton;

@SuppressWarnings("unused")
public class AppSimulator extends Activity
{
    private Spinner mAppSpinner;
    private Spinner mAlgSpinner;
    private TextView mResultsTextView;
    private EditText mTouchEditText;
    private EditText mKeyEditText;
    private CheckBox mTouchCheckBox;
    private CheckBox mKeyCheckBox;
    private ToggleButton mSimButton;
    private ArrayList<Integer> mFreqModes;
    private String mCurrentAppMode;
    private String mCurrentAlgMode;
    private Thread mRunTestThread;
    private Intent mSimTouchEvent;
    private Intent mSimKeyEvent;
    private int mAveragKeyTime;
    private int mAveragTouchTime;

    private static final String[] APPMODES = new String[] {"VERY LOW INTERACTION
APP", "LOW INTERACTION APP", "LOW MED INTERACTION APP", "MED INTERACTION APP", "MED
HIGH INTERACTION APP", "HIGH INTERACTION APP", "VERY HIGH INTERACTION APP" };
    private static final String[] ALGMODES = new String[] {"CONSTANT ALGORITHM",
"STOCHASTIC ALGORITHM", "MOVING STOCHASTIC", "RANDOM", "HIGH BURST - LONG PAUSE" };
    private static final int VERY_LOW_APP_EVENT_MEAN = 7000; //ms
    private static final int LOW_APP_EVENT_MEAN = 6000; //ms
    private static final int LOW_MED_APP_EVENT_MEAN = 5000; //ms
    private static final int MED_APP_EVENT_MEAN = 4000; //ms
    private static final int MED_HIGH_APP_EVENT_MEAN = 3000; //ms
    private static final int HIGH_APP_EVENT_MEAN = 2000;
    private static final int VERY_HIGH_APP_EVENT_MEAN = 1000; //ms

    public void onCreate(Bundle savedInstanceState)
    {
        setContentView(R.layout.app_sim);

        //TextViews
        mResultsTextView = (TextView)findViewById(R.id.app_sim_resultsview);

        //Spinners
        mAppSpinner = (Spinner)findViewById(R.id.app_sim_appspinner);
        mAlgSpinner = (Spinner)findViewById(R.id.app_sim_algspinner);

        mFreqModes = AppMonitor.DVFSController.getFrequencyScaleModes();

        mAppSpinner.setAdapter(new                                ArrayAdapter<String>(this,
R.layout.spinner_item_view, APPMODES));
        mAlgSpinner.setAdapter(new                                ArrayAdapter<String>(this,
R.layout.spinner_item_view, ALGMODES));
        mAppSpinner.setSelection(0);
        mAlgSpinner.setSelection(0);
        mCurrentAppMode = APPMODES[0];
        mCurrentAlgMode = ALGMODES[0];

        mAppSpinner.setOnItemClickListener(mOnAppModeSelectedListener);
        mAlgSpinner.setOnItemClickListener(mOnAlgModeSelectedListener);

        //Toggle Button

```

```

        mSimButton = (ToggleButton) findViewById(R.id.app_sim_simulatebutton);
        mSimButton.setOnCheckedChangeListener (mOnSimButtonChecked);

        //Check Box
        mTouchCheckBox =
(CheckBox) findViewById(R.id.app_sim_touchevents_checkbox);
        mKeyCheckBox = (CheckBox) findViewById(R.id.app_sim_keyevents_checkbox);

        //Edit Text
        mTouchEditText =
(EditText) findViewById(R.id.app_sim_touchevents_time_textbox);
        mKeyEditText =
(EditText) findViewById(R.id.app_sim_simulatekey_time_textbox);
        mAverageKeyTime = 5000;
        mAverageTouchTime = 5000;

        //Simulation
        mSimTouchEvent = new Intent();
        mSimTouchEvent.setAction(AppMonitor.GLOBAL_TOUCH_EVENT);
        mSimKeyEvent = new Intent();
        mSimKeyEvent.setAction(AppMonitor.GLOBAL_KEY_EVENT);
        super.onCreate(savedInstanceState);
    }

    private OnCheckedChangeListener mOnSimButtonChecked = new
OnCheckedChangeListener()
    {
        public void onCheckedChanged(CompoundButton buttonView, boolean
isChecked)
        {
            if (isChecked)
            {
                int defaultEventTime = getDefaultEventTime();
                String temp = mTouchEditText.getText().toString();
                Integer iTemp = Integer.parseInt(temp);
                if (iTemp == null)
                {
                    mAverageTouchTime = defaultEventTime;

mTouchEditText.setText(Integer.toString(defaultEventTime));
                }
                else
                {
                    mAverageTouchTime = iTemp;
                }
                temp = mKeyEditText.getText().toString();
                iTemp = Integer.parseInt(temp);
                if (iTemp == null)
                {
                    mAverageKeyTime = defaultEventTime;

mKeyEditText.setText(Integer.toString(defaultEventTime));
                }
                else
                {
                    mAverageKeyTime = iTemp;
                }
                resetTestThread();
                mRunTestThread.start();
            }
            else
            {
                mRunTestThread.interrupt();
            }
        }
    }
}

```



```

        mRunTestThread.stop(new ThreadDeath());
        Intent brightnessIntent = new Intent();

        brightnessIntent.setAction(AppMonitor.SCREENBRIGHTNESS_CHANGE_EVENT);
        brightnessIntent.putExtra("brightness",
getDefaultScreenBrightness());
        sendBroadcast(brightnessIntent);
    }
}

};

private int getDefaultEventTime()
{
    if (mCurrentAppMode.equals(APPMODES[0]))
    {
        return VERY_LOW_APP_EVENT_MEAN;
    }
    if (mCurrentAppMode.equals(APPMODES[1]))
    {
        return LOW_APP_EVENT_MEAN;
    }
    else if (mCurrentAppMode.equals(APPMODES[2]))
    {
        return LOW_MED_APP_EVENT_MEAN;
    }
    else if (mCurrentAppMode.equals(APPMODES[3]))
    {
        return MED_APP_EVENT_MEAN;
    }
    else if (mCurrentAppMode.equals(APPMODES[4]))
    {
        return MED_HIGH_APP_EVENT_MEAN;
    }
    else if (mCurrentAppMode.equals(APPMODES[5]))
    {
        return HIGH_APP_EVENT_MEAN;
    }
    else
    {
        return VERY_HIGH_APP_EVENT_MEAN;
    }
}

private OnItemSelectedListener mOnAlgModeSelectedListener = new
OnItemSelectedListener()
{
    public void onItemSelected(AdapterView<?> arg0, View arg1, int idx, long
arg3)
    {
        if (idx >= 0)
        {
            mCurrentAlgMode = ALGMODES[idx];
        }
        else
        {
            mCurrentAlgMode = ALGMODES[0];
            mAlgSpinner.setSelection(0);
        }
    }

    public void onNothingSelected(AdapterView<?> arg0)
    {
        mCurrentAlgMode = ALGMODES[0];
    }
}

```

```

        mAlgSpinner.setSelection(0);
    }
};

private OnItemSelectedListener mOnAppModeSelectedListener = new
OnItemSelectedListener()
{
    public void onItemSelected(AdapterView<?> arg0, View arg1, int idx, long
arg3)
    {
        if (idx >= 0)
        {
            mCurrentAppMode = APPMODES[idx];
        }
        else
        {
            mCurrentAppMode = APPMODES[0];
            mAppSpinner.setSelection(0);
        }
        mKeyEditText.setText(Integer.toString(getDefaultEventTime()));
        mTouchEditText.setText(Integer.toString(getDefaultEventTime()));
    }

    public void onNothingSelected(AdapterView<?> arg0)
    {
        mCurrentAppMode = APPMODES[0];
        mAppSpinner.setSelection(0);
    }
};

private Handler handler = new Handler()
{
    @Override
    public void handleMessage(Message msg)
    {
        float time = msg.what / 1000F;
        String status = String.format("Run Time: %.2f s\nTouch Event
Count: %d (%.1f s Avg)\nKey Event Count: %d (%.1f s Avg)", time, msg.arg1, time /
msg.arg1, msg.arg2, time / msg.arg2);
        mResultsTextView.setText(status);
    }
};

private void updateStatusText(String status)
{
    mResultsTextView.setText(status);
}

private void resetTestThread()
{
    mRunTestThread = new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            try
            {
                simulateApp();
            }
            catch (InterruptedException e) { }
        }
    });
}
}

```

```

private void simulateApp() throws InterruptedException
{
    Message msg;
    int numOfTouchEvent = 0;
    int numOfKeyEvent = 0;
    int fireTouchEvent = 0;
    int fireKeyEvent = 0;
    float movingRanMultFactor = 1F;
    Random touchRan = new Random(SystemClock.currentThreadTimeMillis());
    Random keyRan = new Random(SystemClock.elapsedRealtime());
    int time = 0;
    long sleepTime = 500;
    boolean highBurstState = true;
    int burstCount = 0;
    while (true)
    {
        if (mCurrentAlgMode == ALGMODES[0])//Constant
        {
            fireTouchEvent += 500;
            fireKeyEvent += 500;
            if (mTouchCheckBox.isChecked() && fireTouchEvent >=
mAverageTouchTime)
            {
                simulateTouchEvent();
                fireTouchEvent = 0;
                numOfTouchEvent++;
            }
            if (mKeyCheckBox.isChecked() && fireKeyEvent >=
mAverageKeyTime)
            {
                simulateKeyEvent();
                fireKeyEvent = 0;
                numOfKeyEvent++;
            }
        }
        else if (mCurrentAlgMode == ALGMODES[1]) //Stochastic
        {
            fireTouchEvent += (int) (Math.abs(touchRan.nextGaussian()) *
mAverageTouchTime * 800 / mAverageTouchTime);
            fireKeyEvent += (int) (Math.abs(keyRan.nextGaussian()) *
mAverageKeyTime * 800 / mAverageKeyTime);
            if (mTouchCheckBox.isChecked() && fireTouchEvent >=
(mAverageTouchTime))
            {
                simulateTouchEvent();
                fireTouchEvent = 0;
                numOfTouchEvent++;
            }
            if (mKeyCheckBox.isChecked() && fireKeyEvent >=
(mAverageKeyTime))
            {
                simulateKeyEvent();
                fireKeyEvent = 0;
                numOfKeyEvent++;
            }
        }
        else if (mCurrentAlgMode == ALGMODES[2]) //Moving stochastic
        {
            fireTouchEvent += (int) (Math.abs(touchRan.nextGaussian()) *
mAverageTouchTime * 800 / mAverageTouchTime);
            fireKeyEvent += (int) (Math.abs(keyRan.nextGaussian()) *
mAverageKeyTime * 800 / mAverageKeyTime);

```

```

        if (mTouchCheckBox.isChecked() && fireTouchEvent >=
(mAverageTouchTime * movingRanMultFactor))
        {
            simulateTouchEvent();
            fireTouchEvent = 0;
            numOfTouchEvent++;
        }
        if (mKeyCheckBox.isChecked() && fireKeyEvent >=
(mAverageKeyTime * movingRanMultFactor))
        {
            simulateKeyEvent();
            fireKeyEvent = 0;
            numOfKeyEvents++;
        }
        if (time % 3000 == 0) //Update moving every 3 seconds
        {
            if (movingRanMultFactor >= 2F)
            {
                movingRanMultFactor = 0.1F;
            }
            else
            {
                movingRanMultFactor += 0.1F;
            }
        }
    }
    else if (mCurrentAlgMode == ALGMODES[3])//Random
    {
        fireTouchEvent += touchRan.nextInt(mAverageTouchTime);
        fireKeyEvent += keyRan.nextInt(mAverageKeyTime);
        if (mTouchCheckBox.isChecked() && fireTouchEvent >=
mAverageTouchTime / 2)
        {
            simulateTouchEvent();
            fireTouchEvent = 0;
            numOfTouchEvent++;
        }
        if (mKeyCheckBox.isChecked() && fireKeyEvent >=
mAverageKeyTime / 2)
        {
            simulateKeyEvent();
            fireKeyEvent = 0;
            numOfKeyEvents++;
        }
        sleepTime = Math.max(1, touchRan.nextInt(mAverageTouchTime
* 2));
    }
    else //High Burst Long Pause
    {
        if (highBurstState)
        {
            if (burstCount++ > 20)
            {
                burstCount = 0;
                highBurstState = false;
                sleepTime = 10000;
            }
            else
            {
                if (mTouchCheckBox.isChecked())
                {
                    simulateTouchEvent();
                    numOfTouchEvent++;
                }
            }
        }
    }
}

```

```

        }
        if (mKeyCheckBox.isChecked())
        {
            simulateKeyEvent();
            numOfKeyEvents++;
        }
    }
}
else
{
    highBurstState = true;
    sleepTime = 200;
}
}
Thread.sleep(sleepTime);
msg = Message.obtain();
time += sleepTime;
msg.what = time;
msg.arg1 = numOfTouchEvent;
msg.arg2 = numOfKeyEvents;
handler.sendMessage(msg);
}
}

private float getDefaultScreenBrightness()
{
    float temp;
    try
    {
        int brightness = android.provider.Settings.System.getFloat(getContentResolver(),
            android.provider.Settings.System.SCREEN_BRIGHTNESS);
        temp = brightness / 255F;
        if (temp < 0.4F)
        {
            temp = 0.4F;
        }
        return temp;
    }
    catch (SettingNotFoundException e)
    {
        return 1F;
    }
}

private void simulateTouchEvent()
{
    sendBroadcast(mSimTouchEvent);
}

private void simulateKeyEvent()
{
    sendBroadcast(mSimKeyEvent);
}
}

```

### A.3 AppUtil.java (Strategy 1)

```

package csu.research.AURA;

import java.util.List;

```

```

public class AppUtil
{
    /**
     * Returns probability of x within a normal probability distribution function
     of mean/standard deviation<br>
     * within +- 3 standard deviations - N(u, s)
     * @param x - Value
     * @param mean - Mean of normal pdf
     * @param stdDev - Standard deviation of normal pdf
     * @param res - Resolution of discrete step size
     * @return Probability of x within a normal distribution given the mean and
     standard deviation
     * @throws IllegalArgumentException If standard deviation is <= 0 or resolution
     is <= 0
     */
    public static float getNormProb(float x, float mean, float stdDev, int res)
    throws IllegalArgumentException
    {
        if (stdDev <= 0)
        {
            throw new IllegalArgumentException("Standard deviation must be
greater than 0!");
        }
        else if (res <= 0)
        {
            throw new IllegalArgumentException("Resolution must be greater
than 0!");
        }
        float maxX = mean + 3 * stdDev;
        float minX = mean - 3 * stdDev;
        double C = (1 / (Math.sqrt(2 * Math.PI) * stdDev));
        double prob = 0F;
        double y;
        double incd = (maxX - minX) / res;
        for (double d = minX; d < x; d += incd)
        {
            y = -1F * Math.pow(d - mean, 2) / (2 * Math.pow(stdDev, 2));
            prob += (C * Math.exp(y)) * incd;
        }
        return (float) prob;
    }

    /**
     *
     * @param x - Data point
     * @param lowerBound - Lower bound of uniform interval
     * @param upperBound - Upper bound of uniform interval
     * @return Probability of x within the uniform given interval
     * @throws IllegalArgumentException If lower bound is not < upper bound
     */
    public static float getUniProb(float x, float lowerBound, float upperBound)
    throws IllegalArgumentException
    {
        if (lowerBound >= upperBound)
        {
            throw new IllegalArgumentException("Lower bound must be less than
upper bound!");
        }
        if (x >= lowerBound && x <= upperBound)
        {
            return x / (upperBound - lowerBound);
        }
    }
}

```

```

        return 0F;
    }

    /**
     * Returns the relationship between two random variables (data sets).  If the
     occurrence of X makes
     * Y more likely to occur, then the covariance is positive; it is negative if
     X's occurrence makes Y
     * less likely to happen and is 0 if there is no dependence.
     * @param x - Random 1 Variable Data Set
     * @param y - Random 2 Variable Data Set
     * @return Covariance of X/Y
     * @throws IllegalArgumentException If list sizes are not the same or empty
     */
    public static float getCovariance(List<Float> x, List<Float> y) throws
    IllegalArgumentException
    {
        if (x == null || y == null || x.size() == 0 || y.size() == 0)
        {
            throw new IllegalArgumentException("List sizes must be greater
than 0!");
        }
        else if (x.size() != y.size())
        {
            throw new IllegalArgumentException("Lists must be the same size to
calculate covariance!");
        }
        int size = x.size();
        float xMean, yMean, xyMean, xSum = 0, ySum = 0, xySum = 0;
        for (int i = 0; i < size; i++)
        {
            xSum += x.get(i);
            ySum += y.get(i);
            xySum += x.get(i) * y.get(i);
        }
        xMean = xSum / size;
        yMean = ySum / size;
        xyMean = xySum / size;
        return xyMean - (xMean * yMean);
    }

    /**
     * Calculates mean of data set
     * @param set - Set of values to calculate mean
     * @return Mean (Average of values)
     * @throws IllegalArgumentException If set is empty
     */
    public static float getMean(List<Float> set) throws IllegalArgumentException
    {
        if (set == null || set.size() <= 0)
        {
            throw new IllegalArgumentException("List size must be greater than
0!");
        }
        float sum = 0;
        for (int i = 0; i < set.size(); i++)
        {
            sum += set.get(i);
        }
        return sum / set.size();
    }

    /**

```

```

    * Calculates standard deviation of data set
    * @param set - Set of values to calculate standard deviation
    * @return Standard deviation
    * @throws IllegalArgumentException If set is empty
    */
    public static float getStandardDeviation(List<Float> set) throws
    IllegalArgumentException
    {
        float mean = getMean(set);
        float sum = 0;
        for (int i = 0; i < set.size(); i++)
        {
            sum += Math.pow(set.get(i) - mean, 2);
        }
        return (float) Math.sqrt(sum / set.size());
    }

    /**
    * Calculates median of data set
    * @param set - Set of values to calculate median
    * @return Median of values
    * @throws IllegalArgumentException If set is empty
    */
    public static float getMedian(List<Float> set) throws IllegalArgumentException
    {
        if (set == null || set.size() <= 0)
        {
            throw new IllegalArgumentException("List size must be greater than
0!");
        }
        if (set.size() > 1 && (set.size() % 2) == 0) //Even number of points
        {
            java.util.Collections.sort(set);
            float first = set.get(set.size() / 2 - 1);
            float second = set.get(set.size() / 2);
            return (first + second) / 2;
        }
        else if (set.size() > 1) //Odd number of points
        {
            java.util.Collections.sort(set);
            return set.get(set.size() / 2);
        }
        else //Size == 1
        {
            return set.get(0);
        }
    }

    public static float saturate(float value, float min, float max)
    {
        if (value < min)
        {
            return min;
        }
        else if (value > max)
        {
            return max;
        }
        else
        {
            return value;
        }
    }
}

```



```
}
```

## A.4 AURA.java (Strategy 1)

```
package csu.research.AURA;

import java.io.File;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import android.app.Activity;
import android.app.ActivityManager;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.Environment;
import android.os.IBinder;
import android.os.SystemClock;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.KeyEvent;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.AdapterView.AdapterContextMenuInfo;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.AdapterView.OnItemLongClickListener;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.RadioButton;
import android.widget.TextView;
import android.widget.Toast;

@SuppressWarnings("unused")
public class AURA extends Activity
{
    private final int OPTIONS_GROUP_DEFAULT = 0;
    private final int OPTIONS_GROUP_SERVICE = 1;
    private final int OPTIONS_GROUP_TEST = 2;
    private final int OPTIONS_MENU_SAVE = 1;
    private final int OPTIONS_MENU_EXPORT = 2;
    private final int OPTIONS_MENU_STOP = 3;
    private final int OPTIONS_MENU_START = 4;
    private final int OPTIONS_MENU_BACKUP = 5;
    private final int OPTIONS_MENU_RUNTEST = 6;
    private final int OPTIONS_MENU_RUNSIM = 7;
    private final int OPTIONS_MENU_SHOWSTATS = 8;
    private boolean mEnableNotifications;
    private Button mCloseServiceButton;
    private Button mSaveDatabaseButton;
    private Button mExportDatabaseButton;
```

```

private ListView mAppListView;
private RadioButton mTrainedAppsRadioButton;
private RadioButton mUntrainedAppsRadioButton;
private RadioButton mIgnoredAppsRadioButton;
private TextView mClassificationTextView;
private TextView mTouchStatsTextView;
private TextView mKeyStatsTextView;
private TextView mAlgorithmTextView;

@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mEnableNotifications = true;
    initializeView(savedInstanceState);
}

@Override
protected void onPause()
{
    super.onPause();
}

public boolean onKeyDown(int keyCode, KeyEvent event)
{
    return super.onKeyDown(keyCode, event);
}

@Override
protected void onResume()
{
    updateListView();
    super.onResume();
}

@Override
protected void onDestroy()
{
    super.onDestroy();
}

@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo
menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    if (v.getId() == R.id.AppListView)
    {
        inflater.inflate(R.menu.listview_context_menu, menu);
    }
    else if (v.getId() == R.id.AlgorithmTextView)
    {
        inflater.inflate(R.menu.algorithm_context_menu, menu);
    }
}

@Override
public boolean onContextItemSelected(Menu.Item item)
{
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId())

```

```

{
    case R.id.IgnoreAppMenuItem:
        ignoreApp(info.position);
        return true;
    case R.id.ViewAppStatsMenuItem:
        viewXYChart(info.position);
        return true;
    case R.id.ViewAppSensorInfoMenuItem:
        String appName = "";
        if (mTrainedAppsRadioButton.isChecked())
        {
            appName
AppMonitor.AppDatabase.getTrainedApps().get(info.position).getName();
        }
        else if (mUntrainedAppsRadioButton.isChecked())
        {
            appName
AppMonitor.AppDatabase.getUntrainedApps().get(info.position).getName();
        }
        else if (mIgnoredAppsRadioButton.isChecked())
        {
            appName
AppMonitor.AppDatabase.getIgnoredAppNames().get(info.position);
        }
        ApplicationInfo appInfo
AppMonitor.AppDatabase.getApplicationInfo(appName);
        return true;
    case R.id.MarkAppMenuItem:
        markAppAsTrained(info.position);
        return true;
    case R.id.UnmarkAppMenuItem:
        markAppAsUntrained(info.position);
        return true;
    case R.id.DeleteAppMenuItem:
        deleteApp(info.position);
        return true;
    case R.id.ChangeAppSettingsMenuItem:
        changeAppSettings(info.position);
        return true;
    case R.id.ResetQTableMenuItem:
        resetQTable(info.position);
        return true;
    case R.id.PowerSaverMenuItem:
        changeAlgorithm(ControlAlgorithm.POWERSAVER);
        return true;
    case R.id.ChangeBlindnessMenuItem:
        changeAlgorithm(ControlAlgorithm.CHANGE_BLINDNESS);
        return true;
    case R.id.QLearningMenuItem:
        changeAlgorithm(ControlAlgorithm.Q_LEARNING);
        return true;
    case R.id.AdaptMDPMenuItem:
        changeAlgorithm(ControlAlgorithm.NORMAL_MDP_ADAPT);
        return true;
    case R.id.NormalMDPMenuItem:
        changeAlgorithm(ControlAlgorithm.NORMAL_MDP);
        return true;
    case R.id.MovingAverageMenuItem:
        changeAlgorithm(ControlAlgorithm.MOVING_AVERAGE);
        return true;
    default:
        return super.onContextItemSelected(item);
}

```

```

    }

    private void changeAlgorithm(ControlAlgorithm algorithm)
    {
        if (!isAppMonitorServiceRunning())
        {
            Toast.makeText(getApplicationContext(), "App Service not running!",
Toast.LENGTH_LONG).show();
        }
        else
        {
            AppMonitor.Algorithm = algorithm;
            Toast.makeText(getApplicationContext(), String.format("Algorithm changed:
%s", AppMonitor.Algorithm), Toast.LENGTH_LONG).show();

            mAlgorithmTextView.setText(ControlAlgorithm.getName(AppMonitor.Algorithm));
        }
    }

    private void changeAppSettings(int position)
    {
        if (AppMonitor.AppDatabase != null && position >= 0 &&
AppMonitor.DVFSController != null && mTrainedAppsRadioButton.isChecked())
        {
            Intent intent = new Intent(this, AppSettings.class);
            intent.setAction(AppSettings.APPMON_APP_SETTINGS);

            ArrayList<Integer> freqModes =
AppMonitor.DVFSController.getFrequencyScaleModes();
            String appName =
AppMonitor.AppDatabase.getTrainedApps().get(position).getName();
            ApplicationInfo appInfo =
AppMonitor.AppDatabase.getApplicationInfo(appName);
            if (appInfo != null)
            {
                intent.putExtra("name", appName);
                intent.putExtra("classification",
AppClassification.toInt(appInfo.getAppClassification()));
                intent.putExtra("isdynamic", appInfo.isDynamic());
                intent.putExtra("touchmean",
(int)Math.round(appInfo.getTouchMean()));
                intent.putExtra("touchdev",
(int)Math.round(appInfo.getTouchStandardDeviation()));
                intent.putExtra("keymean", (int)Math.round(appInfo.getKeyMean()));
                intent.putExtra("keydev",
(int)Math.round(appInfo.getKeyStandardDeviation()));
                startActivityForResult(intent, OPTIONS_GROUP_DEFAULT);
            }
        }
    }

    public void resetQTable(int position)
    {
        if (AppMonitor.AppDatabase != null && position >= 0 &&
AppMonitor.DVFSController != null && mTrainedAppsRadioButton.isChecked())
        {
            String appName =
AppMonitor.AppDatabase.getTrainedApps().get(position).getName();
            ApplicationInfo appInfo =
AppMonitor.AppDatabase.getApplicationInfo(appName);
            if (appInfo != null)
            {
                appInfo.initQValues();
            }
        }
    }

```

```

        Toast.makeText(getApplicationContext(), "Q Table reset!",
Toast.LENGTH_LONG).show();
    }
}

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (resultCode == RESULT_OK && data != null)
    {
        String appName = data.getStringExtra("name");
        ApplicationInfo appInfo =
AppMonitor.AppDatabase.getApplicationInfo(appName);
        if (appInfo != null)
        {
            boolean isDynamic = data.getBooleanExtra("isdynamic", true);
            AppClassification classification =
AppClassification.fromInt((data.getIntExtra("classification",
AppClassification.toInt(appInfo.getAppClassification()))));
            int touchMean = data.getIntExtra("touchmean", -1);
            int touchDev = data.getIntExtra("touchdev", -1);
            int keyMean = data.getIntExtra("keymean", -1);
            int keyDev = data.getIntExtra("keydev", -1);
            appInfo.classifyApplication(classification);
            if (!isDynamic && touchMean >= 0 && touchDev >= 0 && keyMean >= 0
&& keyDev >= 0)
            {
                appInfo.setStaticStatistics(touchMean, touchDev, keyMean,
keyDev);
                appInfo.setStatisticsMode(false);
                Toast.makeText(getApplicationContext(), "Application set to
static!", Toast.LENGTH_LONG).show();
            }
            else
            {
                appInfo.setStatisticsMode(true);
                Toast.makeText(getApplicationContext(), "Application set to
dynamic!", Toast.LENGTH_LONG).show();
            }
        }
        else
        {
            Toast.makeText(getApplicationContext(), "Application settings were
not changed!", Toast.LENGTH_LONG).show();
        }
    }
}

private void viewXYChart(int position)
{
    if (AppMonitor.AppDatabase != null && !mIgnoredAppsRadioButton.isChecked())
    {
        Intent intent = new Intent(this, AppChart.class);
        LinkedList<ApplicationInfo> apps;
        ApplicationInfo appInfo;
        ArrayList<Integer> x = new ArrayList<Integer>();
        float[] y;
        //Touch Mean
        if (mTrainedAppsRadioButton.isChecked())
        {
            apps = AppMonitor.AppDatabase.getTrainedApps();

```

```

        appInfo = apps.get(position);
        y = new float[appInfo.getTouchMeanList().size()];
    }
    else
    {
        apps = AppMonitor.AppDatabase.getUntrainedApps();
        appInfo = apps.get(position);
        y = new float[appInfo.getTouchMeanList().size()];
    }
    intent.putExtra("title", appInfo.getShortName());
    intent.putExtra("meantouchname", "Mean Touch");
    intent.putExtra("meankeyname", "Mean Key");
    for (int i = 0; i < y.length; i++)
    {
        x.add(i);
        y[i] = appInfo.getTouchMeanList().get(i);
    }
    intent.putExtra("meantouchx", x);
    intent.putExtra("meantouchy", y);
    intent.putExtra("meantouch", appInfo.getTouchMean());
    //Key Mean
    if (mTrainedAppsRadioButton.isChecked())
    {
        apps = AppMonitor.AppDatabase.getTrainedApps();
        appInfo = apps.get(position);
        y = new float[appInfo.getKeyMeanList().size()];
    }
    else
    {
        apps = AppMonitor.AppDatabase.getUntrainedApps();
        appInfo = apps.get(position);
        y = new float[appInfo.getKeyMeanList().size()];
    }

    x.clear();
    for (int i = 0; i < y.length; i++)
    {
        x.add(i);
        y[i] = appInfo.getKeyMeanList().get(i);
    }
    intent.putExtra("meankeyx", x);
    intent.putExtra("meankeyy", y);
    intent.putExtra("meankey", appInfo.getKeyMean());
    startActivity(intent);
}

private boolean isAppMonitorServiceRunning()
{
    boolean appMonActive = false;
    ActivityManager am =
(ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    List<ActivityManager.RunningServiceInfo> runningServices =
am.getRunningServices(1000);
    for (int i = 0; i < runningServices.size(); i++)
    {
        if
(runningServices.get(i).service.getShortClassName().equals(".AppMonitor"))
        {
            appMonActive = true;
            break;
        }
    }
    return appMonActive;
}

```

```

    }

    private boolean startAppMonitorService()
    {
        boolean appMonActive = isAppMonitorServiceRunning();
        if (!appMonActive)
        {
            startService(new Intent(this, AppMonitor.class));
        }
        return !appMonActive;
    }

    private void initializeView(Bundle savedInstanceState)
    {
        //Radio buttons
        mTrainedAppsRadioButton =
(RadioButton) findViewById(R.id.TrainedAppsRadioButton);
        mUntrainedAppsRadioButton =
(RadioButton) findViewById(R.id.UntrainedAppsRadioButton);
        mIgnoredAppsRadioButton =
(RadioButton) findViewById(R.id.IgnoredAppsRadioButton);
        mTrainedAppsRadioButton.setOnClickListener(onRadioButtonClicked);
        mUntrainedAppsRadioButton.setOnClickListener(onRadioButtonClicked);
        mIgnoredAppsRadioButton.setOnClickListener(onRadioButtonClicked);

        //List views
        mAppListView = (ListView) findViewById(R.id.AppListView);
        mAppListView.setOnItemClickListener(mAppListViewClicked);

        //Text views
        mClassificationTextView = (TextView) findViewById(R.id.ClassificationTextView);
        mTouchStatsTextView = (TextView) findViewById(R.id.TouchTextView);
        mKeyStatsTextView = (TextView) findViewById(R.id.KeyTextView);
        mAlgorithmTextView = (TextView) findViewById(R.id.AlgorithmTextView);

        //Menus
        registerForContextMenu(mAppListView);
        registerForContextMenu(mAlgorithmTextView);

        //Update View
        updateListView();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        menu.add(OPTIONS_GROUP_DEFAULT, OPTIONS_MENU_SAVE, 0, "Save Database");
        menu.getItem(0).setIcon(R.drawable.save_icon);
        menu.add(OPTIONS_GROUP_DEFAULT, OPTIONS_MENU_EXPORT, 0, "Export Database");
        menu.getItem(1).setIcon(R.drawable.export_icon);
        menu.add(OPTIONS_GROUP_DEFAULT, OPTIONS_MENU_BACKUP, 0, "Backup Database");
        menu.getItem(2).setIcon(R.drawable.backup_icon);
        menu.add(OPTIONS_GROUP_SERVICE, OPTIONS_MENU_START, 0, "Start Monitoring");
        menu.getItem(3).setIcon(R.drawable.start_icon);
        menu.add(OPTIONS_GROUP_SERVICE, OPTIONS_MENU_STOP, 0, "Stop Monitoring");
        menu.getItem(4).setIcon(R.drawable.stop_icon);
        menu.add(OPTIONS_GROUP_TEST, OPTIONS_MENU_RUNTEST, 0, "Run Test Module");
        menu.getItem(5).setIcon(R.drawable.test_icon);
        menu.add(OPTIONS_GROUP_TEST, OPTIONS_MENU_RUNSIM, 0, "App Simulator");
        menu.getItem(6).setIcon(R.drawable.sim_icon);
        menu.add(OPTIONS_GROUP_TEST, OPTIONS_MENU_SHOWSTATS, 0, "Show Recent Stats");
        menu.getItem(7).setIcon(R.drawable.sim_icon);
    }

```

```

        return super.onCreateOptionsMenu(menu);
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu)
    {
        menu.setGroupVisible(OPTIONS_GROUP_DEFAULT, true);
        menu.setGroupVisible(OPTIONS_GROUP_SERVICE, true);
        menu.setGroupVisible(OPTIONS_GROUP_TEST, isAppMonitorServiceRunning());
        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        switch (item.getItemId())
        {
            {
            case OPTIONS_MENU_SAVE:
                if (AppMonitor.AppDatabase != null && saveDatabase())
                {
                    Toast.makeText(getApplicationContext(), "Current App Database Saved!",
Toast.LENGTH_LONG).show();
                }
                else
                {
                    Toast.makeText(getApplicationContext(), "Error Occurred Saving
Database!", Toast.LENGTH_LONG).show();
                }
                return true;
            case OPTIONS_MENU_EXPORT:
                if (AppMonitor.AppDatabase != null && exportDatabase())
                {
                    Toast.makeText(getApplicationContext(), "Current App Database
Exported!", Toast.LENGTH_LONG).show();
                }
                else
                {
                    Toast.makeText(getApplicationContext(), "Error Occurred Exporting
Database!", Toast.LENGTH_LONG).show();
                }
                return true;
            case OPTIONS_MENU_STOP:
                if (stopAppMonitorService())
                {
                    Toast.makeText(getApplicationContext(), "App Monitor Service
Stopped!", Toast.LENGTH_LONG).show();
                }
                else
                {
                    Toast.makeText(getApplicationContext(), "App Monitor Service Is Not
Running!", Toast.LENGTH_LONG).show();
                }
                return true;
            case OPTIONS_MENU_START:
                if (startAppMonitorService())
                {
                    Toast.makeText(getApplicationContext(), "App Monitor Service
Started!", Toast.LENGTH_LONG).show();
                }
                else
                {
                    Toast.makeText(getApplicationContext(), "App Monitor Service Is
Already Running!", Toast.LENGTH_LONG).show();
                }
            }
        }
    }

```



```

        }
        return true;
    case OPTIONS_MENU_BACKUP:
        if (backupDatabase())
        {
            Toast.makeText(getApplicationContext(), "Current App Database Backed
Up!", Toast.LENGTH_LONG).show();
        }
        else
        {
            Toast.makeText(getApplicationContext(), "Error Occurred Backing Up
Database!", Toast.LENGTH_LONG).show();
        }
        return true;
    case OPTIONS_MENU_RUNTEST:
        startActivity(new Intent(this, DVFSSTest.class));
        return true;
    case OPTIONS_MENU_RUNSIM:
        startActivity(new Intent(this, AppSimulator.class));
        return true;
    case OPTIONS_MENU_SHOWSTATS:
        if (AppMonitor.LastStatsToast != null)
        {
            AppMonitor.LastStatsToast.show();
        }
        else
        {
            Toast.makeText(getApplicationContext(), "Stats N/A",
Toast.LENGTH_LONG).show();
        }
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private OnItemClickListener mAppListViewClicked = new OnItemClickListener()
{
    @Override
    public void onItemClick(AdapterView<?> adapView, View view, int position,
long id)
    {
        if (position > ListView.INVALID_POSITION && AppMonitor.AppDatabase !=
null)
        {
            String appName = (String) adapView.getItemAtPosition(position);
            ApplicationInfo appInfo =
(ApplicationInfo) AppMonitor.AppDatabase.getApplicationInfo(appName);
            if (appInfo.getAppClassification() != null)

mClassificationTextView.setText(appInfo.getAppClassification().toString());
            mTouchStatsTextView.setText(String.format("Touch: %.1f +- %.1f s",
appInfo.getTouchMean() / 1000, appInfo.getTouchStandardDeviation() / 1000));
            mKeyStatsTextView.setText(String.format("Key: %.1f +- %.1f s",
appInfo.getKeyMean() / 1000, appInfo.getKeyStandardDeviation() / 1000));
        }
        else
        {
            mClassificationTextView.setText(R.string.app_classification);
            mTouchStatsTextView.setText(R.string.touch_stats);
            mKeyStatsTextView.setText(R.string.key_stats);
        }
    }
};

```

```

private OnClickListener onRadioButtonClicked = new OnClickListener()
{
    public void onClick(View v)
    {
        if (AppMonitor.AppDatabase != null)
        {
            ArrayAdapter<String> adapter;
            LinkedList<ApplicationInfo> appInfo;
            int size = 1;
            String[] apps = new String[]{" "};
            if (v.getId() == R.id.TrainedAppsRadioButton)
            {
                size =
AppMonitor.AppDatabase.getNumberOfTrainedApps();
                if (size > 0)
                {
                    appInfo =
AppMonitor.AppDatabase.getTrainedApps();
                    apps = new String[size];
                    for (int i = 0; i < size; i++)
                    {
                        apps[i] = appInfo.get(i).getName();
                    }
                }
            }
            else if (v.getId() == R.id.UntrainedAppsRadioButton)
            {
                size =
AppMonitor.AppDatabase.getNumberOfUntrainedApps();
                if (size > 0)
                {
                    appInfo =
AppMonitor.AppDatabase.getUntrainedApps();
                    apps = new String[size];
                    for (int i = 0; i < size; i++)
                    {
                        apps[i] = appInfo.get(i).getName();
                    }
                }
            }
            else if (v.getId() == R.id.IgnoredAppsRadioButton)
            {
                size =
AppMonitor.AppDatabase.getNumberOfIgnoredApps();
                if (size > 0)
                {
                    apps = new String[size];
                    for (int i = 0; i < size; i++)
                    {
                        apps[i] =
AppMonitor.AppDatabase.getIgnoredAppNames().get(i);
                    }
                }
            }
            adapter = new ArrayAdapter<String>(AURA.this,
R.layout.list_item_view, apps);
            mAppListView.setAdapter(adapter);
        }
        else
        {
            Toast.makeText(getApplicationContext(), "Service is not
available!", Toast.LENGTH_LONG).show();
        }
    }
}

```

```

        }
    }
};

private boolean stopAppMonitorService()
{
    return stopService(new Intent(this, AppMonitor.class));
}

private void ignoreApp(int position)
{
    if (AppMonitor.AppDatabase != null)
    {
        String appName;
        if (mTrainedAppsRadioButton.isChecked())
        {
            appName =
AppMonitor.AppDatabase.getTrainedApps().get(position).getName();
            if (AppMonitor.AppDatabase.addIgnoredApp(appName))
            {
                Toast.makeText(getApplicationContext(), "Application will
be ignored!", Toast.LENGTH_LONG).show();
            }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
ignoring application!", Toast.LENGTH_LONG).show();
            }
        }
        else if (mUntrainedAppsRadioButton.isChecked())
        {
            appName =
AppMonitor.AppDatabase.getUntrainedApps().get(position).getName();
            if (AppMonitor.AppDatabase.addIgnoredApp(appName))
            {
                Toast.makeText(getApplicationContext(), "Application will
be ignored!", Toast.LENGTH_LONG).show();
            }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
ignoring application!", Toast.LENGTH_LONG).show();
            }
        }
        updateListView();
    }
}

private boolean saveDatabase()
{
    try
    {
        if (AppMonitor.AppDatabase != null &&
Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED))
        {
            String filePath = String.format("%s/AppMonitor",
Environment.getExternalStorageDirectory());
            if (!new File(filePath).exists())
            {
                new File(filePath).mkdirs();
            }
            if (new File(filePath).canWrite())
            {

```

```

        return AppMonitor.AppDatabase.save(filePath);
    }
}
return false;
}
catch (Exception e)
{
    return false;
}
}

private boolean backupDatabase()
{
    try
    {
        if (AppMonitor.AppDatabase != null &&
Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED))
        {
            String filePath = String.format("%s/AppMonitor/Backup",
Environment.getExternalStorageDirectory());
            if (!new File(filePath).exists())
            {
                new File(filePath).mkdirs();
            }
            if (new File(filePath).canWrite())
            {
                return AppMonitor.AppDatabase.backup(filePath);
            }
        }
        return false;
    }
    catch (Exception e)
    {
        return false;
    }
}

private boolean exportDatabase()
{
    try
    {
        if (AppMonitor.AppDatabase != null &&
Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED))
        {
            String filePath = String.format("%s/AppMonitor",
Environment.getExternalStorageDirectory());
            if (!new File(filePath).exists())
            {
                new File(filePath).mkdirs();
            }
            if (new File(filePath).canWrite())
            {
                return
AppMonitor.AppDatabase.exportDatabase(String.format("%s/AppDatabase.txt", filePath));
            }
        }
        return false;
    }
    catch (Exception e)
    {
        return false;
    }
}
}

```

```

private void deleteApp(int position)
{
    if (AppMonitor.AppDatabase != null)
    {
        String appName;
        if (mUntrainedAppsRadioButton.isChecked())
        {
            appName =
AppMonitor.AppDatabase.getUntrainedApps().get(position).getName();
            if
(AppMonitor.AppDatabase.removeUntrainedApp(AppMonitor.AppDatabase.getApplicationInfo(a
ppName)))
            {
                Toast.makeText(getApplicationContext(), "Application was
removed!", Toast.LENGTH_LONG).show();
            }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
removing application!", Toast.LENGTH_LONG).show();
            }
        }
        else if (mTrainedAppsRadioButton.isChecked())
        {
            appName =
AppMonitor.AppDatabase.getTrainedApps().get(position).getName();
            if
(AppMonitor.AppDatabase.removeTrainedApp(AppMonitor.AppDatabase.getApplicationInfo(app
Name)))
            {
                Toast.makeText(getApplicationContext(), "Application was
removed!", Toast.LENGTH_LONG).show();
            }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
removing application!", Toast.LENGTH_LONG).show();
            }
        }
        else if (mIgnoredAppsRadioButton.isChecked())
        {
            appName =
AppMonitor.AppDatabase.getIgnoredAppNames().get(position);
            if (AppMonitor.AppDatabase.removeIgnoredApp(appName))
            {
                Toast.makeText(getApplicationContext(), "Application was
removed!", Toast.LENGTH_LONG).show();
            }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
removing application!", Toast.LENGTH_LONG).show();
            }
        }
        updateListView();
    }
}

private void markAppAsTrained(int position)
{
    if (AppMonitor.AppDatabase != null)
    {

```

```

        String appName;
        if (mUntrainedAppsRadioButton.isChecked())
        {
            appName
AppMonitor.AppDatabase.getUntrainedApps().get(position).getName();
            if
(AppMonitor.AppDatabase.markAppAsTrained(AppMonitor.AppDatabase.getApplicationInfo(app
Name)))
                {
                    Toast.makeText(getApplicationContext(), "Application is
marked!", Toast.LENGTH_LONG).show();
                }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
marking application!", Toast.LENGTH_LONG).show();
            }
            updateListView();
        }
    }
}

private void markAppAsUntrained(int position)
{
    if (AppMonitor.AppDatabase != null)
    {
        String appName;
        if (mTrainedAppsRadioButton.isChecked())
        {
            appName
AppMonitor.AppDatabase.getTrainedApps().get(position).getName();
            if
(AppMonitor.AppDatabase.markAppAsUntrained(AppMonitor.AppDatabase.getApplicationInfo(a
ppName)))
                {
                    Toast.makeText(getApplicationContext(), "Application is
unmarked!", Toast.LENGTH_LONG).show();
                }
            else
            {
                Toast.makeText(getApplicationContext(), "Error occurred
unmarking application!", Toast.LENGTH_LONG).show();
            }
        }
        updateListView();
    }
}

private void updateListView()
{
    if (AppMonitor.AppDatabase != null)
    {
        ArrayAdapter<String> adapter;
        LinkedList<ApplicationInfo> appInfo;
        int size = 1;
        String[] apps = new String[]{" "};
        if (mTrainedAppsRadioButton.isChecked())
        {
            size = AppMonitor.AppDatabase.getNumberOfTrainedApps();
            if (size > 0)
            {
                appInfo = AppMonitor.AppDatabase.getTrainedApps();
                apps = new String[size];
            }
        }
    }
}

```

```

        for (int i = 0; i < size; i++)
        {
            apps[i] = appInfo.get(i).getName();
        }
    }
}
else if (mUntrainedAppsRadioButton.isChecked())
{
    size = AppMonitor.AppDatabase.getNumberOfUntrainedApps();
    if (size > 0)
    {
        appInfo = AppMonitor.AppDatabase.getUntrainedApps();
        apps = new String[size];
        for (int i = 0; i < size; i++)
        {
            apps[i] = appInfo.get(i).getName();
        }
    }
}
else if (mIgnoredAppsRadioButton.isChecked())
{
    size = AppMonitor.AppDatabase.getNumberOfIgnoredApps();
    if (size > 0)
    {
        apps = new String[size];
        for (int i = 0; i < size; i++)
        {
            apps[i] =
AppMonitor.AppDatabase.getIgnoredAppNames().get(i);
        }
    }
}
adapter = new ArrayAdapter<String>(AURA.this,
R.layout.list_item_view, apps);
mAppListView.setAdapter(adapter);
}
else
{
    Toast.makeText(getApplicationContext(), "Service is not
available!", Toast.LENGTH_LONG).show();
}
}

// private void bindToAppMonitor()
// {
//     if (!mServiceConnected && mServiceConnection != null)
//     {
//         mServiceConnected = bindService(new Intent(AppMon.this,
AppMonitor.class), mServiceConnection, Context.BIND_AUTO_CREATE);
//     }
// }
//
// private void unbindToAppMonitor()
// {
//     if (mServiceConnected && mServiceConnection != null)
//     {
//         unbindService(mServiceConnection);
//         mServiceConnected = false;
//     }
// }

// private ServiceConnection mServiceConnection = new ServiceConnection()
// {

```

```

//      @Override
//      public void onServiceConnected(ComponentName name, IBinder service)
//      {
//          mAppMonitor = ((AppMonitor.AppMonitorBinder)service).getService();
//          Toast.makeText(getApplicationContext(), "App Monitor Service
Connected!", Toast.LENGTH_SHORT).show();
//      }
//
//      @Override
//      public void onServiceDisconnected(ComponentName name)
//      {
//          mAppMonitor = null;
//          Toast.makeText(getApplicationContext(), "App Monitor Service
Disconnected!", Toast.LENGTH_SHORT).show();
//      }
//  };
}

```

## A.5 BatteryModels.java (Strategy 1)

```

package csu.research.AURA;

public class BatteryModels
{
    private static final float BATTERY_CAPACITY_MAH = 1000F;
    private static final float BATTERY_NOMINAL_VOLTAGE = 3.9F;
    private static final float INITIAL_SOC_PCT = 100;

    public static float getPercentSOCSaved(float deltaCPUPowerSaved)
    {
        if (deltaCPUPowerSaved != 0F)
        {
            return INITIAL_SOC_PCT - (INITIAL_SOC_PCT * (1 - (1 /
(BATTERY_CAPACITY_MAH * 3600)) * deltaCPUPowerSaved / BATTERY_NOMINAL_VOLTAGE));
        }
        return 0F;
    }

    public static float getPercentSOCSaved(float deltaCPUPowerSaved, float
initialSOC, float batteryVoltage)
    {
        if (batteryVoltage <= 0)
        {
            batteryVoltage = BATTERY_NOMINAL_VOLTAGE;
        }
        if (deltaCPUPowerSaved != 0F)
        {
            return initialSOC - (initialSOC * (1 - (1 / (BATTERY_CAPACITY_MAH
* 3600)) * deltaCPUPowerSaved / batteryVoltage));
        }
        return 0F;
    }
}

```

## A.6 ControlAlgorithm.java (Strategy 1)

```

package csu.research.AURA;

public enum ControlAlgorithm

```



```

{
    POWERSAVER,
    CHANGE_BLINDNESS,
    Q_LEARNING,
    NORMAL_MDP,
    NORMAL_MDP_ADAPT,
    MOVING_AVERAGE;

    public static final String[] CONTROL_ALGORITHMS = new String[] {"POWERSAVER",
"CHANGE BLINDNESS", "Q LEARNING", "NORMAL MDP", "NORMAL MDP ADAPT", "MOVING AVERAGE"};

    public static ControlAlgorithm getFromString(String algorithm)
    {
        algorithm = algorithm.replaceAll(" ", "").toLowerCase();
        if (algorithm.equals("powersaver"))
        {
            return ControlAlgorithm.POWERSAVER;
        }
        else if (algorithm.equals("change_blindness"))
        {
            return ControlAlgorithm.CHANGE_BLINDNESS;
        }
        else if (algorithm.equals("q_learning"))
        {
            return ControlAlgorithm.Q_LEARNING;
        }
        else if (algorithm.equals("normal_mdp"))
        {
            return ControlAlgorithm.NORMAL_MDP;
        }
        else if (algorithm.equals("normal_mdp_adapt"))
        {
            return ControlAlgorithm.NORMAL_MDP_ADAPT;
        }
        else
        {
            return ControlAlgorithm.MOVING_AVERAGE;
        }
    }

    public static String getName(ControlAlgorithm algorithm)
    {
        switch (algorithm)
        {
            case POWERSAVER:
                return "Power Saver";
            case CHANGE_BLINDNESS:
                return "Change Blindness";
            case Q_LEARNING:
                return "Q Learning";
            case NORMAL_MDP:
                return "Normal MDP";
            case NORMAL_MDP_ADAPT:
                return "Normal MDP Adapt";
            case MOVING_AVERAGE:
                return "Moving Average";
            default:
                return "Undefined";
        }
    }
}

```

## A.7 CPUInfo.java (Strategy 1)

```
package csu.research.AURA;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class CPUInfo
{
    private static int mNormalUserProcessTime = -1;
    private static int mNicedUserProcessTime = -1;
    private static int mSystemProcessTime = -1;
    private static int mIdleProcessTime = -1;
    private static int mIoWaitTime = -1;
    private static int mIrqTime = -1;
    private static int mSoftIRQTime = -1;
    private static int mStealTime = -1;
    private static int mGuestTime = -1;
    private static int mIdlePct = 0;
    private static int mSystemPct = 0;
    private static int mUserPct = 0;
    private static int mIrqPct = 0;
    private static int mIoWaitPct = 0;
    private static int mTotalPct = 0;
    private static int mContextSwitches = -1;
    private static int mProcessesCreated = -1;
    private static int mProcessesRunning = -1;
    private static int mProcessesBlocked = -1;

    private static String readCPUInfo() {
        FileReader fstream;
        try {
            fstream = new FileReader("/proc/stat");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return "error";
        }
        BufferedReader in = new BufferedReader(fstream, 500);
        String result = "";
        String line;
        try {
            while ((line = in.readLine()) != null) {
                result = String.format("%s%s\n", result, line);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return result;
    }

    public static int[] getCPUStats() {
        int[] stats = null;
        try {
            String info = readCPUInfo();
            if (info.length() > 0) {
                String[] lines = info.split("\n");
                String line;
                String[] buf;
                int temp, temp1, temp2, temp3, temp4, temp5, temp6;
            }
        }
    }
}
```

```

int total, totalDiff;
int deltaContextSwitches = 0;
for (int i = 0; i < lines.length; i++) {
    line = lines[i];
    if (line.startsWith("cpu ")) {
        if (line.startsWith("cpu ")) {
            buf = line.replaceAll("cpu ", "").trim().split(" ");
            if (buf.length >= 9) {
                temp = mNormalUserProcessTime;
                mNormalUserProcessTime = Integer.parseInt(buf[0]);
                temp1 = mNicedUserProcessTime;
                mNicedUserProcessTime = Integer.parseInt(buf[1]);
                temp2 = mSystemProcessTime;
                mSystemProcessTime = Integer.parseInt(buf[2]);
                temp3 = mIdleProcessTime;
                mIdleProcessTime = Integer.parseInt(buf[3]);
                temp4 = mIoWaitTime;
                mIoWaitTime = Integer.parseInt(buf[4]);
                temp5 = mIrqTime + mSoftIRQTime;
                mIrqTime = Integer.parseInt(buf[5]);
                mSoftIRQTime = Integer.parseInt(buf[6]);
                temp6 = mStealTime + mGuestTime;
                mStealTime = Integer.parseInt(buf[7]);
                mGuestTime = Integer.parseInt(buf[8]);
                total = mNormalUserProcessTime + mNicedUserProcessTime +
mSystemProcessTime + mIdleProcessTime + mIoWaitTime + mIrqTime + mSoftIRQTime +
mStealTime + mGuestTime;
                totalDiff = total - (temp + temp1 + temp2 + temp3 + temp4
+ temp5 + temp6);
                mIdlePct = ((mIdleProcessTime - temp3) * 100) / totalDiff;
                mSystemPct = ((mSystemProcessTime - temp2) * 100) /
totalDiff;
                mUserPct = ((mNormalUserProcessTime +
mNicedUserProcessTime) - (temp + temp1)) * 100) / totalDiff;
                mIrqPct = ((mIrqTime + mSoftIRQTime) - temp5) * 100) /
totalDiff;
                mIoWaitPct = ((mIoWaitTime - temp4) * 100) / totalDiff;
                mTotalPct = ((mNormalUserProcessTime +
mNicedUserProcessTime) - (temp + temp1)) +
(mSystemProcessTime - temp2) * 100) / totalDiff;
            }
        }
    }
    else if (line.startsWith("ctxt")) {
        buf = line.replaceAll("ctxt ", "").trim().split(" ");
        if (buf.length == 1) {
            temp = mContextSwitches;
            mContextSwitches = Integer.parseInt(buf[0]);
            deltaContextSwitches = mContextSwitches - temp;
        }
    }
    else if (line.startsWith("processes")) {
        buf = line.replaceAll("processes ", "").trim().split(" ");
        if (buf.length == 1) {
            mProcessesCreated = Integer.parseInt(buf[0]);
        }
    }
    else if (line.startsWith("procs_running")) {
        buf = line.replaceAll("procs_running ", "").trim().split(" ");
        if (buf.length == 1) {
            mProcessesRunning = Integer.parseInt(buf[0]);
        }
    }
    else if (line.startsWith("procs_blocked")) {
        buf = line.replaceAll("procs_blocked ", "").trim().split(" ");

```

```

        if (buf.length == 1) {
            mProcessesBlocked = Integer.parseInt(buf[0]);
        }
    }
    stats = new int[] {mIdlePct, mSystemPct, mUserPct, mIrqPct,
mIoWaitPct, mContextSwitches, deltaContextSwitches, mProcessesCreated,
mProcessesRunning, mProcessesBlocked };
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
    return stats;
}

public static int getCPUIdlePct()
{
    int[] stats = getCPUStats();
    return stats[0];
}

public static int getCPUUtilizationPct()
{
    return 100 - getCPUIdlePct();
}

public static int getIdlePct() {
    return mIdlePct;
}

public static int getSystemPct() {
    return mSystemPct;
}

public static int getUserPct() {
    return mUserPct;
}

public static int getIrqPct() {
    return mIrqPct;
}

public static int getIoWaitPct() {
    return mIoWaitPct;
}

public static int getTotalPct() {
    // total = user + system + idle + io_wait
    return mTotalPct;
}

public static int getContextSwitches() {
    return mContextSwitches;
}

public static int getProcessesCreated() {
    return mProcessesCreated;
}

public static int getProcessesRunning() {

```

```

        return mProcessesRunning;
    }

    public static int getProcessesBlocked() {
        return mProcessesBlocked;
    }
}

```

## A.8 DVFSControl.java (Strategy 1)

```

package csu.research.AURA;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import android.util.Log;

public class DVFSControl
{
    private final String      FREQ_MODES_FILE      =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies";
    private final String      SCALE_FREQ_FILE      =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed";
    private final String      MAX_FREQ_FILE        =
"/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq";
    private final String      MIN_FREQ_FILE        =
"/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq";
    private final String      SET_MAX_FREQ_FILE    =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
    private final String      SET_MIN_FREQ_FILE    =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
    private final String      GOV_FILE             =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor";
    private final String      CUR_FREQ_FILE        =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq";
    private final String      AVA_GOV_FILE         =
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors";
    // private final String[] SET_NOMINAL_FREQ_CMD = {"su","-c","echo \"528000\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};
    // private final String[] SET_LOWEST_FREQ_CMD = {"su","-c","echo \"128000\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};
    // private final String[] SET_LOWNOM_FREQ_CMD = {"su","-c","echo \"245760\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};

    // NEW FREQUENCIES FOR NEXUS ONE
    private final String[] SET_NOMINAL_FREQ_CMD = {"su","-c","echo \"729600\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};
    private final String[] SET_LOWEST_FREQ_CMD = {"su","-c","echo \"245000\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};
    private final String[] SET_LOWNOM_FREQ_CMD = {"su","-c","echo \"537600\" >
"/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed"};

    private FrequencyGovernors governor = FrequencyGovernors.Unknown;
    private boolean errorStatus = true;
    private String error = "UNKNOWN";
    private ArrayList<Integer> freqModes;
    private int minFreq = 0;
    private int maxFreq = 0;
}

```

```

public DVFSControl()
{
    freqModes = new ArrayList<Integer>();
    try
    {
        initializeMonitor();
    }
    catch (FileNotFoundException e)
    {
        error = "Could not find/access frequency modes file!";
        e.printStackTrace();
    }
    catch (IOException e)
    {
        error = "Could not read frequency modes file!";
        e.printStackTrace();
    }
    catch (NumberFormatException e)
    {
        error = "Error parsing frequency modes file (Number format
exception)!";
        e.printStackTrace();
    }
}

/**
 * Called at instantiation to find all available frequency levels, min/max
supported frequency levels,
 * and available scaling govenors
 * @throws IOException
 */
private void initializeMonitor() throws IOException
{
    readCPUMinMaxFrequencies();
    String temp = new BufferedReader(new FileReader(FREQ_MODES_FILE),
1000).readLine();
    String[] vals = temp.split(" ");
    int iTemp;
    for (int i = 0; i < vals.length; i++)
    {
        iTemp = Integer.parseInt(vals[i]);
        if (iTemp >= minFreq && iTemp <= maxFreq)
        {
            freqModes.add(iTemp);
        }
    }
    errorStatus = false;
    error = "NO ERROR";
    setFrequencyGovernor(FrequencyGovernors.Userspace);
    if (getCurrentGovernor() == FrequencyGovernors.Userspace)
    {
        setMinMaxScaleFrequencies();
    }
}

/**
 * Instantiation error description
 * @return A description of the error that occurred at instantiation
 */
public String getError()
{
    return error;
}

```

```

    }

    /**
     * Instantiation error status
     * @return
     * true if an error has occurred<br>
     * false if an error has not occurred
     */
    public boolean hasErrorOccurred()
    {
        return errorStatus;
    }

    /**
     * List of supported DVFS levels
     * @return
     * ArrayList of available frequency levels (voltage is not explicitly
    controlled)
     */
    public ArrayList<Integer> getFrequencyScaleModes()
    {
        return freqModes;
    }

    /**
     * Number of supported DVFS levels
     * @return
     * The number of supported DVFS levels (voltage is not explicitly controlled)
     */
    public int getNumberOfFrequencyScaleModes()
    {
        return freqModes.size();
    }

    /**
     * Changes the DVFS level by setting a desired frequency set-point (voltage is
    not explicitly controlled)<br>
     * This is only supported if the frequency governor is "userspace"<br><br>
     * NOTE: This will return false even if the frequency level was changed, if the
    input frequency was rounded due to incorrect DVFS level support
     * @param freqkHz Integer in units of kHz to set the desired frequency level.
    Value will be rounded to nearest DVFS level
     * @return
     * true if frequency level was successfully changed<br>
     * false if frequency level was not successfully changed
     */
    public boolean setCPUFrequency(int freqkHz)
    {
        try
        {
            if (getCurrentGovernor() == FrequencyGovernors.Userspace)
            {
                String[] cmdStr = {"su", "-c", String.format("echo \"%d\" >
%s", freqkHz, SCALE_FREQ_FILE)};
                Runtime.getRuntime().exec(cmdStr);
                return getCPUFrequency() == freqkHz;
            }
            return false;
        }
        catch (IOException e)
        {
            Log.d("SETCPUFREQ:", e.getMessage());
            error = "Set frequency IO exception!";
        }
    }

```

```

        e.printStackTrace();
        return false;
    }
    catch(Exception e)
    {
        Log.d("SETCPUFREQ*", e.getMessage());
        error = e.getMessage();
        e.printStackTrace();
        return false;
    }
}

public boolean setNominalCPUFrequency()
{
    try
    {
        Runtime.getRuntime().exec(SET_NOMINAL_FREQ_CMD);
        return true;
    }
    catch (IOException e)
    {
        Log.d("SETCPUFREQ:", e.getMessage());
        error = "Set frequency IO exception!";
        e.printStackTrace();
        return false;
    }
    catch(Exception e)
    {
        Log.d("SETCPUFREQ*", e.getMessage());
        error = e.getMessage();
        e.printStackTrace();
        return false;
    }
}

public boolean setLowestCPUFrequency()
{
    try
    {
        Runtime.getRuntime().exec(SET_LOWEST_FREQ_CMD);
        return true;
    }
    catch (IOException e)
    {
        Log.d("SETCPUFREQ:", e.getMessage());
        error = "Set frequency IO exception!";
        e.printStackTrace();
        return false;
    }
    catch(Exception e)
    {
        Log.d("SETCPUFREQ*", e.getMessage());
        error = e.getMessage();
        e.printStackTrace();
        return false;
    }
}

public boolean setLowNominalCPUFrequency()
{
    try
    {
        Runtime.getRuntime().exec(SET_LOWNOM_FREQ_CMD);

```



```

        return true;
    }
    catch (IOException e)
    {
        Log.d("SETCPUFREQ:", e.getMessage());
        error = "Set frequency IO exception!";
        e.printStackTrace();
        return false;
    }
    catch (Exception e)
    {
        Log.d("SETCPUFREQ*:", e.getMessage());
        error = e.getMessage();
        e.printStackTrace();
        return false;
    }
}

/**
 * Minimum CPU DVFS level
 * @return
 * The minimum supported DVFS level
 */
public int getMinCPUFrequency()
{
    return minFreq;
}

/**
 * Maximum CPU DVFS level
 * @return
 * The maximum supported DVFS level
 */
public int getMaxCPUFrequency()
{
    return maxFreq;
}

/**
 * Median CPU DVFS level
 * @return The median supported DVFS level
 */
public int getMedianCPUFrequency()
{
    int median = minFreq;
    if (freqModes.size() > 2)
    {
        median = freqModes.get(freqModes.size() / 2);
    }
    return median;
}

/**
 * The current DVFS level
 * @return
 * The current DVFS level of the processor if parsing was successful;
otherwise, will return -1
 */
public int getCPUFrequency()
{
    try
    {
        byte[] b = new byte[1024];

```

```

        Runtime.getRuntime().exec(String.format("cat
CUR_FREQ_FILE)).getInputStream().read(b);                                %s",
        String freq = new String(b).replace("\n", "").trim();
        if (freq.length() > 0)
        {
            return Integer.parseInt(freq);
        }
        return -1;
    }
    catch (FileNotFoundException e)
    {
        error = "Error reading cpu frequency file!";
        e.printStackTrace();
        return -1;
    }
    catch (IOException e)
    {
        error = "Error reading cpu frequency file!";
        e.printStackTrace();
        return -1;
    }
}

public boolean stepDownFrequency()
{
    int idx = indexOfFrequency(getCPUFrequency());
    if (getCPUFrequency() > minFreq && idx > 0)
    {
        return setCPUFrequency(freqModes.get(idx - 1));
    }
    return false;
}

public boolean stepUpFrequency()
{
    int idx = indexOfFrequency(getCPUFrequency());
    if (getCPUFrequency() < maxFreq && idx < freqModes.size() - 1)
    {
        return setCPUFrequency(freqModes.get(idx + 1));
    }
    return false;
}

public int indexOfFrequency(int frequency)
{
    if (freqModes.contains(frequency))
    {
        return freqModes.indexOf(frequency);
    }
    return -1;
}

/**
 * Called at instantiation to parse/set CPU min/max supported DVFS levels
(frequencies)
 */
private void readCPUMinMaxFrequencies()
{
    try
    {
        //Read Max Frequency
        String result = null;
        String cmdStr = String.format("cat %s", MAX_FREQ_FILE);

```

```

        Process process = Runtime.getRuntime().exec(cmdStr);
        InputStream in = process.getInputStream();
        byte[] read = new byte[1024];
        if (in.read(read) > 0)
        {
            result = new String(read);
        }
        else
        {
            error = "Error reading max frequency!";
        }
        in.close();
        maxFreq = Integer.parseInt(result.replace("\n", "").trim());
        //Read Min Frequency
        cmdStr = String.format("cat %s", MIN_FREQ_FILE);
        process = Runtime.getRuntime().exec(cmdStr);
        in = process.getInputStream();
        read = new byte[1024];
        if (in.read(read) > 0)
        {
            result = new String(read);
        }
        else
        {
            error = "Error reading min frequency!";
        }
        in.close();
        minFreq = Integer.parseInt(result.replace("\n", "").trim());
    }
    catch (IOException e)
    {
        error = "Error occurred reading min/max frequencies!";
        e.printStackTrace();
    }
    catch (NumberFormatException e)
    {
        error = "Error occurred parsing min/max frequencies!";
        e.printStackTrace();
    }
}

/**
 * Used internally to set min/max scaling frequencies for the current governor
 (only useful for OnDemand and UserSpace governors)
 */
private void setMinMaxScaleFrequencies()
{
    String[] maxStr = {"su", "-c", String.format("echo \"%d\" > %s", maxFreq,
SET_MAX_FREQ_FILE)};
    String[] minStr = {"su", "-c", String.format("echo \"%d\" > %s", minFreq,
SET_MIN_FREQ_FILE)};
    try
    {
        Runtime.getRuntime().exec(maxStr);
        Runtime.getRuntime().exec(minStr);
    }
    catch (IOException e)
    {
        e.printStackTrace();
        error = "Error setting scaling min/max frequencies!";
    }
}
}

```

```

/**
 * Change the current frequency governor
 * @param gov The desired frequency governor
 * @return
 * true if governor was successfully changed<br>
 * false if governor was not changed
 */
public boolean setFrequencyGovernor(FrequencyGovernors gov)
{
    try
    {
        byte[] b = new byte[1024];
        String[] str ={"su", "-c", String.format("echo \"%s\" > %s",
gov.toString().toLowerCase(), GOV_FILE)};
        Runtime.getRuntime().exec(str).getErrorStream().read(b);
        if (getCurrentGovernor() == gov)
        {
            return true;
        }
        else if (new String(b).toLowerCase().contains("permission
denied"))
        {
            error = "Error settings frequency governor due to
permissions!";
        }
        return false;
    }
    catch (IOException e)
    {
        error = "Error reading current frequency governor setting!";
        e.printStackTrace();
        return false;
    }
    catch(Exception e)
    {
        error = e.getMessage();
        e.printStackTrace();
        return false;
    }
}

/**
 * List of available governors as String[]
 * @return
 * An array of available governors
 */
public String getAvailableGovernors()
{
    try
    {
        byte[] b = new byte[1024];
        InputStream in = Runtime.getRuntime().exec(String.format("cat %s",
AVA_GOV_FILE)).getInputStream();
        if (in.read(b) > 0)
        {
            String result = new String(b);
            result = result.replace("\n", "").trim();
            in.close();
            return result;
        }
        in.close();
        return "";
    }
}

```

```

    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
        error = "Error reading available governors!";
        return "";
    }
    catch (IOException e)
    {
        e.printStackTrace();
        error = "Error reading available governors!";
        return "";
    }
}

/**
 * Current frequency governor
 * @return
 * The current frequency governor
 */
public FrequencyGovernors getCurrentGovernor()
{
    try
    {
        String result = null;
        String cmdStr = String.format("cat %s", GOV_FILE.replace(".txt",
""));
        InputStream in =
Runtime.getRuntime().exec(cmdStr).getInputStream();
        byte[] read = new byte[1024];
        if (in.read(read) > 0)
        {
            result = new String(read);
            result = result.replace("\n", "").trim().toLowerCase();
            if (result.equals("ondemand"))
            {
                governor = FrequencyGovernors.OnDemand;
            }
            else if (result.equals("conservative"))
            {
                governor = FrequencyGovernors.Conservative;
            }
            else if (result.equals("powersave"))
            {
                governor = FrequencyGovernors.Powersave;
            }
            else if (result.equals("userspace"))
            {
                governor = FrequencyGovernors.Userspace;
            }
            else if (result.equals("performance"))
            {
                governor = FrequencyGovernors.Performance;
            }
            else if (result.equals("interactive"))
            {
                governor = FrequencyGovernors.Interactive;
            }
            else
            {
                governor = FrequencyGovernors.Unknown;
                throw new Exception("Unknown frequency governor!");
            }
        }
    }
}

```

```

        }
        else
        {
            error = "Error reading current frequency governor
setting!";
        }
        in.close();
        return governor;
    }
    catch (IOException e)
    {
        error = "Error reading current frequency governor setting!";
        e.printStackTrace();
        return governor;
    }
    catch (Exception e)
    {
        error = e.getMessage();
        e.printStackTrace();
        return governor;
    }
}
} //End class

```

## A.9 DVFSTest.java (Strategy 1)

```

package csu.research.AURA;

import java.util.ArrayList;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.os.SystemClock;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.ToggleButton;

public class DVFSTest extends Activity
{
    private TextView mUpdateRateTextView;
    private TextView mCurFreqTextView;
    private Spinner mLowFreqSpinner;
    private Spinner mHighFreqSpinner;
    private Spinner mModeSpinner;
    private SeekBar mUpdateRateSeekBar;
    private ToggleButton mToggleButton;
    private int lowFreq;
    private int highFreq;
    private int updateRate;
    private ArrayList<Integer> freqModes;

```

```

private String currentMode;
private boolean isRunning = false;
private Thread mRunTestThread;

private static final String[] MODES = new String[]{"FREQ - TOGGLE EXTREMES",
"FREQ - STEP UP", "FREQ STEP DOWN", "SCREEN - TOGGLE EXTREMES", "SCREEN - STEP UP",
"SCREEN - STEP DOWN", "FREQ TOGGLE WITH UTIL", "FREQ DELTA TIME", "DELTA TIME COST"};
private static final float MIN_BRIGHTNESS = 0.1F;
private static final float MAX_BRIGHTNESS = 1F;
private static final float DELTA_BRIGHTNESS = 0.1F;

public void onCreate(Bundle savedInstanceState)
{
    setContentView(R.layout.dvfs_test);
    mUpdateRateTextView =
(TextView) findViewById(R.id.dvfs_test_updaterate_textview);
    mCurFreqTextView = (TextView)
findViewById(R.id.DVFS_Test_CurFreq_TextView);
    //Spinners
    mLowFreqSpinner = (Spinner) findViewById(R.id.DVFS_Test_LowFreq_Spinner);
    mHighFreqSpinner =
(Spinner) findViewById(R.id.DVFS_Test_HighFreq_Spinner);
    mModeSpinner = (Spinner) findViewById(R.id.DVFS_Test_Mode_Spinner);
    freqModes = AppMonitor.DVFSController.getFrequencyScaleModes();
    lowFreq = freqModes.get(0);
    highFreq = freqModes.get(freqModes.size() - 1);
    mLowFreqSpinner.setAdapter(new ArrayAdapter<Integer>(this,
R.layout.spinner_item_view, freqModes));
    mHighFreqSpinner.setAdapter(new ArrayAdapter<Integer>(this,
R.layout.spinner_item_view, freqModes));
    mModeSpinner.setAdapter(new ArrayAdapter<String>(this,
R.layout.spinner_item_view, MODES));
    mLowFreqSpinner.setSelection(freqModes.indexOf(lowFreq));
    mHighFreqSpinner.setSelection(freqModes.indexOf(highFreq));
    mModeSpinner.setSelection(0);

    mLowFreqSpinner.setOnItemClickListener(mLowFreqOnItemSelectedListener);

    mHighFreqSpinner.setOnItemClickListener(mHighFreqOnItemSelectedListener);
    mModeSpinner.setOnItemClickListener(mModeOnItemSelectedListener);
    currentMode = MODES[0];

    //SeekBar
    mUpdateRateSeekBar =
(SeekBar) findViewById(R.id.DVFS_Test_UpdateRate_SeekBar);
    mUpdateRateSeekBar.setOnSeekBarChangeListener(mUpdateRateChangeListener);
    updateRate = mUpdateRateSeekBar.getProgress();

    //Toggle Button
    mToggleButton = (ToggleButton) findViewById(R.id.DVFS_Test_ToggleButton);
    mToggleButton.setOnCheckedChangeListener(mOnToggleButtonCheckedChanged);
    super.onCreate(savedInstanceState);
}

private void setUpdateRateText(int rate)
{
    mUpdateRateTextView.setText(String.format("%d msec", rate));
}

private OnCheckedChangeListener mOnToggleButtonCheckedChanged = new
OnCheckedChangeListener()
{
    @Override

```

```

public void onCheckedChanged(CompoundButton btn, boolean isChecked)
{
    if (isRunning && !isChecked) //
    {
        mRunTestThread.interrupt();
        mRunTestThread.stop(new ThreadDeath());
        isRunning = false;
        Intent brightnessIntent = new Intent();

        brightnessIntent.setAction(AppMonitor.SCREENBRIGHTNESS_CHANGE_EVENT);
        brightnessIntent.putExtra("brightness", MAX_BRIGHTNESS);
        sendBroadcast(brightnessIntent);
    }
    else if (!isRunning && isChecked)
    {
        resetTestThread();
        mRunTestThread.start();
        isRunning = true;
    }
}

};

private OnItemSelectedListener mModeOnItemSelectedListener = new
OnItemSelectedListener()
{
    @Override
    public void onItemSelected(AdapterView<?> arg0, View arg1, int position,
long arg3)
    {
        if (position >= 0 && position < MODES.length)
        {
            currentMode = MODES[position];
            Intent brightnessIntent = new Intent();

            brightnessIntent.setAction(AppMonitor.SCREENBRIGHTNESS_CHANGE_EVENT);
            brightnessIntent.putExtra("brightness", MAX_BRIGHTNESS);
            sendBroadcast(brightnessIntent);
        }
    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0)
    {
    }
}

};

private OnItemSelectedListener mLowFreqOnItemSelectedListener = new
OnItemSelectedListener()
{
    @Override
    public void onItemSelected(AdapterView<?> arg0, View arg1, int position,
long arg3)
    {
        if (position >= 0)
        {
            lowFreq = freqModes.get(position);

            AppMonitor.DVFSController.setCPUFrequency(freqModes.get(position));
            updateFreqText(String.format("%.1f           MHz",
freqModes.get(position) / 1000F));
        }
    }
}

```



```

        @Override
        public void onNothingSelected(AdapterView<?> arg0)
        {
        }
    };

    private OnItemSelectedListener mHighFreqOnItemSelectedListener = new
    OnItemSelectedListener()
    {
        @Override
        public void onItemSelected(AdapterView<?> arg0, View arg1, int position,
long arg3)
        {
            if (position >= 0)
            {
                highFreq = freqModes.get(position);
            }
        }

        @Override
        public void onNothingSelected(AdapterView<?> arg0)
        {
        }
    };

    private OnSeekBarChangeListener mUpdateRateChangeListener = new
    OnSeekBarChangeListener()
    {
        @Override
        public void onProgressChanged(SeekBar arg0, int arg1, boolean arg2)
        {
        }

        @Override
        public void onStartTrackingTouch(SeekBar seekBar)
        {
        }

        @Override
        public void onStopTrackingTouch(SeekBar seekBar)
        {
            if (seekBar.getProgress() <= 100)
            {
                if (seekBar.getProgress() % 10 == 0)
                {
                    updateRate = seekBar.getProgress();
                }
                else if (seekBar.getProgress() > 10)
                {
                    updateRate = seekBar.getProgress() -
seekBar.getProgress() % 10;
                    seekBar.setProgress(updateRate);
                }
                else
                {
                    updateRate = 10;
                    seekBar.setProgress(10);
                }
            }
        }
    };

```

```

    }
    else
    {
        if (seekBar.getProgress() % 100 == 0)
        {
            updateRate = seekBar.getProgress();
        }
        else if (seekBar.getProgress() > 100)
        {
            updateRate = seekBar.getProgress() -
seekBar.getProgress() % 100;
            seekBar.setProgress(updateRate);
        }
        else
        {
            updateRate = 100;
            seekBar.setProgress(100);
        }
    }
    setUpdateRateText(updateRate);
}
};

private void resetTestThread()
{
    mRunTestThread = new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            try
            {
                int curFreq = highFreq;
                float curBrightness = MAX_BRIGHTNESS;
                Intent brightnessIntent = new Intent();

                brightnessIntent.setAction(AppMonitor.SCREENBRIGHTNESS_CHANGE_EVENT);
                long startTicks = 0;
                long stopTicks = 0;
                boolean toggle = true;
                while (isRunning)
                {
                    //FREQ MODES
                    if (currentMode.equals(MODES[0]))
                    {
                        if (curFreq == highFreq)
                        {
                            AppMonitor.DVFSController.setCPUFrequency(lowFreq);
                            curFreq = lowFreq;
                        }
                        else
                        {
                            AppMonitor.DVFSController.setCPUFrequency(highFreq);
                            curFreq = highFreq;
                        }
                        handler.sendMessage(curFreq);
                    }
                    else if (currentMode.equals(MODES[1]))
                    {
                        if (curFreq >= highFreq)
                        {

```

```

AppMonitor.DVFSController.setCPUFrequency(lowFreq);
                                curFreq = lowFreq;
                                }
                                else
                                {

AppMonitor.DVFSController.stepUpFrequency();
                                curFreq
AppMonitor.DVFSController.getCPUFrequency();
                                }
                                handler.sendMessage(curFreq);
                                }
                                else if (currentMode.equals(MODES[2]))
                                {
                                    if (curFreq <= lowFreq)
                                    {

AppMonitor.DVFSController.setCPUFrequency(highFreq);
                                    curFreq = highFreq;
                                    }
                                    else
                                    {

AppMonitor.DVFSController.stepDownFrequency();
                                    curFreq
AppMonitor.DVFSController.getCPUFrequency();
                                    }
                                    handler.sendMessage(curFreq);
                                    }
                                    //SCREEN MODES
                                    else if (currentMode.equals(MODES[3]))
                                    {
                                        if (curBrightness >= MAX_BRIGHTNESS)
                                        {
                                            curBrightness = MIN_BRIGHTNESS;
                                        }
                                        else
                                        {
                                            curBrightness = MAX_BRIGHTNESS;
                                        }
                                        brightnessIntent.putExtra("brightness",
curBrightness);

                                        sendBroadcast(brightnessIntent);
                                        handler.sendMessage((int)
(curBrightness * 100));
                                    }
                                    else if (currentMode.equals(MODES[4]))
                                    {
                                        if (curBrightness >= MAX_BRIGHTNESS)
                                        {
                                            curBrightness = MIN_BRIGHTNESS;
                                        }
                                        else
                                        {
                                            curBrightness = curBrightness +
DELTA_BRIGHTNESS;
                                        }
                                        brightnessIntent.putExtra("brightness",
curBrightness);

                                        sendBroadcast(brightnessIntent);
                                        handler.sendMessage((int)
(curBrightness * 100));

```

```

    }
    else if (currentMode.equals(MODES[5]))
    {
        if (curBrightness <= MIN_BRIGHTNESS)
        {
            curBrightness = MAX_BRIGHTNESS;
        }
        else
        {
            curBrightness = curBrightness -
DELTA_BRIGHTNESS;
        }
        brightnessIntent.putExtra("brightness",
curBrightness);
        sendBroadcast(brightnessIntent);
        handler.sendMessage((int)
(curBrightness * 100));
    }
    else if (currentMode.equals(MODES[6]))
    {
        int[] stats = CPUInfo.getCPUStats();
        if (curFreq == highFreq)
        {
            AppMonitor.DVFSController.setCPUFrequency(lowFreq);
            curFreq = lowFreq;
        }
        else
        {
            AppMonitor.DVFSController.setCPUFrequency(highFreq);
            curFreq = highFreq;
        }
        for (int i = 0; i < 5; i++)
        {
            Message msg = Message.obtain();
            msg.arg1 = curFreq;
            msg.arg2 = stats[0];
            handler.sendMessage(msg);
            for (int j = 0; j < updateRate *
10; j++)
            {
                j++;
                j--;
                Thread.sleep(1);
            }
            stats = CPUInfo.getCPUStats();
        }
    }
    else if (currentMode.equals(MODES[7]))
    {
        if(toggle)
        {
            AppMonitor.DVFSController.setCPUFrequency(lowFreq);
            startTicks =
SystemClock.currentThreadTimeMillis();
            AppMonitor.DVFSController.setCPUFrequency(highFreq);
            while (AppMonitor.DVFSController.getCPUFrequency() != highFreq);
        }
    }
}

```

```

        stopTicks = SystemClock.currentThreadTimeMillis();
        curFreq = highFreq;

        toggle = false;
    }
    else
    {
        AppMonitor.DVFSController.setCPUFrequency(highFreq);
        startTicks = SystemClock.currentThreadTimeMillis();
        AppMonitor.DVFSController.setCPUFrequency(lowFreq);
        while (AppMonitor.DVFSController.getCPUFrequency() !=
lowFreq);

        stopTicks = SystemClock.currentThreadTimeMillis();
        curFreq = lowFreq;

        toggle = true;
    }
    Message msg = Message.obtain();
    msg.arg1 = curFreq;
    msg.arg2 = (int) (stopTicks -
startTicks);

    handler.sendMessage(msg);
}
else if (currentMode.equals(MODES[8]))
{
    AppMonitor.DVFSController.setCPUFrequency(lowFreq);
    curFreq = lowFreq;

    while (AppMonitor.DVFSController.getCPUFrequency() != curFreq);
    int sum = 0;
    int[] stats;
    while (true)
    {
        sum = 0;
        for (int i = 0; i < 50; i++)
        {
            stats = CPUInfo.getCPUStats();

            sum = sum + stats[0];
            //Thread.sleep(updateRate
/ 10);
        }
        Message msg = Message.obtain();
        msg.arg1 = curFreq;
        msg.arg2 = (int) (sum / 50F);
        handler.sendMessage(msg);
        Thread.sleep(5000);
    }
    Thread.sleep(4000);
}
}
catch (InterruptedException e)
{
    //handler.sendEmptyMessage(-1);
}
}
});
}

public synchronized void updateFreqText(String text)

```

```

        {
            mCurFreqTextView.setText(text);
        }

private Handler handler = new Handler()
{
    @Override
    public void handleMessage(Message msg)
    {
        if (isRunning)
        {
            if (currentMode.equals(MODES[6])
                {
                    updateFreqText (String.format ("%d,%d",          msg.arg2,
msg.arg1 / 1000));
                }
            else if (currentMode.equals (MODES[7]) ||
currentMode.equals (MODES[8]))
                {
                    updateFreqText (String.format ("%d,%d",          msg.arg2,
msg.arg1));
                }
            else
            {
                if (msg.what < 0)
                {
                    updateFreqText ("N/A");
                }
                else if (msg.what > 100)
                {
                    updateFreqText (String.format ("%1f          MHz",
msg.what / 1000F));
                }
                else
                {
                    updateFreqText (String.format ("%d          %%",
msg.what));
                }
            }
        }
    }
};
}

```

## A.10 FrequencyGovernors.java (Strategy 1)

```

package csu.research.AURA;

public enum FrequencyGovernors
{
    Conservative,
    OnDemand,
    Powersave,
    Performance,
    Userspace,
    Interactive,
    Unknown
}

```

## A.11 MiscParamInfo.java (Strategy 1)

```
package csu.research.AURA;

class MiscParamInfo implements Comparable<MiscParamInfo>
{
    private float mUpTimeSec;
    private int mAvgCpuUtil;
    private float mScreenOnPct;
    private int mScreenBrightnessPct;
    private float mWifiConnectedPct;
    private float mWifiHasTrafficPct;
    private long mWifiTraffic;
    private float mOffHookPct;
    private float mDataHasTrafficPct;
    private long mDataTraffic;
    private float mAudioPlayingPct;
    private float mBluetoothOnPct;
    private float mGpsEnabledPct;

    public MiscParamInfo(float upTimeSec, int cpu, float screenOn, int
screenBrightness, float wifiConn, float wifiHasTraffic,
        long wifiTraffic, float offHook, float dataHasTraffic, long dataTraffic,
float audio, float bt, float gps)
    {
        mUpTimeSec = upTimeSec;
        mAvgCpuUtil = cpu;
        mScreenOnPct = screenOn;
        mScreenBrightnessPct = screenBrightness;
        mWifiConnectedPct = wifiConn;
        mWifiHasTrafficPct = wifiHasTraffic;
        mWifiTraffic = wifiTraffic;
        mOffHookPct = offHook;
        mDataHasTrafficPct = dataHasTraffic;
        mDataTraffic = dataTraffic;
        mAudioPlayingPct = audio;
        mBluetoothOnPct = bt;
        mGpsEnabledPct = gps;
    }

    public float getUpTimeSec(){ return mUpTimeSec; }
    public int getAvgCpuUtil(){ return mAvgCpuUtil; }
    public float getScreenOnPct(){ return mScreenOnPct; }
    public int getScreenBrightnessPct(){ return mScreenBrightnessPct; }
    public float getWifiConnectedPct(){ return mWifiConnectedPct; }
    public float getWifiHasTrafficPct() { return mWifiHasTrafficPct; }
    public long getWifiTraffic(){ return mWifiTraffic; }
    public float getOffHookPct(){ return mOffHookPct; }
    public float getDataHasTrafficPct() { return mDataHasTrafficPct; }
    public long getDataTraffic(){ return mDataTraffic; }
    public float getAudioPlayingPct(){ return mAudioPlayingPct; }
    public float getBluetoothOnPct(){ return mBluetoothOnPct; }
    public float getGpsEnabledPct(){ return mGpsEnabledPct; }

    @Override
    public int compareTo(MiscParamInfo another) {
        return (int) (mUpTimeSec - another.getUpTimeSec());
    }
}
```

## A.12 NetInfo.java (Strategy 1)

```
package csu.research.AURA;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Method;
import java.util.Vector;

import android.location.LocationManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.net.wifi.WifiManager;
import android.telephony.TelephonyManager;
import android.util.Log;

public class NetInfo {
    //final private String NET_STAT_FILE = "/proc/net/netstat";
    final private String DEV_FILE = "/proc/self/net/dev";
    final private String WIFI_INTERFACE = "wlan0";
    final private String CELL_INTERFACE = "rmnet";
    final private String ETH_INTERFACE = "eth0";
    private WifiManager mWifiManager;
    private TelephonyManager mTelephonyManager;
    private ConnectivityManager mConnectivityManager;
    private static int mCellIn = 0;
    private static int mCellOut = 0;
    private static int mWifiIn = 0;
    private static int mWifiOut = 0;
    private static int mEthIn = 0;
    private static int mEthOut = 0;

    Method dataConnSwitchMethod;
    Class telephonyManagerClass;
    Object ITelephonyStub;
    Class ITelephonyClass;

    public NetInfo(WifiManager wifiManager, ConnectivityManager connectivityManager,
        TelephonyManager telephonyManager) {
        this.mWifiManager = wifiManager;
        this.mConnectivityManager = connectivityManager;
        this.mTelephonyManager = telephonyManager;
    }

    public long getCellTraffic() {
        FileReader fstream;
        try {
            fstream = new FileReader(DEV_FILE);
        } catch (FileNotFoundException e) {
            Log.e("MonNet", "Could not read " + DEV_FILE);
            return -1;
        }
        BufferedReader in = new BufferedReader(fstream, 500);
        String line;
        String[] segs;
        try {
            while ((line = in.readLine()) != null) {
                if (line.startsWith(CELL_INTERFACE)) {
                    segs = line.trim().split("[: ]+");
                    // cell in
                }
            }
        }
    }
}
```



```

        mCellIn += Integer.parseInt(segs[1]);

        // cell out
        mCellOut += Integer.parseInt(segs[9]);
    }
}
return mCellIn + mCellOut;
} catch (IOException e) {
    Log.e("MonNet", e.toString());
    return -1;
}
}

public long getWifiTraffic() {
    FileReader fstream;
    try {
        fstream = new FileReader(DEV_FILE);
    } catch (FileNotFoundException e) {
        Log.e("MonNet", "Could not read " + DEV_FILE);
        return -1;
    }
    BufferedReader in = new BufferedReader(fstream, 500);
    String line;
    String[] segs;
    try {
        while ((line = in.readLine()) != null) {
            if (line.startsWith(WIFI_INTERFACE)) {
                segs = line.trim().split("[: ]+");
                // wifi in
                mWifiIn += Integer.parseInt(segs[1]);

                // wifi out
                mWifiOut += Integer.parseInt(segs[9]);
            }
        }
        return mWifiIn + mWifiOut;
    } catch (IOException e) {
        Log.e("MonNet", e.toString());
        return -1;
    }
}

public long getEthernetTraffic() {
    FileReader fstream;
    try {
        fstream = new FileReader(DEV_FILE);
    } catch (FileNotFoundException e) {
        Log.e("MonNet", "Could not read " + DEV_FILE);
        return -1;
    }
    BufferedReader in = new BufferedReader(fstream, 500);
    String line;
    String[] segs;
    try {
        while ((line = in.readLine()) != null) {
            if (line.startsWith(ETH_INTERFACE)) {
                segs = line.trim().split("[: ]+");
                // ethernet in
                mEthIn += Integer.parseInt(segs[1]);

                // ethernet out
                mEthOut += Integer.parseInt(segs[9]);
            }
        }
    }
}

```

```

        }
        return mEthIn + mEthOut;
    } catch (IOException e) {
        Log.e("MonNet", e.toString());
        return -1;
    }
}

public String getCallStatus() {
    if (mTelephonyManager.getCallState() == TelephonyManager.CALL_STATE_OFFHOOK) {
        return "off hook";
    }
    else if (mTelephonyManager.getCallState() ==
TelephonyManager.CALL_STATE_RINGING) {
        return "ringing";
    }
    return "idle";
}

public String getDataStatus() {
    if (mTelephonyManager.getDataState() == TelephonyManager.DATA_CONNECTED) {
        if (mTelephonyManager.getNetworkType() ==
TelephonyManager.NETWORK_TYPE_UMTS) {
            return "3g connected";
        }
        else if (mTelephonyManager.getNetworkType() ==
TelephonyManager.NETWORK_TYPE_CDMA ||
mTelephonyManager.getNetworkType() ==
TelephonyManager.NETWORK_TYPE_EDGE) {
            return "edge connected";
        }
    }
    return "disconnected";
}

public void enableData() {
    if(mTelephonyManager.getDataState() == TelephonyManager.DATA_CONNECTED){
        return;
    }

    try{
        telephonyManagerClass
Class.forName(mTelephonyManager.getClass().getName());
        Method
getITelephonyMethod
telephonyManagerClass.getDeclaredMethod("getITelephony");
getITelephonyMethod.setAccessible(true);
ITelephonyStub = getITelephonyMethod.invoke(mTelephonyManager);
ITelephonyClass = Class.forName(ITelephonyStub.getClass().getName());

        dataConnSwitchmethod = ITelephonyClass
            .getDeclaredMethod("enableDataConnectivity");

        dataConnSwitchmethod.setAccessible(true);
        dataConnSwitchmethod.invoke(ITelephonyStub);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

public void disableData() {
    if(mTelephonyManager.getDataState() != TelephonyManager.DATA_CONNECTED){
        return;
    }
}

```

```

    }

    try{
        telephonyManagerClass
Class.forName(mTelephonyManager.getClass().getName());
        Method
        getITelephonyMethod
telephonyManagerClass.getDeclaredMethod("getITelephony");
        getITelephonyMethod.setAccessible(true);
        ITelephonyStub = getITelephonyMethod.invoke(mTelephonyManager);
        ITelephonyClass = Class.forName(ITelephonyStub.getClass().getName());

        dataConnSwitchmethod = ITelephonyClass
            .getDeclaredMethod("disableDataConnectivity");

        dataConnSwitchmethod.setAccessible(true);
        dataConnSwitchmethod.invoke(ITelephonyStub);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

public boolean isDataConnected() {
    if(mTelephonyManager.getDataState() == TelephonyManager.DATA_CONNECTED){
        return true;
    }
    return false;
}

public void enableWifi() {
    try {
        mWifiManager.setWifiEnabled(true);
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

public void disableWifi() {
    mWifiManager.setWifiEnabled(false);
}

public boolean isWifiConnected() {
    NetworkInfo.State state = getWifiState();
    if (state != null && state == NetworkInfo.State.CONNECTED) {
        return true;
    }
    return false;
}

public boolean isWifiEnabled() {
    return mWifiManager.isWifiEnabled();
}

public boolean isWifiDisconnected() {
    NetworkInfo.State state = getWifiState();
    if (state != null && state == NetworkInfo.State.DISCONNECTED) {
        return true;
    }
    return false;
}

public NetworkInfo.State getWifiState() {

```

```

        try {
            NetworkInfo state =
mConnectivityManager.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
//WifiInfo.getDetailedStateOf(wifiManager.getConnectionInfo().getSupplicantState());
            return state.getState();
        }
        catch (Exception e) {
            return null;
        }
    }

    public int getWifiSignalLevel() {
        return mWifiManager.getConnectionInfo().getRssi();
    }
}

```

### A.13 PowerModels.java (Strategy 1)

```

package csu.research.AURA;

public class PowerModels
{
    // private static final float CPU_POWER_MODEL_M = 0.000002F; // mW/%/kHz
    // private static final float CPU_POWER_MODEL_B = 3.0511F; // mW/%
    //
    // private static final float COST_MODEL_COEFFICIENT = 250.84F; //mW/kHz
    // private static final float COST_MODEL_EXPONENT = -0.447F;
    //
    // private static final float SCREEN_POWER_MODEL_M = 5.968F; // mW/%
    //
    // NEW VALUES FOR NEXUS ONE
    private static final float CPU_POWER_MODEL_M = 0.0021F; // mW/%/kHz
    private static final float CPU_POWER_MODEL_B = -0.2834F; // mW/%

    private static final float COST_MODEL_COEFFICIENT = 0.0448F; //mW/kHz
    private static final float COST_MODEL_EXPONENT = 0.4482F;

    private static final float SCREEN_POWER_MODEL_M = 3.3296F; // mW/%

    public static float getMWCPUPower(int freqKHz, int cpuUtilPct)
    {
        return ((CPU_POWER_MODEL_M * freqKHz) + CPU_POWER_MODEL_B) * cpuUtilPct;
    }

    public static float getMWFrequencySwitchPower(int freqKHz)
    {
        return (float) (COST_MODEL_COEFFICIENT * Math.pow(freqKHz,
COST_MODEL_EXPONENT));
    }

    public static float getMWCPUPowerSavings(int fromFreqKHz, int toFreqKHz, int
cpuUtilPct)
    {
        return (getMWCPUPower(fromFreqKHz, cpuUtilPct) -
getMWFrequencySwitchPower(fromFreqKHz) - getMWCPUPower(toFreqKHz, cpuUtilPct));
    }

    public static float getPercentCPUPowerSavings(int fromFreqKHz, int toFreqKHz,
int cpuUtilPct)
    {

```

```

        float fromPowerMinusCost = getMWCPUPower(fromFreqKHz, cpuUtilPct) -
getMWFrequencySwitchPower(fromFreqKHz);
        float toPower = getMWCPUPower(toFreqKHz, cpuUtilPct);
        if (fromPowerMinusCost > 0F)
        {
            return ((fromPowerMinusCost - toPower) * 100) / fromPowerMinusCost;
        }
        return 0F;
    }

    public static float getMJCPUEnergySavings(int fromFreqKHz, int toFreqKHz, int
cpuUtilPct, float deltaTimeMsec)
    {
        return getMWCPUPowerSavings(fromFreqKHz, toFreqKHz, cpuUtilPct) *
deltaTimeMsec / 1000F;
    }

    public static float getMWScreenPower(int brightnessPct)
    {
        return SCREEN_POWER_MODEL_M * brightnessPct;
    }

    public static float getMWScreenPowerSavings(int fromBrightnessPct, int
toBrightnessPct)
    {
        return
            getMWScreenPower(fromBrightnessPct) -
getMWScreenPower(toBrightnessPct);
    }

    public static float getMWScreenPowerSavings(float fromBrightnessPct, float
toBrightnessPct)
    {
        return getMWScreenPowerSavings((int) (100 * fromBrightnessPct), (int)
(100 * toBrightnessPct));
    }

    public static float getPercentScreenPowerSavings(float fromBrightnessFraction,
float toBrightnessFraction)
    {
        float fromPower = getMWScreenPower((int)(100 * fromBrightnessFraction));
        float toPower = getMWScreenPower((int) (100 * toBrightnessFraction));
        if (fromPower > 0F)
        {
            return ((fromPower - toPower) * 100) / fromPower;
        }
        return 0F;
    }

    public static float getMJScreenEnergySavings(int fromBrightnessPct, int
toBrightnessPct, float deltaTimeMsec)
    {
        return getMWScreenPowerSavings(fromBrightnessPct, toBrightnessPct) *
deltaTimeMsec / 1000F;
    }
}

//*****OLD POWER
MODELS*****//
// private static final float LINEAR_POWER_MODEL_M = 0.00008F; //mW
// private static final float LINEAR_POWER_MODEL_B = 973.64F; //mW
//

```

```

// private static final float LINEAR_SWITCHTIME_MODEL_M = -0.0005F; //ms
// private static final float LINEAR_SWITCHTIME_MODEL_B = 467.53F; //ms
//
// private static final int RELATIVE_FREQUENCY_KHZ = 528000; //kHz
//
// private static final float SWITCH_POWER_MW = 50; //mW
//
// /**
//  * ONLY VALID FOR FREQUENCIES BELOW 528 MHz
//  * @param freqKHz - Frequency below 528 MHz
//  * @param dT_msec - Delta time in ms
//  * @return Energy saved in mJ
//  */
// public static float getMJEnergySavedLinearNoCost(int freqKHz, int dT_msec)
// {
//     return ((LINEAR_POWER_MODEL_M * freqKHz) + LINEAR_POWER_MODEL_B) *
dT_msec / 1000F;
// }
//
// /**
//  * ONLY VALID FOR FREQUENCIES BELOW 528 MHz
//  * @param freqKHz - Frequency below 528 MHz
//  * @param dT_msec - Delta time in ms
//  * @return Energy saved in mJ
//  */
// public static float getMJEnergySavedLinearWithCostConstant(int freqKHz, int
dT_msec)
// {
//     return getMJEnergySavedLinearNoCost(freqKHz, dT_msec -
getMSSwitchTimeLinear(RELATIVE_FREQUENCY_KHZ));
// }
//
// /**
//  * ONLY VALID FOR FREQUENCIES BELOW 528 MHz
//  * @param freqKHz - Frequency below 528 MHz
//  * @param dT_msec - Delta time in ms
//  * @return Energy saved in mJ
//  */
// public static float getMJEnergySavedLinearWithCost(int freqKHz, int dT_msec)
// {
//     return getMJEnergySavedLinearNoCost(freqKHz, dT_msec) -
getMJSwitchEnergyLinear(freqKHz);
// }
//
// /**
//  * ONLY VALID FOR FREQUENCIES BELOW 528 MHz
//  * @param freqKHz - Frequency below 528 MHz
//  * @return Energy cost in mJ
//  */
// public static float getMJEnergySwitchTimeLinearNoCostConstant(int freqKHz)
// {
//     return getMJEnergySavedLinearNoCost(freqKHz,
getMSSwitchTimeLinear(freqKHz));
// }
//
// /**
//  * VALID ACROSS ENTIRE FREQUENCY RANGE 128 MHz - 614 MHz
//  * @param freqKHz - Frequency between 128000 kHz - 614400 kHz
//  * @return Time to switch in ms
//  */
// public static int getMSSwitchTimeLinear(int freqKHz)
// {

```

```

//          return      (int)      ((LINEAR_SWITCHTIME_MODEL_M      *      freqKHz)      +
LINEAR_SWITCHTIME_MODEL_B);
//      }
//
//      /**
//      * VALID ACROSS ENTIRE FREQUENCY RANGE 128 MHz - 614 MHz
//      * @param freqKHz - Frequency between 128000 kHz - 614400 kHz
//      * @return Amount of energy consumed to perform frequency switch
//      */
//      public static float getMJSwitchEnergyLinear(int freqKHz)
//      {
//          return getMSSwitchTimeLinear(freqKHz) * SWITCH_POWER_MW;
//      }

```

## A.14 SimpleXYSeries.java (Strategy 1)

```

package csu.research.AURA;

import com.androidplot.series.XYSeries;

public class SimpleXYSeries implements XYSeries
{
    private static final int[] vals = {0, 25, 55, 2, 80, 30, 99, 0, 44, 6};

    // f(x) = x
    @Override
    public Number getX(int index) {
        return index;
    }

    // range begins at 0
    public Number getMinX() {
        return 0;
    }

    // range ends at 9
    public Number getMaxX() {
        return 9;
    }

    @Override
    public String getTitle() {
        return "Some Numbers";
    }

    // range consists of all the values in vals
    @Override
    public int size() {
        return vals.length;
    }

    // return vals[index]
    @Override
    public Number getY(int index) {
        // make sure index isnt something unexpected:
        if(index < 0 || index > 9) {
            throw new IllegalArgumentException("Only values between 0 and 9 are
allowed.");
        }
    }
}

```

```

        return vals[index];
    }

    // smallest value in vals is 0
    public Number getMinY() {
        return 0;
    }

    // largest value in vals is 99
    public Number getMaxY() {
        return 99;
    }
}

```

### A.15 State.java (Strategy 1)

```

package csu.research.AURA;

public enum State
{
    DEFAULT,
    LOW,
    HIGH
}

```

### A.16 TrainingSetEntry.java (Strategy 1)

```

package csu.research.AURA;

class TrainingSetEntry
{
    private String meanLevel;
    private String stdDevLevel;
    private String eventTimeLevel;

    public TrainingSetEntry(String mLevel, String sdLevel, String etLevel)
    {
        meanLevel = mLevel;
        stdDevLevel = sdLevel;
        eventTimeLevel = etLevel;
    }

    public String getMeanLevel() { return meanLevel; }
    public String getStdDevLevel() { return stdDevLevel; }
    public String getEventTimeLevel() { return eventTimeLevel; }
    public void setMeanLevel(String ml) { meanLevel = ml; }
    public void setStdDevLevel(String sdl) { stdDevLevel = sdl; }
    public void setEventTimeLevel(String ttl) { eventTimeLevel = ttl; }
}

```

### A.17 AppClassification.java (Strategy 1)

```

package csu.research.AURA;

public enum AppClassification
{
    NotClassified,
}

```



```

VeryLowInteraction,    // journal addition
LowInteraction,
LowMedInteraction,    // journal addition
MedInteraction,
MedHighInteraction,   // journal addition
HighInteraction,
VeryHighInteraction;  // journal addition

    public static final String[] CLASSIFIED_DESCRIPTIONS = new String[] {"VERY LOW
INTERACTION",
    "LOW INTERACTION",
    "LOW MED INTERACTION",
    "MED INTERACTION",
    "MED HIGH INTERACTION",
    "HIGH INTERACTION",
    "VERY HIGH INTERACTION" };

    public static int toInt(AppClassification classification)
    {
        switch (classification)
        {
            case NotClassified:
                return 0;
            case VeryLowInteraction:
                return 1;
            case LowInteraction:
                return 2;
            case LowMedInteraction:
                return 3;
            case MedInteraction:
                return 4;
            case MedHighInteraction:
                return 5;
            case HighInteraction:
                return 6;
            case VeryHighInteraction:
                return 7;
            default:
                return -1;
        }
    }

    public static AppClassification fromInt(int classification)
    {
        switch (classification)
        {
            case 0:
                return AppClassification.NotClassified;
            case 1:
                return AppClassification.VeryLowInteraction;
            case 2:
                return AppClassification.LowInteraction;
            case 3:
                return AppClassification.LowMedInteraction;
            case 4:
                return AppClassification.MedInteraction;
            case 5:
                return AppClassification.MedHighInteraction;
            case 6:
                return AppClassification.HighInteraction;
            case 7:
                return AppClassification.VeryHighInteraction;
            default:

```

```

        return AppClassification.NotClassified;
    }
}

public static String toString(AppClassification classification)
{
    switch (classification)
    {
        case NotClassified:
            return "Not Classified";
        case VeryLowInteraction:
            return "Very Low Interaction";
        case LowInteraction:
            return "Low Interaction";
        case LowMedInteraction:
            return "Low Med Interaction";
        case MedInteraction:
            return "Med Interaction";
        case MedHighInteraction:
            return "Med High Interaction";
        case HighInteraction:
            return "High Interaction";
        case VeryHighInteraction:
            return "Very High Interaction";
        default:
            return null;
    }
}
}

```

## A.18 AppClassifier.java (Strategy 1)

```

package csu.research.AURA;

public abstract class AppClassifier
{
    public static AppClassification rangeAppClassifier(float touchMean, float
touchDev, float keyMean, float keyDev)
    {
        int MIN_MEAN_LEVEL = 500; //msec
        int MAX_VERY_HIGH_MEAN_LEVEL = 1000; // journal addition
        int MAX_HIGH_MEAN_LEVEL = 2000; //msec
        int MAX_MED_HIGH_MEAN_LEVEL = 3000; // journal addition
        int MAX_MED_MEAN_LEVEL = 4000; //msec
        int MAX_LOW_MED_MEAN_LEVEL = 5000; // journal addition
        int MAX_LOW_MEAN_LEVEL = 6000; // journal addition
        //HIGH - Minimum mean level <= HIGH_MEAN_LEVEL
        //MED - Minimum mean level <= MED_MEAN_LEVEL
        //LOW - Minimum mean level > MED_MEAN_LEVEL
        AppClassification classification = AppClassification.NotClassified;
        //First classify on average time between events
        if (touchMean <= keyMean && touchMean > 0 && touchMean >= MIN_MEAN_LEVEL)
        {
            //Classify based on touch events (minimum of the two)
            if (touchMean <= MAX_VERY_HIGH_MEAN_LEVEL) {
                classification = AppClassification.VeryHighInteraction;
            }
            else if (touchMean <= MAX_HIGH_MEAN_LEVEL) {
                classification = AppClassification.HighInteraction;
            }
            else if (touchMean <= MAX_MED_HIGH_MEAN_LEVEL) {

```

```

        classification = AppClassification.MedHighInteraction;
    }
    else if (touchMean <= MAX_MED_MEAN_LEVEL) {
        classification = AppClassification.MedInteraction;
    }
    else if (touchMean <= MAX_LOW_MED_MEAN_LEVEL) {
        classification = AppClassification.LowMedInteraction;
    }
    else if (touchMean <= MAX_LOW_MEAN_LEVEL) {
        classification = AppClassification.LowInteraction;
    }
    else {
        classification = AppClassification.VeryLowInteraction;
    }
}
else if (keyMean < touchMean && keyMean > 0 && keyMean >= MIN_MEAN_LEVEL)
{
    //Classify based on touch events (minimum of the two)
    if (keyMean <= MAX_VERY_HIGH_MEAN_LEVEL) {
        classification = AppClassification.VeryHighInteraction;
    }
    else if (keyMean <= MAX_HIGH_MEAN_LEVEL) {
        classification = AppClassification.HighInteraction;
    }
    else if (keyMean <= MAX_MED_HIGH_MEAN_LEVEL) {
        classification = AppClassification.MedHighInteraction;
    }
    else if (keyMean <= MAX_MED_MEAN_LEVEL) {
        classification = AppClassification.MedInteraction;
    }
    else if (keyMean <= MAX_LOW_MED_MEAN_LEVEL) {
        classification = AppClassification.LowMedInteraction;
    }
    else if (keyMean <= MAX_LOW_MEAN_LEVEL) {
        classification = AppClassification.LowInteraction;
    }
    else {
        classification = AppClassification.VeryLowInteraction;
    }
}
//Should also look at standard deviation and alter accordingly here...
return classification;
}

// public static AppClassification powerAppClassifier(MiscParamInfo m)
// {
//     // TODO: change these levels
//     int MIN_PWR_LEVEL = 1; //mWatts
//     int MAX_HIGH_PWR_LEVEL = 2; //mWatts
//     int MAX_MED_PWR_LEVEL = 2; //mWatts
//     AppClassification classification = AppClassification.NotClassified;
//     //
//     // TODO: CPU coefficient should change depending on operating frequency!
//     // Power is in mW
//     double power = 3.97*m.getAvgCpuUtil() + 150.31*m.getScreenOnPct() +
// 2.07*m.getScreenBrightnessPct() +
//     // 389.97*m.getOffHookPct() + 522.67*m.getDataHasTrafficPct() +
// 3.47*m.getDataTraffic() +
//     // 1.77*m.getWifiConnectedPct() + 658.93*m.getWifiHasTrafficPct() +
// 0.518*m.getWifiTraffic() +
//     // 275.65*m.getAudioPlayingPct() + 169.08/*system idle?*/;
//     //
//     if (power >= MIN_PWR_LEVEL)

```

```

//      {
//          //Classify based on power
//          if (power <= MAX_HIGH_PWR_LEVEL)
//          {
//              // TODO: change classification
//              classification = AppClassification.HighInteraction;
//          }
//          else if (power <= MAX_MED_PWR_LEVEL)
//          {
//              // TODO: change classification
//              classification = AppClassification.MedInteraction;
//          }
//          else
//          {
//              // TODO: change classification
//              classification = AppClassification.LowInteraction;
//          }
//      }
//      return classification;
//  }
}

```

## A.19 ApplicationDatabase.java (Strategy 1)

```

package csu.research.AURA;

import android.widget.Toast;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Calendar;
import java.util.LinkedList;

public class ApplicationDatabase implements Serializable
{
    //Serial Interface
    private static final long serialVersionUID = 2330475987385493970L;

    //Attributes
    private LinkedList<ApplicationInfo> trainedApps;
    private LinkedList<ApplicationInfo> untrainedApps;
    private LinkedList<String> ignoredApps;
    private String saveError = "";

    //Constructor(s)
    public ApplicationDatabase()
    {
        trainedApps = new LinkedList<ApplicationInfo>();
        untrainedApps = new LinkedList<ApplicationInfo>();
        ignoredApps = new LinkedList<String>();
    }

    //Methods
    public boolean addUntrainedApp(ApplicationInfo appInfo)
    {

```

```

        if (!untrainedApps.contains(appInfo)
!ignoredApps.contains(appInfo.getName()))
        {
            return untrainedApps.add(appInfo);
        }
        return false;
    }

    public boolean addIgnoredApp(String appName)
    {
        for (int i = 0; i < untrainedApps.size(); i++)
        {
            if (untrainedApps.get(i).getName().equals(appName))
            {
                untrainedApps.remove(i);
                break;
            }
        }
        for (int i = 0; i < trainedApps.size(); i++)
        {
            if (trainedApps.get(i).getName().equals(appName))
            {
                trainedApps.remove(i);
                break;
            }
        }
        if (!ignoredApps.contains(appName))
        {
            return ignoredApps.add(appName);
        }
        return false;
    }

    public boolean removeIgnoredApp(String appName)
    {
        if (ignoredApps.contains(appName))
        {
            return ignoredApps.remove(appName);
        }
        return false;
    }

    public boolean removeUntrainedApp(ApplicationInfo appInfo)
    {
        if (appInfo != null && untrainedApps.contains(appInfo))
        {
            return untrainedApps.remove(appInfo);
        }
        return false;
    }

    public boolean removeTrainedApp(ApplicationInfo appInfo)
    {
        if (appInfo != null && trainedApps.contains(appInfo))
        {
            return trainedApps.remove(appInfo);
        }
        return false;
    }

    public boolean markAppAsTrained(ApplicationInfo appInfo)
    {

```

```

        if(appInfo != null && untrainedApps.contains(appInfo) &&
untrainedApps.remove(appInfo))
        {
            if (trainedApps.contains(appInfo))
            {

trainedApps.get(trainedApps.indexOf(appInfo)).setTrained(true);
            }
            else
            {
                appInfo.setTrained(true);
                trainedApps.add(appInfo);
            }
            return true;
        }
        return false;
    }

    public boolean markAppAsUntrained(ApplicationInfo appInfo)
    {
        if(appInfo != null && trainedApps.contains(appInfo) &&
trainedApps.remove(appInfo))
        {
            if (untrainedApps.contains(appInfo))
            {

untrainedApps.get(untrainedApps.indexOf(appInfo)).setTrained(false);
            }
            else
            {
                appInfo.setTrained(false);
                untrainedApps.add(appInfo);
            }
            return true;
        }
        return false;
    }

    public ApplicationInfo getApplicationInfo(String appName)
    {
        ApplicationInfo temp = new ApplicationInfo(appName, null);
        if (untrainedApps.contains(temp))
        {
            return untrainedApps.get(untrainedApps.indexOf(temp));
        }
        else if (trainedApps.contains(temp))
        {
            return trainedApps.get(trainedApps.indexOf(temp));
        }
        return null;
    }

    public boolean containsApplication(String appName)
    {
        return getApplicationInfo(appName) != null;
    }

    public boolean containsApplication(ApplicationInfo appInfo)
    {
        return containsApplication(appInfo.getName());
    }

    public int getNumberOfTrainedApps()

```

```

    {
        return trainedApps.size();
    }

    public int getNumberOfUntrainedApps()
    {
        return untrainedApps.size();
    }

    public int getNumberOfIgnoredApps()
    {
        return ignoredApps.size();
    }

    public LinkedList<ApplicationInfo> getUntrainedApps()
    {
        java.util.Collections.sort(untrainedApps);
        return untrainedApps;
    }

    public LinkedList<ApplicationInfo> getTrainedApps()
    {
        java.util.Collections.sort(trainedApps);
        return trainedApps;
    }

    public LinkedList<String> getIgnoredAppNames()
    {
        java.util.Collections.sort(ignoredApps);
        return ignoredApps;
    }

    public boolean isApplicationTrained(String appName)
    {
        ApplicationInfo temp = getApplicationInfo(appName);
        if (temp != null)
        {
            return temp.isTrained();
        }
        return false;
    }

    public boolean save(String filePath)
    {
        try
        {
            FileOutputStream fos = new
FileOutputStream(String.format("%s/AppStats.ser", filePath));
            ObjectOutputStream out = new ObjectOutputStream(fos);
            out.writeObject(this);
            out.close();
            for (int i = 0; i < untrainedApps.size(); i++)
            {
                untrainedApps.get(i).save(true);
            }
            for (int i = 0; i < trainedApps.size(); i++)
            {
                trainedApps.get(i).save(true);
            }
            saveError = "";
            return true;
        }
        catch (FileNotFoundException e)

```

```

        {
            e.printStackTrace();
            saveError = String.format("File path not found!\n%s",
e.getMessage());
            return false;
        }
        catch (IOException e)
        {
            e.printStackTrace();
            saveError = String.format("Serialization Error!\n%s",
e.getMessage());
            return false;
        }
    }

    public boolean backup(String filePath)
    {
        try
        {
            FileOutputStream fos = new
FileOutputStream(String.format("%s/AppStatsBackup.bak", filePath));
            ObjectOutputStream out = new ObjectOutputStream(fos);
            out.writeObject(this);
            out.close();
            for (int i = 0; i < untrainedApps.size(); i++)
            {
                untrainedApps.get(i).save(true);
            }
            for (int i = 0; i < trainedApps.size(); i++)
            {
                trainedApps.get(i).save(true);
            }
            saveError = "";
            return true;
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
            saveError = String.format("File path not found!\n%s",
e.getMessage());
            return false;
        }
        catch (IOException e)
        {
            e.printStackTrace();
            saveError = String.format("Serialization Error!\n%s",
e.getMessage());
            return false;
        }
    }

    public String getSaveError()
    {
        return saveError;
    }

    public boolean isDirty()
    {
        for (int i = 0; i < untrainedApps.size(); i++)
        {
            if (untrainedApps.get(i).isDirty())
            {
                return true;
            }
        }
    }

```



```

        }
    }
    for (int i = 0; i < trainedApps.size(); i++)
    {
        if (trainedApps.get(i).isDirty())
        {
            return true;
        }
    }
    return false;
}

public boolean exportDatabase(String filePath)
{
    try
    {
        ApplicationInfo appInfo;
        int[] usageHist;
        String temp;
        File file = new File(filePath);
        PrintWriter writer = new PrintWriter(file);
        //Write Trained Apps
        writer.println(String.format("%d/%d/%d - %d:%d:%d",
Calendar.MONTH, Calendar.DAY_OF_MONTH, Calendar.YEAR, Calendar.HOUR, Calendar.MINUTE,
Calendar.SECOND));
        writer.println();
        writer.println("TRAINED APPS");
        writer.println(String.format("Trained Apps = %d",
getNumberOfTrainedApps()));
        for (int i = 0; i < trainedApps.size(); i++)
        {
            appInfo = trainedApps.get(i);
            //Write Name
            writer.println(String.format("Name: %s",
appInfo.getName()));
            writer.println(String.format("Package: %s",
appInfo.getPackage()));
            //Write Usage Statistics
            writer.println(String.format("Mean Usage Time: %.3f s",
appInfo.getUsageMean()));
            writer.println(String.format("Std Dev Usage Time: %.3f s",
appInfo.getUsageDev()));
            writer.println(String.format("Median Usage Time: %.3f s",
appInfo.getUsageMedian()));
            writer.println(String.format("Min Usage Time: %.3f s",
appInfo.getUsageMin()));
            writer.println(String.format("Max Usage Time: %.3f s",
appInfo.getUsageMax()));
            usageHist = appInfo.getUsageHistogram();
            temp = "Usage Histogram:";
            for (int j = 0; j < usageHist.length; j++)
            {
                temp = String.format("%s %d", temp, usageHist[j]);
            }
            //Write Usage Histogram
            writer.println(temp);
            //Write Touch Statistics
            writer.println(String.format("Mean Touch Time: %.3f ms",
appInfo.getTouchMean()));
            writer.println(String.format("Std Dev Touch Time: %.3f ms",
appInfo.getTouchStandardDeviation()));
            writer.println(String.format("Median Touch Time: %.3f ms",
appInfo.getTouchMedian()));
        }
    }
}

```

```

writer.println(String.format("Min Touch Time: %.3f ms",
appInfo.getTouchMin()));
writer.println(String.format("Max Touch Time: %.3f ms",
appInfo.getTouchMax()));
//Write Key Statistics
writer.println(String.format("Mean Key Time: %.3f ms",
appInfo.getKeyMean()));
writer.println(String.format("Std Dev Key Time: %.3f ms",
appInfo.getKeyStandardDeviation()));
writer.println(String.format("Median Key Time: %.3f ms",
appInfo.getKeyMedian()));
writer.println(String.format("Min Key Time: %.3f ms",
appInfo.getKeyMin()));
writer.println(String.format("Max Key Time: %.3f ms",
appInfo.getKeyMax()));
}
//Write Untrained Apps
writer.println();
writer.println("UNTRAINED APPS");
writer.println(String.format("Untrained Apps = %d",
getNumberOfUntrainedApps()));
for (int i = 0; i < untrainedApps.size(); i++)
{
appInfo = untrainedApps.get(i);
//Write Name
writer.println(String.format("Name: %s",
appInfo.getName()));
writer.println(String.format("Package: %s",
appInfo.getPackage()));
//Write Usage Statistics
writer.println(String.format("Mean Usage Time: %.3f s",
appInfo.getUsageMean()));
writer.println(String.format("Std Dev Usage Time: %.3f s",
appInfo.getUsageDev()));
writer.println(String.format("Median Usage Time: %.3f s",
appInfo.getUsageMedian()));
writer.println(String.format("Min Usage Time: %.3f s",
appInfo.getUsageMin()));
writer.println(String.format("Max Usage Time: %.3f s",
appInfo.getUsageMax()));
usageHist = appInfo.getUsageHistogram();
temp = "Usage Histogram:";
for (int j = 0; j < usageHist.length; j++)
{
temp = String.format("%s %d", temp, usageHist[j]);
}
//Write Usage Histogram
writer.println(temp);
//Write Touch Statistics
writer.println(String.format("Mean Touch Time: %.3f ms",
appInfo.getTouchMean()));
writer.println(String.format("Std Dev Touch Time: %.3f ms",
appInfo.getTouchStandardDeviation()));
writer.println(String.format("Median Touch Time: %.3f ms",
appInfo.getTouchMedian()));
writer.println(String.format("Min Touch Time: %.3f ms",
appInfo.getTouchMin()));
writer.println(String.format("Max Touch Time: %.3f ms",
appInfo.getTouchMax()));
//Write Key Statistics
writer.println(String.format("Mean Key Time: %.3f ms",
appInfo.getKeyMean()));

```

```

        writer.println(String.format("Std Dev Key Time: %.3f ms",
appInfo.getKeyStandardDeviation()));
        writer.println(String.format("Median Key Time: %.3f ms",
appInfo.getKeyMedian()));
        writer.println(String.format("Min Key Time: %.3f ms",
appInfo.getKeyMin()));
        writer.println(String.format("Max Key Time: %.3f ms",
appInfo.getKeyMax()));
    }
    writer.close();
    return true;
}
catch (Exception e)
{
    return false;
}
}
}

```

## A.20 ApplicationInfo.java (Strategy 1)

```

package csu.research.AURA;

import java.io.Serializable;
import java.lang.reflect.Array;
import java.util.ArrayList;

public class ApplicationInfo implements Comparable<ApplicationInfo>, Serializable
{
    private static final long serialVersionUID = -7087509230873380357L;
    public static final int MAXIMUM_DATA_SIZE = 25;
    public static final int MAX_TR_DATA_SIZE = 50;
    public static final int MAXIMUM_MISS_COUNT = 5;
    public static final int SMOOTHING_CONSTANT = 1;

    private int[] usageHistogram = new int[24 * 4]; //15 minute resolution
    private String name = "";
    private String pkg = "";
    private boolean isTrained;
    private boolean isDirty;
    private int missCount = 0;
    private AppClassification appClassification;
    private ArrayList<Float> meanTouchTime = new ArrayList<Float>();
    private ArrayList<Float> meanKeyTime = new ArrayList<Float>();
    private ArrayList<Float> usageTime = new ArrayList<Float>();
    private ArrayList<Float> devTouchTime = new ArrayList<Float>();
    private ArrayList<Float> devKeyTime = new ArrayList<Float>();
    private ArrayList<Float> medianTouchTime = new ArrayList<Float>();
    private ArrayList<Float> medianKeyTime = new ArrayList<Float>();
    private ArrayList<MiscParamInfo> miscStats= new ArrayList<MiscParamInfo>();
    private ArrayList<TrainingSetEntry> trainingSet = new
ArrayList<TrainingSetEntry>();
    private int minFreq = 528000;
    private int maxFreq = 528000;
    private int defFreq = 528000;
    private boolean mIsDynamic = true;
    private float mStaticTouchMean;
    private float mStaticTouchDev;
    private float mStaticKeyMean;
    private float mStaticKeyDev;

```

```

        // journal additions
        private int numStates = 22;        // 2 states (below nominal and nominal) * 11
eval intervals (0-9, >9)
        private int numQStateDims = 2;
        private int[] QStateDims;
        private double [] qValues;
private Object qValuesTable;
        private int actions = 2;        // number of actions

        public ApplicationInfo(String name, String pkg)
        {
            this.name = name;
            this.pkg = pkg;
            isTrained = false;
            isDirty = true;
            appClassification = AppClassification.NotClassified;

            // journal additions
            QStateDims = new int[numQStateDims];
            QStateDims[0] = 2;
            QStateDims[1] = 11;
            int dims[] = {QStateDims[0], QStateDims[1], actions};
            // Create n-dimensional array with size given in stateDims array.
            qValuesTable = Array.newInstance( double.class, dims );
            initQValues();
        }

        public String getName()
        {
            return name;
        }

        public String getShortName()
        {
            String temp = name;
            if (temp.contains("."))
            {
                temp = name.substring(name.lastIndexOf(".") + 1, name.length());
            }
            return temp;
        }

        public String getPackage()
        {
            return pkg;
        }

        public boolean isTrained()
        {
            return isTrained;
        }

        public void setTrained(boolean trained)
        {
            isTrained = trained;
            if (isTrained) //Classify app
            {
                classifyApplication(AppClassifier.rangeAppClassifier(getTouchMean(),
getTouchStandardDeviation(),
getKeyMean(), getKeyStandardDeviation()));
            }
            else //"Declassify" app

```

```

        {
            appClassification = AppClassification.NotClassified;
        }
    }

    public void setTrainedWithClassification(AppClassification classification)
    {
        isTrained = true;
        classifyApplication(classification);
    }

    public boolean setMinMaxDefaultFrequencies(int min, int max, int def)
    {
        if (max >= min && def <= max && def >= min)
        {
            minFreq = min;
            maxFreq = max;
            defFreq = def;
            isDirty = true;
            return true;
        }
        return false;
    }

    public void setStatisticsMode(boolean isDynamic)
    {
        mIsDynamic = isDynamic;
        isDirty = true;
    }

    public void setStaticStatistics(float touchMean, float touchDev, float keyMean,
float keyDev)
    {
        mStaticTouchMean = touchMean;
        mStaticTouchDev = touchDev;
        mStaticKeyMean = keyMean;
        mStaticKeyDev = keyDev;
    }

    public boolean isDynamic()
    {
        return mIsDynamic;
    }

    public int getMinFrequency()
    {
        return minFreq;
    }

    public int getMaxFrequency()
    {
        return maxFreq;
    }

    public int getDefaultFrequency()
    {
        return defFreq;
    }

    public void classifyApplication(AppClassification classification)
    {
        appClassification = classification;
        isDirty = true;
    }

```

```

    }

    public AppClassification getAppClassification()
    {
        return appClassification;
    }

    public int compareTo(ApplicationInfo another)
    {
        return name.compareTo(another.getName());
    }

    @Override
    public boolean equals(Object another)
    {
        if (another instanceof ApplicationInfo)
        {
            ApplicationInfo appInfo = (ApplicationInfo)another;
            return name.equals(appInfo.getName());
        }
        return super.equals(another);
    }

    public void addTouchStats(float meanTouch, float devTouch, float medianTouch)
    {
        if (meanTouch > 0F)
        {
            if (meanTouchTime.size() < MAXIMUM_DATA_SIZE)
            {
                meanTouchTime.add(meanTouch);
                devTouchTime.add(devTouch);
                medianTouchTime.add(medianTouch);
            }
            else
            {
                java.util.Collections.sort(meanTouchTime);
                java.util.Collections.sort(devTouchTime);
                java.util.Collections.sort(medianTouchTime);
                float midPoint = meanTouchTime.get(MAXIMUM_DATA_SIZE / 2);
                float meanDiff = Math.abs(meanTouch - midPoint);
                float minDiff = Math.abs(meanTouch - meanTouchTime.get(0));
                float maxDiff = Math.abs(meanTouch -
meanTouchTime.get(meanTouchTime.size() - 1));
                float diff = Math.max(Math.max(meanDiff, minDiff),
maxDiff);

                if (diff == minDiff) //Maximum deviation is from minimum
value
                {
                    meanTouchTime.remove(0);
                    devTouchTime.remove(0);
                    medianTouchTime.remove(0);
                    meanTouchTime.add(0, meanTouch);
                    devTouchTime.add(0, devTouch);
                    medianTouchTime.add(0, medianTouch);
                }
                else if (diff == maxDiff) //Maximum deviation is from
maximum value
                {
                    meanTouchTime.remove(MAXIMUM_DATA_SIZE - 1);
                    devTouchTime.remove(MAXIMUM_DATA_SIZE - 1);
                    medianTouchTime.remove(MAXIMUM_DATA_SIZE - 1);
                    meanTouchTime.add(meanTouch);
                    devTouchTime.add(devTouch);
                }
            }
        }
    }

```

```

        medianTouchTime.add(medianTouch);
    }
    else
    {
        if (missCount++ > MAXIMUM_MISS_COUNT)
        {
            //Unmark app for retraining...
        }
    }
}
isDirty = true;
}
}

public void addKeyStats(float meanKey, float devKey, float medianKey)
{
    if (meanKey > 0F)
    {
        if (meanKeyTime.size() < MAXIMUM_DATA_SIZE)
        {
            meanKeyTime.add(meanKey);
            devKeyTime.add(devKey);
            medianKeyTime.add(medianKey);
        }
        else
        {
            java.util.Collections.sort(meanKeyTime);
            java.util.Collections.sort(devKeyTime);
            java.util.Collections.sort(medianKeyTime);
            float midPoint = meanKeyTime.get(MAXIMUM_DATA_SIZE / 2);
            float meanDiff = Math.abs(meanKey - midPoint);
            float minDiff = Math.abs(meanKey - meanKeyTime.get(0));
            float maxDiff = Math.abs(meanKey -
meanKeyTime.get(meanKeyTime.size() - 1));
            float diff = Math.max(Math.max(meanDiff, minDiff),
maxDiff);

            if (diff == minDiff)
            {
                meanKeyTime.remove(0);
                devKeyTime.remove(0);
                medianKeyTime.remove(0);
                meanKeyTime.add(0, meanKey);
                devKeyTime.add(0, devKey);
                medianKeyTime.add(0, medianKey);
            }
            else if (diff == maxDiff)
            {
                meanKeyTime.remove(MAXIMUM_DATA_SIZE - 1);
                devKeyTime.remove(MAXIMUM_DATA_SIZE - 1);
                medianKeyTime.remove(MAXIMUM_DATA_SIZE - 1);
                meanKeyTime.add(meanKey);
                devKeyTime.add(devKey);
                medianKeyTime.add(medianKey);
            }
        }
        isDirty = true;
    }
}

public void addUsageStats(int startMinuteOfDay, int stopMinuteOfDay, float
upTimeSec)
{
    if (upTimeSec > 0F)

```

```

        {
            int idx = startMinuteOfDay % 15;
            if (idx >= 0 && idx <= (24 * 15 - 1))
            {
                usageHistogram[idx] = usageHistogram[idx] + 1;
            }
            if (usageTime.size() < MAXIMUM_DATA_SIZE)
            {
                usageTime.add(upTimeSec);
            }
            else
            {
                java.util.Collections.sort(usageTime);
                float midPoint = usageTime.get(MAXIMUM_DATA_SIZE / 2);
                if (upTimeSec <= midPoint)
                {
                    usageTime.remove(0);
                    usageTime.add(0, upTimeSec);
                }
                else
                {
                    usageTime.remove(MAXIMUM_DATA_SIZE - 1);
                    usageTime.add(upTimeSec);
                }
            }
            isDirty = true;
        }
    }

    public void addMiscStats(float upTimeSec, int cpuTotalPct, int screenOn, int
screenBrightness, int wifiConnected, int wifiHasTraffic,
        long wifiTraffic, int offHook, int dataHasTraffic, long dataTraffic, int
audioPlaying, int bluetoothOn, int gpsEnabled)
    {
        if (upTimeSec > 0)
        {
            int avgCpuUtil = (int) (cpuTotalPct/upTimeSec);
            float screenOnPct = screenOn/upTimeSec;
            float wifiConnectedPct = wifiConnected/upTimeSec;
            float wifiHasTrafficPct = wifiHasTraffic/upTimeSec;
            float offHookPct = offHook/upTimeSec;
            float dataHasTrafficPct = dataHasTraffic/upTimeSec;
            float audioPlayingPct = audioPlaying/upTimeSec;
            float bluetoothOnPct = bluetoothOn/upTimeSec;
            float gpsEnabledPct = gpsEnabled/upTimeSec;

            MiscParamInfo m = new MiscParamInfo(upTimeSec, avgCpuUtil, screenOnPct,
screenBrightness, wifiConnectedPct, wifiHasTrafficPct,
                wifiTraffic, offHookPct, dataHasTrafficPct, dataTraffic,
audioPlayingPct, bluetoothOnPct, gpsEnabledPct);

            if (miscStats.size() < MAXIMUM_DATA_SIZE)
            {
                miscStats.add(m);
            }
            else
            {
                java.util.Collections.sort(miscStats);
                float midPoint = miscStats.get(MAXIMUM_DATA_SIZE / 2).getUpTimeSec();
                if (upTimeSec <= midPoint)
                {
                    miscStats.remove(0);
                }
            }
        }
    }

```



```

        miscStats.add(0, m);
    }
    else
    {
        miscStats.remove(MAXIMUM_DATA_SIZE - 1);
        miscStats.add(m);
    }
}
}

public void addTrainingSetEntry(String mean, String stdDev, String eventTime)
{
    int i;

    TrainingSetEntry t = new TrainingSetEntry(mean, stdDev, eventTime);

    if (trainingSet.size() < MAX_TR_DATA_SIZE)
    {
        // add training set entry
        trainingSet.add(t);
    }
    else
    {
        // delete oldest training set entry and add new one
        for (i = 0; i < trainingSet.size()-1; i++)
        {
            trainingSet.get(i).setMeanLevel(trainingSet.get(i+1).getMeanLevel());

trainingSet.get(i).setStdDevLevel(trainingSet.get(i+1).getStdDevLevel());

trainingSet.get(i).setEventTimeLevel(trainingSet.get(i+1).getEventTimeLevel());
        }
        trainingSet.remove(MAXIMUM_DATA_SIZE - 1);
        trainingSet.add(t);
    }
}

public float findTTPProb(String etLevel)
{
    int i;
    int count = 0;

    if (trainingSet.size() > 0)
    {
        for (i = 0; i < trainingSet.size(); i++)
        {
            if (trainingSet.get(i).getEventTimeLevel().compareTo(etLevel) == 0)
            {
                count++;
            }
        }

        return count/(float)trainingSet.size();
    }

    return 0;
}

public float findMeanAndTTPProb(String mLevel, String etLevel)
{
    int i;
    int classCount = 0;

```

```

int mCount = 0;
int highObserved = 0;
int medObserved = 0;
int lowObserved = 0;
int numAttrsObserved = 0;

if (trainingSet.size() > 0)
{
    for (i = 0; i < trainingSet.size(); i++)
    {
        if (trainingSet.get(i).getEventTimeLevel().compareTo(etLevel) == 0)
        {
            classCount++;
            if (trainingSet.get(i).getMeanLevel().compareTo(mLevel) == 0)
            {
                mCount++;
            }

            if (highObserved == 0 &&
trainingSet.get(i).getMeanLevel().compareTo("high") == 0)
            {
                highObserved = 1;
            }

            if (medObserved == 0 &&
trainingSet.get(i).getMeanLevel().compareTo("med") == 0)
            {
                medObserved = 1;
            }

            if (lowObserved == 0 &&
trainingSet.get(i).getMeanLevel().compareTo("low") == 0)
            {
                lowObserved = 1;
            }
        }
    }

    numAttrsObserved = highObserved + medObserved + lowObserved;

    return (mCount+SMOOTHING_CONSTANT)/(float) (classCount +
SMOOTHING_CONSTANT*numAttrsObserved);
}

public float findSDandTTPProb(String sdLevel, String etLevel)
{
    int i;
    int classCount = 0;
    int sdCount = 0;
    int highObserved = 0;
    int medObserved = 0;
    int lowObserved = 0;
    int numAttrsObserved = 0;

    if (trainingSet.size() > 0)
    {
        for (i = 0; i < trainingSet.size(); i++)
        {
            if (trainingSet.get(i).getEventTimeLevel().compareTo(etLevel) == 0)
            {
                classCount++;
                if (trainingSet.get(i).getStdDevLevel().compareTo(sdLevel) == 0)
                {
                    sdCount++;
                }
            }
        }
    }
}

```

```

        }

        if (highObserved == 0 &&
trainingSet.get(i).getStdDevLevel().compareTo("high") == 0)
        {
            highObserved = 1;
        }
        if (medObserved == 0 &&
trainingSet.get(i).getStdDevLevel().compareTo("med") == 0)
        {
            medObserved = 1;
        }
        if (lowObserved == 0 &&
trainingSet.get(i).getStdDevLevel().compareTo("low") == 0)
        {
            lowObserved = 1;
        }
    }
}

numAttrsObserved = highObserved + medObserved + lowObserved;

return (sdCount+SMOOTHING_CONSTANT)/(float)(classCount +
SMOOTHING_CONSTANT*numAttrsObserved);
}

public ArrayList<TrainingSetEntry> getTrainingSet()
{
    return trainingSet;
}

public ArrayList<Float> getTouchMeanList()
{
    return meanTouchTime;
}

public ArrayList<Float> getKeyMeanList()
{
    return meanKeyTime;
}

public float getTouchMean()
{
    if (mIsDynamic)
    {
        return calculateMean(meanTouchTime);
    }
    return mStaticTouchMean;
}

public float getTouchStandardDeviation()
{
    if (mIsDynamic)
    {
        return calculateMean(devTouchTime);
    }
    return mStaticTouchDev;
}

public float getTouchMedian()
{
    if (mIsDynamic)

```

```

        {
            return calculateMean(medianTouchTime);
        }
        return mStaticTouchMean;
    }

    public float getTouchMin()
    {
        if (meanTouchTime.size() > 0)
        {
            return java.util.Collections.min(meanTouchTime);
        }
        return 0F;
    }

    public float getTouchMax()
    {
        if (meanTouchTime.size() > 0)
        {
            return java.util.Collections.max(meanTouchTime);
        }
        return 0F;
    }

    public float getKeyMedian()
    {
        if (mIsDynamic)
        {
            return calculateMean(medianKeyTime);
        }
        return mStaticKeyMean;
    }

    public float getKeyMin()
    {
        if (meanKeyTime.size() > 0)
        {
            return java.util.Collections.min(meanKeyTime);
        }
        return 0F;
    }

    public float getKeyMax()
    {
        if (meanKeyTime.size() > 0)
        {
            return java.util.Collections.max(meanKeyTime);
        }
        return 0F;
    }

    public float getKeyStandardDeviation()
    {
        if (mIsDynamic)
        {
            return calculateMean(devKeyTime);
        }
        return mStaticKeyDev;
    }

    public float getKeyMean()
    {
        if (mIsDynamic)

```

```

        {
            return calculateMean(meanKeyTime);
        }
        return mStaticKeyMean;
    }

    public float getUsageMean()
    {
        return calculateMean(usageTime);
    }

    public float getUsageDev()
    {
        return calculateStandardDev(usageTime, getUsageMean());
    }

    public float getUsageMedian()
    {
        return calculateMedian(usageTime);
    }

    public float getUsageMin()
    {
        return java.util.Collections.min(usageTime);
    }

    public float getUsageMax()
    {
        return java.util.Collections.max(usageTime);
    }

    public int[] getUsageHistogram()
    {
        return usageHistogram;
    }

    public boolean isDirty()
    {
        return isDirty;
    }

    public void save(boolean wasSaved)
    {
        isDirty = !wasSaved;
    }

    private float calculateMean(ArrayList<Float> list)
    {
        float sum = 0;
        if (list.size() > 0)
        {
            for (int i = 0; i < list.size(); i++)
            {
                sum += list.get(i);
            }
            return sum / list.size();
        }
        return 0F;
    }

    private float calculateStandardDev(ArrayList<Float> list, float mean)
    {
        float sum = 0F;

```

```

        if (list.size() > 0)
        {
            for (int i = 0; i < list.size(); i++)
            {
                sum += Math.pow((list.get(i) - mean), 2);
            }
            return (float)Math.sqrt(sum / list.size());
        }
        return 0F;
    }

private float calculateMedian(ArrayList<Float> list)
{
    if (list.size() > 1 && (list.size() % 2) == 0) //Even number of points
    {
        java.util.Collections.sort(list);
        float first = list.get(list.size() / 2 - 1);
        float second = list.get(list.size() / 2);
        return (first + second) / 2;
    }
    else if (list.size() > 1) //Odd number of points
    {
        java.util.Collections.sort(list);
        return list.get(list.size() / 2);
    }
    else if (list.size() == 1)
    {
        return list.get(0);
    }
    return 0F;
}

// journal addition
/*****
 * Q Learning functions start here
 */
public void initQValues() {

    int i;
    int state[] = new int[QStateDims.length];

    for( int j = 0 ; j < numStates ; j++ ) {

        qValues = (double[]) myQValues( state );

        for( i = 0 ; i < actions ; i++ ) {
            // initialize qValues to all 0s
            Array.setDouble( qValues, i, 0); //+ 0.0000000000000000001 *
Math.random() );
        }

        state = getNextState( state );
    }
}

private int[] getNextState( int[] state ) {

    int i;
    int actualdim = 0;

    state[actualdim]++;
    if( state[actualdim] >= QStateDims[actualdim] ) {

```

```

        while( ( actualdim < QStateDims.length ) && ( state[actualdim] >=
QStateDims[actualdim] ) ) {
            actualdim++;

            if( actualdim == QStateDims.length )
                return state;

            state[actualdim]++;
        }
        for( i = 0 ; i < actualdim ; i++ )
            state[i] = 0;
        actualdim = 0;
    }
    return state;
}

```

```

private double[] myQValues( int[] state ) {

    int i;
    Object curTable = qValuesTable;

    for( i = 0 ; i < QStateDims.length - 1; i++ ) {
        //descend in each dimension
        curTable = Array.get( curTable, state[i] );
    }

    //at last dimension of Array get QValues.
    return (double[]) Array.get( curTable, state[i] );
}

```

```

public double[] getQValuesAt( int[] state ) {

    int i;
    Object curTable = qValuesTable;
    double[] returnValues;

    for( i = 0 ; i < QStateDims.length - 1 ; i++ ) {
        //descend in each dimension
        curTable = Array.get( curTable, state[i] );
    }

    //at last dimension of Array get QValues.
    qValues = (double[]) Array.get( curTable, state[i] );
    returnValues = new double[ qValues.length ];
    System.arraycopy( qValues, 0, returnValues, 0, qValues.length );
    return returnValues;
}

```

```

public void setQValue( int[] state, int action, double newQValue ) {
    qValues = myQValues( state );
    Array.setDouble( qValues, action, newQValue );
}

```

```

public double getMaxQValue( int[] state ) {

    double maxQ = -Double.MAX_VALUE;

    qValues = myQValues( state );

    for( int action = 0 ; action < qValues.length ; action++ ) {
        if( qValues[action] > maxQ ) {
            maxQ = qValues[action];
        }
    }
}

```

```

    }
    return maxQ;
}

public double getQValue( int[] state, int action ) {

    double qValue = 0;

    qValues = myQValues( state );
    qValue = qValues[action];

    return qValue;
}

public int getBestAction( int[] state ) {

    double maxQ = -Double.MAX_VALUE;
    int selectedAction = -1;
    int[] doubleValues = new int[qValues.length];
    int maxDV = 0;

    qValues = myQValues( state );

    for( int action = 0 ; action < qValues.length ; action++ ) {
        //System.out.println( "STATE: [" + state[0] + "," + state[1] + "]" );
        //System.out.println( "action:qValue, maxQ " + action + ":" +
qValues[action] + "," + maxQ );

        if( qValues[action] > maxQ ) {
            selectedAction = action;
            maxQ = qValues[action];
            maxDV = 0;
            doubleValues[maxDV] = selectedAction;
        }
        else if( qValues[action] == maxQ ) {
            maxDV++;
            doubleValues[maxDV] = action;
        }
    }

    if( maxDV > 0 ) {
        //System.out.println( "DOUBLE values, random selection, maxdv =" + maxDV
);
        int randomIndex = (int) ( Math.random() * ( maxDV + 1 ) );
        selectedAction = doubleValues[ randomIndex ];
    }

    if( selectedAction == -1 ) {
        //System.out.println("RANDOM Choice !" );
        selectedAction = (int) ( Math.random() * qValues.length );
    }

    return selectedAction;
}

/*public double getBestAction( int[]state, int bestAction ) {
private double bestAction( int[] state, int bestAction ) {

double maxQ = 0;
bestAction = -1;

qValues = getQValuesAt( state );

```



```

    for( int action = 0 ; action < qValues.length ; action++ ) {
        if( qValues[action] > maxQ ) {
            bestAction = action;
            maxQ = qValues[action];
        }
    }
    return maxQ;
}
*/

/*****
 * end Q Learning functions
 */
}

```

## A.21 AppMonitor.java (Strategy 1)

```

package csu.research.AURA;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.ObjectInputStream;
import java.io.PrintWriter;
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.Vector;

import android.app.ActivityManager;
import android.app.AlertDialog;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.BroadcastReceiver;
import android.content.ContentResolver;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.location.GpsSatellite;
import android.location.GpsStatus;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.location.LocationProvider;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.wifi.WifiManager;
import android.os.BatteryManager;
import android.os.Binder;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.IBinder;

```

```

import android.os.PowerManager;
import android.os.SystemClock;
import android.provider.Settings;
import android.provider.Settings.SettingNotFoundException;
import android.telephony.TelephonyManager;
import android.view.Window;
import android.view.WindowManager;
import android.widget.Toast;

public class AppMonitor extends Service {
    private static final int NUM_SATELLITES = 255;

    public final static String GLOBAL_TOUCH_EVENT =
"android.app.GLOBAL_TOUCH_EVENT";
    public final static String GLOBAL_FOCUS_EVENT =
"android.app.GLOBAL_FOCUS_EVENT";
    public final static String SCREENBRIGHTNESS_CHANGE_EVENT =
"android.app.SCREENBRIGHTNESS_CHANGE_EVENT";

    public final static String GLOBAL_CONFIG_EVENT =
"android.inputmethodservice.GLOBAL_CONFIG_EVENT";
    public final static String GLOBAL_KEY_EVENT =
"android.inputmethodservice.GLOBAL_KEY_EVENT";
    public final static String GLOBAL_SOFTKEY_EVENT =
"android.inputmethodservice.GLOBAL_SOFTKEY_EVENT";

    public final static String APPMONITOR_GET_DATABASE_INFO_EVENT =
"csu.research.AURA.APPMONITOR_GET_DATABASE_INFO_EVENT";
    public final static String APPMONITOR_DATABASE_INFO_EVENT =
"csu.research.AURA.APPMONITOR_DATABASE_INFO_EVENT";

    public final static int APPMONITOR_NOTIFICATION_ID = 12345;

    public final static int EVENT_AVERAGE_WINDOW = 6;

    public final static int NUM_EVENTS_CLASSIFIER = 10;

    private final IBinder mBinder = new AppMonitorBinder();
    public class AppMonitorBinder extends Binder {
        public AppMonitor getService() {
            return AppMonitor.this;
        }
    }

    private boolean isInitialized = false;
    private boolean isAlgorithmRunning = false;
    private DVFSControl dvfs;
    private boolean notificationsEnabled;
    private String currentFocusedAppName;
    private boolean isHardKeyboardOpen;
    private int touchEvents;
    private int keyEvents;
    private int numLocChanges;
    private long previousTouchEventTime;
    private long previousKeyEventTime;
    private long eventStartTime;
    private ApplicationDatabase appDatabase;
    private ApplicationInfo currentFocusedApp;
    private ArrayList<Long> deltaTouchEventTimeList = new ArrayList<Long>();
    private ArrayList<Long> deltaKeyEventTimeList = new ArrayList<Long>();
    private int startMinuteOfDay;
    private Thread mRunDVFSAlgorithm;
    private int mBatteryLevel = -1;

```

```

        private boolean mTrackBatteryLevelChanges = false;
        private ArrayList<Float> mAverageWindowedEventTime = new
ArrayList<Float>(EVENT_AVERAGE_WINDOW);
        private ArrayList<Float> mTimeAverageWindowedEventTime = new
ArrayList<Float>(EVENT_AVERAGE_WINDOW);

        // journal additions
        int numQStateDims = 2;
int numActions = 2;
int[] QState;
int[] newQState;
int action; // 1 = change state, 0 = don't
        double epsilon;
double temp;
double alpha;
double gamma;
double lambda;
double reward;
int epochs;
public int epochsdone;
Vector trace = new Vector();
int[] saPair;
boolean randomEps;

        @SuppressWarnings("unused")
        private static boolean resetBrightness = false;
        private static int highCount = 0;
        private static int lowCount = 0;
        private static int batteryLevelChanged = 0;
        private static float totalPowerSaved = 0F;
        private static float totalPowerSavedTime = 0F;
        private static float totalEnergySaved = 0F;
        private static float totalSOCSaved = 0F;
        private static float totalPowerNominal = 0F;
        private static float avgPowerSavedPct = 0F;
        private static float avgCpuUtil = 0F;
        private static int missCount = 0;
        private static int hitCount = 0;

        public static Toast LastStatsToast = null;
        public static ApplicationDatabase AppDatabase;
        public static DVFSControl DVFSController;
        public static ControlAlgorithm Algorithm = ControlAlgorithm.NORMAL_MDP;

        private PackageManager mPackageManager;
private NetInfo mNetInfo;
private PowerManager mPowerManager;
private LocationManager mLocationManager;
private ActivityManager mActivityManager;
private NotificationManager mNotificationManager;
private AudioManager mAudioM;
private Handler mLogHandler;
private boolean mThreadDone;
private int mCpuTotalPct;
private int mScreenOn;
private int mScreenBrightness;
private Thread mParamLogThread;
private AppClassification previousClassification;
private int unchangedClassCount;

        public IBinder onBind(Intent intent) {
            return mBinder;
        }

```

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);
    return START_STICKY;
}

public void setNotifications(boolean areEnabled) {
    notificationsEnabled = areEnabled;
}

public boolean areNotificationsEnabled() {
    return notificationsEnabled;
}

private void resetStats() {
    mCpuTotalPct = 0;
    mScreenOn = 0;
    mScreenBrightness = 0;

    previousClassification = AppClassification.NotClassified;
    unchangedClassCount = 0;

    touchEvents = 0;
    keyEvents = 0;
    previousKeyEventTime = 0L;
    previousTouchEventTime = 0L;
    deltaTouchEventTimeList = new ArrayList<Long>();
    deltaKeyEventTimeList = new ArrayList<Long>();
}

public ApplicationDatabase getApplicationDatabase() {
    return appDatabase;
}

public boolean isHardKeyboardOpen() {
    return isHardKeyboardOpen;
}

@Override
public void onLowMemory() {
    setForeground(true);
    Toast.makeText(getBaseContext(), "App Monitor Service Low Memory
Warning!", Toast.LENGTH_LONG).show();
    super.onLowMemory();
}

@Override
public void onDestroy() {
    Toast.makeText(getBaseContext(), "App Monitor Service Destroyed!",
Toast.LENGTH_LONG).show();
    if (!saveDatabase()) {
        Toast.makeText(getBaseContext(), "Error occurred saving
application database!", Toast.LENGTH_LONG);
    }
    shutdown();
    super.onDestroy();
}

@Override
public boolean onUnbind(Intent intent) {
    if (intent.getAction().equals("KILL_APP_MONITOR")) {
        stopSelf();
    }
}

```

```

        return true;
    }
    return super.onUnbind(intent);
}

@Override
public void onCreate() {
    if (!isInitialized) {
        initialize();
    }
    Toast.makeText(getBaseContext(), "App Monitor Service Created!",
Toast.LENGTH_LONG).show();
    AppDatabase = appDatabase;
    DVFSController = dvfs;
    Algorithm = ControlAlgorithm.NORMAL_MDP;
    super.onCreate();
}

private boolean saveDatabase() {
    try {
        if (appDatabase.isDirty() &&
Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath = String.format("%s/AppMonitor",
Environment.getExternalStorageDirectory());
            if (!new File(filePath).exists()) {
                new File(filePath).mkdirs();
            }
            if (new File(filePath).canWrite()) {
                return appDatabase.save(filePath);
            }
        }
        return false;
    }
    catch (Exception e) {
        return false;
    }
}

private boolean loadDatabase() {
    try {
        if (Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath =
String.format("%s/AppMonitor/AppStats.ser",
Environment.getExternalStorageDirectory());
            if (new File(filePath).exists()) {
                FileInputStream fis = new FileInputStream(filePath);
                ObjectInputStream in = new ObjectInputStream(fis);
                appDatabase = (ApplicationDatabase) in.readObject();
                in.close();
                return true;
            }
        }
        appDatabase = new ApplicationDatabase();
        return false;
    }
    catch (Exception e) {
        appDatabase = new ApplicationDatabase();
        return false;
    }
}

public boolean isAlgorithmActive() {

```

```

        return isAlgorithmRunning;
    }

    private void initialize() {
        if (!loadDatabase()) {
            Toast.makeText(getBaseContext(), "Error occurred loading default
database!", Toast.LENGTH_LONG).show();
        }

        mLogHandler = new Handler();
        mPackageManager = getPackageManager();
        mPowerManager = (PowerManager) getSystemService(Context.POWER_SERVICE);
        mActivityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
        WifiManager wifiManager = (WifiManager) getSystemService(WIFI_SERVICE);
        TelephonyManager cellManager = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        ConnectivityManager connectivityManager = (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);
        mAudioM = (AudioManager) getSystemService(AUDIO_SERVICE);
        mNetInfo = new NetInfo(wifiManager, connectivityManager, cellManager);
        mThreadDone = false;

        // journal additions
        QState = new int[numQStateDims];
        newQState = new int[numQStateDims];
        epsilon = 0.1;
        temp = 1;
        alpha = 1;
        gamma = 0.1;
        lambda = 0.1;
        randomEps = false;

        dvfs = new DVFSControl();
        resetDVFSThread();
        setForeground(true);
        notificationsEnabled = true;
        currentFocusedAppName = "";
        currentFocusedApp = new ApplicationInfo("", "");
        isHardKeyboardOpen = false;
        IntentFilter filt = new IntentFilter();
        filt.addAction(GLOBAL_CONFIG_EVENT);
        filt.addAction(GLOBAL_FOCUS_EVENT);
        filt.addAction(GLOBAL_KEY_EVENT);
        filt.addAction(GLOBAL_SOFTKEY_EVENT);
        filt.addAction(GLOBAL_TOUCH_EVENT);
        filt.addAction(Intent.ACTION_SCREEN_OFF);
        filt.addAction(Intent.ACTION_SCREEN_ON);
        filt.addAction(Intent.ACTION_BATTERY_CHANGED);
        registerReceiver(new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                setForeground(true);
                long currentTime = SystemClock.elapsedRealtime();
                startMinuteOfDay = (Calendar.HOUR_OF_DAY * 3600) +
Calendar.MINUTE;
                if (intent.getAction().equals(GLOBAL_CONFIG_EVENT)) {
                    isHardKeyboardOpen =
intent.getBooleanExtra("ishardkeyboardopen", false);
                }
            }
        });
    }

```

```

        else if (intent.getAction().equals(GLOBAL_FOCUS_EVENT) &&
intent.getBooleanExtra("hasfocus", false)) {
            String activity = intent.getStringExtra("activity");
            if (activity != null) {
                if (!currentFocusedAppName.equals(activity)) {
                    if (eventStartTime != 0L &&
!appDatabase.getIgnoredAppNames().contains(currentFocusedAppName)) {
                        if (isAlgorithmRunning) {
                            stopDVFSAlgorithm();
                            String low =
String.format("Low Frequency: %.1f %%", lowCount * 100F / Math.max((lowCount +
highCount), 1));
                            String high =
String.format("\nHigh Frequency: %.1f %%", highCount * 100F / Math.max((lowCount +
highCount), 1));
                            String misses =
String.format("\nMiss Ratio: %.1f %%", missCount * 100F / Math.max((missCount +
hitCount), 1));
                            String power =
String.format("\nPower Saved: %.2f mW", totalPowerSaved);
                            totalEnergySaved =
(totalPowerSaved * totalPowerSavedTime) / 1000F;
                            String energy =
String.format("\nEnergy Saved: %.2f J", totalEnergySaved);
                            avgPowerSavedPct =
totalPowerSaved*100 / totalPowerNominal;
                            String powerPct =
String.format("\nPower Avg Pct Saved: %.1f %%", avgPowerSavedPct);
                            String powerNominal =
String.format("\nTotal Power Nominal: %.2f mW", totalPowerNominal);
                            totalSOCsSaved =
BatteryModels.getPercentSOCsSaved(totalPowerSaved);
                            String soc =
String.format("\nSOC Saved: %.3f %%", totalSOCsSaved);
                            String cpuUtil =
String.format("\nAvg CPU Util: %.1f %%", avgCpuUtil / Math.max((lowCount + highCount),
1));
                            String batt =
String.format("\nBattery SOC Changed: %d %%", batteryLevelChanged);
                            String temp = low + high +
misses + power + powerNominal + energy + powerPct + soc + cpuUtil + batt;
                            LastStatsToast =
Toast.makeText(getApplicationContext(), temp, Toast.LENGTH_LONG);
                            LastStatsToast.show();
                        }
                        float touchMean =
calculateMean(deltaTouchEventTimeList);
                        float keyMean =
calculateMean(deltaKeyEventTimeList);
                        float touchSD =
calculateStandardDev(deltaTouchEventTimeList, touchMean);
                        float keySD =
calculateStandardDev(deltaKeyEventTimeList, keyMean);
                        float touchMedian =
calculateMedian(deltaTouchEventTimeList);
                        float keyMedian =
calculateMedian(deltaKeyEventTimeList);
                        int currentMinuteOfDay =
(Calendar.HOUR_OF_DAY * 3600) + Calendar.MINUTE;

                        currentFocusedApp.addTouchStats(touchMean, touchSD, touchMedian);

                        currentFocusedApp.addKeyStats(keyMean, keySD, keyMedian);

```

```

        currentFocusedApp.addUsageStats(startMinuteOfDay,          currentMinuteOfDay,
(float)(currentTime - eventStartTime) / 1000F);
        //currentFocusedApp.addMiscStats((float)(currentTime -
eventStartTime) / 1000F,
        //          mCpuTotalPct, mScreenOn, mScreenBrightness,
mWifiConnected, mWifiHasTraffic, mWifiTraffic,
        //          mOffHook, mDataHasTraffic, mDataTraffic,
mAudioPlaying, mBluetoothOn, mGpsEnabled);
        String          temp          =
currentFocusedAppName.toUpperCase();
        if (temp.contains(".")) {
            temp
temp.substring(temp.lastIndexOf(".") + 1, temp.length());
        }
        if (lowCount > 0 || highCount >
0) {
            float den = lowCount +
highCount;
            postNotification(String.format("%s - [%.1f, %.1f]!", temp, (lowCount * 100 /
den), highCount * 100 / den), R.drawable.monitor_icon);
            lowCount = 0; highCount =
0;
        }
        else {
            postNotification(String.format("%s Updated!", temp), R.drawable.monitor_icon);
        }

        // journal addition
        String          filePath          =
String.format("%s/AppMonitor/TouchTimeCounts_%.s.txt",
Environment.getExternalStorageDirectory(), temp);
        File file = new File(filePath);
        try {
            PrintWriter writer = new PrintWriter(file);
            int oneCount = 0;
            int twoCount = 0;
            int threeCount = 0;
            int fourCount = 0;
            int fiveCount = 0;
            int sixCount = 0;
            int sevenCount = 0;
            for (int i = 0; i <
deltaTouchEventTimeList.size(); i++) {
                long          touchTime          =
                if (touchTime > 0 &&
touchTime < 1500) {
                    oneCount++;
                }
                else if (touchTime < 2500) {
                    twoCount++;
                }
                else if (touchTime < 3500) {
                    threeCount++;
                }
                else if (touchTime < 4500) {
                    fourCount++;
                }
            }
            else if (touchTime < 5500) {

```



```

//                    fiveCount++;
//                }
//                else if (touchTime < 6500) {
//                    sixCount++;
//                }
//                else if (touchTime >= 6500) {
//                    sevenCount++;
//                }
//            }
//
//            writer.println(String.format("1000: %d", oneCount));
//
//            writer.println(String.format("2000: %d", twoCount));
//
//            writer.println(String.format("3000: %d", threeCount));
//
//            writer.println(String.format("4000: %d", fourCount));
//
//            writer.println(String.format("5000: %d", fiveCount));
//
//            writer.println(String.format("6000: %d", sixCount));
//
//            writer.println(String.format("7000+: %d", sevenCount));
//
//                    writer.close();
//                } catch (FileNotFoundException e) {
//                    // TODO Auto-generated catch block
//                    e.printStackTrace();
//                }
//
//                    // journal addition
//                    String filePath =
String.format("%s/AppMonitor/TouchTimeline_%s.txt",
Environment.getExternalStorageDirectory(), temp);
//                    File file = new File(filePath);
//                    try {
//                        PrintWriter writer = new PrintWriter(file);
//                        long touchTime = 0;
//
//                        for (int i = 0; i <
deltaTouchEventTimeList.size(); i++) {
//                            touchTime = touchTime +
//                            writer.println(String.format("%d",
touchTime));
//                        }
//                        writer.close();
//                    } catch (FileNotFoundException e) {
//                        // TODO Auto-generated catch block
//                        e.printStackTrace();
//                    }
//
//                }
//            }
//
//            String pkg =
intent.getStringExtra("package");
//
//            currentFocusedAppName = activity;
//            eventStartTime = currentTime;
//
//            if
(appDatabase.containsApplication(activity)) {
//                currentFocusedApp =
appDatabase.getApplicationInfo(activity);
//            }
//            else {

```

```

                                                                    currentFocusedApp    =    new
ApplicationInfo(activity, pkg);

                                                                    appDatabase.addUntrainedApp(currentFocusedApp);
                                                                    }

                                                                    resetStats();
                                                                    if (currentFocusedApp.isTrained()) {
                                                                    startDVFSAlgorithm();

mAverageWindowedEventTime.clear();
                                                                    }
                                                                    }
                                                                    }
                                                                    }
                                                                    else if (intent.getAction().equals(GLOBAL_TOUCH_EVENT)) {
                                                                    touchEvents++;
                                                                    if (previousTouchEventTime != 0L) {
                                                                    long dT = getDeltaTime(previousTouchEventTime);
                                                                    deltaTouchEventTimeList.add(dT);
                                                                    synchronized(mAverageWindowedEventTime) {
                                                                    if (mAverageWindowedEventTime.size() >=
EVENT_AVERAGE_WINDOW) {

                                                                    mAverageWindowedEventTime.remove(0);

                                                                    mAverageWindowedEventTime.add((float)dT);
                                                                    }
                                                                    else {

                                                                    mAverageWindowedEventTime.add((float)dT);
                                                                    }

                                                                    }

                                                                    float touchMean = calculateMean(deltaTouchEventTimeList);
                                                                    float touchSD = calculateStandardDev(deltaTouchEventTimeList, touchMean);
                                                                    if (currentFocusedApp.isTrained() == false) {
                                                                    bayesianClassifier(touchMean, touchSD, dT);
                                                                    }
                                                                    //
                                                                    //
                                                                    //
                                                                    //
                                                                    if (currentFocusedApp.isTrained() == false) {
                                                                    naiveNumEventClassifier();
                                                                    }

                                                                    }

                                                                    previousTouchEventTime = currentTime;
                                                                    interruptDVFSAlgorithm();
                                                                    }
                                                                    else if (intent.getAction().equals(GLOBAL_KEY_EVENT) ||
intent.getAction().equals(GLOBAL_SOFTKEY_EVENT)) {
                                                                    keyEvents++;
                                                                    if (previousTouchEventTime != 0L) {
                                                                    long dT = getDeltaTime(previousTouchEventTime);
                                                                    deltaTouchEventTimeList.add(dT);
                                                                    synchronized(mAverageWindowedEventTime) {
                                                                    if (mAverageWindowedEventTime.size() >=
EVENT_AVERAGE_WINDOW) {

                                                                    mAverageWindowedEventTime.remove(0);

                                                                    mAverageWindowedEventTime.add((float)dT);

```

```

        }
        else {

mAverageWindowedEventTime.add((float)dT);
        }
    }

    float keyMean = calculateMean(deltaKeyEventTimeList);
    float keySD = calculateStandardDev(deltaKeyEventTimeList,
keyMean);

    if (currentFocusedApp.isTrained() == false) {
        bayesianClassifier(keyMean, keySD, dT);
    }

//    if (currentFocusedApp.isTrained() == false) {
//        naiveNumEventClassifier();
//    }

        previousKeyEventTime = currentTime;
        interruptDVFSAlgorithm();
    }
    else if
(intent.getAction().equals(Intent.ACTION_SCREEN_OFF)) {
        String filePath =
String.format("%s/AppMonitor/AppDatabase.txt",
Environment.getExternalStorageDirectory());
        postNotification(String.format("Saved = %s, Exported
= %s", saveDatabase(), appDatabase.exportDatabase(filePath)), R.drawable.save_icon);
        setScreenOffDVFSLevel();
    }
    else if
(intent.getAction().equals(Intent.ACTION_SCREEN_ON)) {
        if (!isAlgorithmRunning) {
            if (currentFocusedApp != null &&
currentFocusedApp.isTrained()) {
                startDVFSAlgorithm();
            }
            else {
                setScreenOnDVFSLevel();
            }
        }
    }
    else if
(intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) {
        int prev = mBatteryLevel;
        mBatteryLevel =
intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
        if (prev > 0 && mTrackBatteryLevelChanges) {
            batteryLevelChanged += (prev - mBatteryLevel);
        }
    }
}
}, filt);
isInitialized = true;
}

private long getDeltaTime(long time) {
    return SystemClock.elapsedRealtime() - time;
}

private float calculateMean(ArrayList<Long> list) {
    long sum = 0;
    if (list.size() > 0) {

```

```

        for (int i = 0; i < list.size(); i++) {
            sum += list.get(i);
        }
        return (float)sum / list.size();
    }
    return 0F;
}

private float calculateStandardDev(ArrayList<Long> list, float mean) {
    float sum = 0F;
    if (list.size() > 0) {
        for (int i = 0; i < list.size(); i++) {
            sum += Math.pow((list.get(i) - mean), 2);
        }
        return (float)Math.sqrt(sum / list.size());
    }
    return 0F;
}

private float calculateMedian(ArrayList<Long> list) {
    ArrayList<Long> tempList = new ArrayList<Long>(list);
    if (tempList.size() > 1 && (tempList.size() % 2) == 0) { //Even number of
points
        java.util.Collections.sort(tempList);
        float first = tempList.get(tempList.size() / 2 - 1);
        float second = tempList.get(tempList.size() / 2);
        return (first + second) / 2;
    }
    else if (tempList.size() > 1) { //Odd number of points
        java.util.Collections.sort(tempList);
        return tempList.get(tempList.size() / 2);
    }
    else if (tempList.size() == 1) {
        return tempList.get(0);
    }
    return 0F;
}

private void shutdown() {
    if (isAlgorithmRunning) {
        stopDVFSAlgorithm();
    }
    if (dvfs != null) {
        dvfs.setFrequencyGovernor(FrequencyGovernors.OnDemand);
    }
    if (mThreadDone == false) {
        mThreadDone = true;
    }
    if (mNotificationManager != null) {
        mNotificationManager.cancel(APPMONITOR_NOTIFICATION_ID);
    }
}

private void postNotification(String notificationText, int icon) {
    try {
        if (notificationsEnabled) {
            // In this sample, we'll use the same text for the ticker and the
expanded notification
            CharSequence text = "App Monitor";

            // Set the icon, scrolling text and time stamp
            Notification notification = new Notification(icon, text,
System.currentTimeMillis());

```

```

        notification.flags |= Notification.FLAG_AUTO_CANCEL |
Notification.FLAG_ONGOING_EVENT;

        Intent notifyIntent = new Intent(this, AppMonitor.class);
        Bundle bundle = new Bundle();
        bundle.putString("action", "view");
        notifyIntent.putExtras(bundle);

        //notifyIntent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
Intent.FLAG_ACTIVITY_SINGLE_TOP);
        notifyIntent.setAction("csu.research.AURA.AppMonitor");

        // The PendingIntent to launch our activity if the user selects
this notification
        PendingIntent contentIntent = PendingIntent.getService(this, 0,
notifyIntent, 0);

        // Set the info for the views that show in the notification
panel.
        notification.setLatestEventInfo(this, "App Monitor",
notificationText, contentIntent);

        // Send the notification.
        // We use a string id because it is a unique number. We use it
later to cancel.
        mNotificationManager.notify(APPMONITOR_NOTIFICATION_ID,
notification);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

private void startDVFSAlgorithm() {
    if (mRunDVFSAlgorithm == null || isAlgorithmRunning ||
mRunDVFSAlgorithm.isAlive()) {
        interruptDVFSAlgorithm();
        mRunDVFSAlgorithm.interrupt();
        resetDVFSThread();
        mRunDVFSAlgorithm.start();
    }
    else {
        mRunDVFSAlgorithm.start();
    }
}

private void stopDVFSAlgorithm() {
    if (isAlgorithmRunning || mRunDVFSAlgorithm.isAlive()) {
        isAlgorithmRunning = false;
        mTrackBatteryLevelChanges = false;
        interruptDVFSAlgorithm();
        mRunDVFSAlgorithm.interrupt();
        resetDVFSThread();
    }
}

private void interruptDVFSAlgorithm() {
    if (isAlgorithmRunning) {
        synchronized(mRunDVFSAlgorithm) {
            mRunDVFSAlgorithm.notify();
            resetBrightness = true;
        }
    }
}

```

```

    }
}

private void resetDVFSThread() {
    mRunDVFSAlgorithm = new Thread(new Runnable() {
        public void run() {
            if (currentFocusedApp != null &&
currentFocusedApp.isTrained()) {
                try {
                    lowCount = 0; highCount = 0;
                    totalPowerSaved = 0F; totalEnergySaved = 0F;
                    avgPowerSavedPct = 0F; totalPowerSavedTime =
0F;
                    totalPowerNominal = 0F;
                    batteryLevelChanged = 0;
                    mTrackBatteryLevelChanges = true;
                    hitCount = 0; missCount = 0;
                    avgCpuUtil = 0;
                    switch (Algorithm) {
                        case POWERSAVER:
                            isAlgorithmRunning = true;
                            runPowerSaverAlgorithm();
                            break;
                        case CHANGE_BLINDNESS:
                            isAlgorithmRunning = true;
                            runChangeBlindnessAlgorithm();
                            break;
                        case Q_LEARNING:
                            isAlgorithmRunning = true;
                            runQLearningAlgorithm();
                            break;
                        case NORMAL_MDP:
                            isAlgorithmRunning = true;
                            runNormalMDPAlgorithm();
                            break;
                        case NORMAL_MDP_ADAPT:
                            isAlgorithmRunning = true;
                            runAdaptiveMDPAlgorithm();
                            break;
                        case MOVING_AVERAGE:
                            isAlgorithmRunning = true;
                            runMovingAverageAlgorithm();
                            break;
                        default:
                            Toast.makeText(getApplicationContext(), String.format("%s is not classified!",
currentFocusedAppName), Toast.LENGTH_LONG).show();
                            stopDVFSAlgorithm();
                            dvfs.setNominalCPUFrequency();
                    }
                }
            }
            catch (InterruptedException e){
                isAlgorithmRunning = false;
                dvfs.setNominalCPUFrequency();
            }
            catch (Exception e){
                isAlgorithmRunning = false;
                dvfs.setNominalCPUFrequency();
            }
        }
    }
}
}

```

```

    });
}

private void runPowerSaverAlgorithm() throws InterruptedException {
    if (dvfs != null && (dvfs.getCurrentGovernor()
FrequencyGovernors.Userspace
dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
        State state = State.HIGH;
        long basePeriod = 1000;
        int nominalFreq = 729600; // changed for Nexus One
        int lowFreq = 245000; // changed for Nexus One
        int curFreq = dvfs.getCPUFrequency();
        int prevEventCount = touchEvents + keyEvents;
        float adjustScreenDown = 0.1F;
        float defaultBrightness = getDefaultScreenBrightness();
        float currentBrightness = defaultBrightness;
        float maxBrightness = defaultBrightness;
        float minBrightness = 0.4F;
        Intent brightnessIntent = new Intent();
        brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
        int cpuUtil = 0;
        long timeSinceStartup = 0;
        int defaultBrightnessPct = (int)defaultBrightness*100;
        int notificationCount = 0;
        float mean, dev;
        int lowIntervalCount = 0;
        int highStateCount = 0, highStateLimit = 0;

        if (currentFocusedApp.getKeyMean() < currentFocusedApp.getTouchMean()) {
            mean = currentFocusedApp.getKeyMean();
            dev = currentFocusedApp.getKeyStandardDeviation();
            if (mean <= 0) {
                mean = currentFocusedApp.getTouchMean();
                dev = currentFocusedApp.getTouchStandardDeviation();
            }
        }
        else {
            mean = currentFocusedApp.getTouchMean();
            dev = currentFocusedApp.getTouchStandardDeviation();
            if (mean <= 0) {
                mean = currentFocusedApp.getKeyMean();
                dev = currentFocusedApp.getKeyStandardDeviation();
            }
        }

        switch(currentFocusedApp.getAppClassification()) {
            // journal changes
            case VeryLowInteraction:
                basePeriod = 1000;
                adjustScreenDown = 0.4F;
                highStateLimit = 2;
            case LowInteraction:
                basePeriod = 850;
                adjustScreenDown = 0.35F;
                highStateLimit = 3;
                break;
            case LowMedInteraction:
                basePeriod = 700;
                adjustScreenDown = 0.3F;
                highStateLimit = 4;
            case MedInteraction:
                basePeriod = 550;
                adjustScreenDown = 0.25F;

```

```

        highStateLimit = 5;
        break;
    case MedHighInteraction:
        basePeriod = 400;
        adjustScreenDown = 0.2F;
        highStateLimit = 6;
    case HighInteraction:
        basePeriod = 250;
        adjustScreenDown = 0.15F;
        highStateLimit = 7;
        break;
    case VeryHighInteraction:
        basePeriod = 100;
        adjustScreenDown = 0.05F;
        highStateLimit = 8;
}

while (isAlgorithmRunning) {
    //DVFS
    if ((touchEvents + keyEvents) > prevEventCount) {
        prevEventCount = touchEvents + keyEvents;
        if (state == State.LOW) {
            missCount++;
        }
        else {
            hitCount++;
        }
    }
    if (lowIntervalCount >= mean) {
        if (curFreq != nominalFreq) {
            dvfs.setNominalCPUFrequency();
            curFreq = nominalFreq;
        }
        state = State.HIGH;
        lowIntervalCount = 0;
        highStateCount = 0;
    }
    else if (state == State.HIGH && highStateCount++ >= highStateLimit) {
        if (curFreq != lowFreq) {
            dvfs.setLowestCPUFrequency();
            curFreq = lowFreq;
        }
        state = State.LOW;
    }
    //SCREEN
    switch(state) {
        case LOW:
            if (currentBrightness > minBrightness) {
                currentBrightness = AppUtil.saturate(currentBrightness -
adjustScreenDown, minBrightness, maxBrightness);
                brightnessIntent.putExtra("brightness",
currentBrightness);
                sendBroadcast(brightnessIntent);
            }
            lowCount++;
            break;
        case HIGH:
            if (currentBrightness < defaultBrightness) {
                currentBrightness = defaultBrightness;
                brightnessIntent.putExtra("brightness",
currentBrightness);
                sendBroadcast(brightnessIntent);
            }
    }
}

```



```

        highCount++;
        break;
    }

    synchronized(mRunDVFSAlgorithm) {
        mRunDVFSAlgorithm.wait(basePeriod);
    }

    lowIntervalCount += basePeriod;

    cpuUtil = CPUInfo.getCPUUtilizationPct();
    avgCpuUtil += cpuUtil;

    totalPowerNominal += PowerModels.getMWCPUPower(nominalFreq, cpuUtil);
    totalPowerNominal +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

    if (curFreq < nominalFreq || currentBrightness < defaultBrightness) {
        totalPowerSavedTime += (basePeriod / 1000F);

        if (currentBrightness < defaultBrightness) {
            totalPowerSaved +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
        }
        if (curFreq < nominalFreq) {
            totalPowerSaved +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
        }
    }

    if (notificationCount++ >= 10) {
        postNotification(String.format("[%.3f      mW,      %.3f      %%]",
totalPowerSaved, totalSOCSaved) , R.drawable.cpu_icon);
        notificationCount = 0;
    }
}
}

private void runChangeBlindnessAlgorithm() throws InterruptedException {
    if (dvfs != null && (dvfs.getCurrentGovernor() ==
FrequencyGovernors.Userspace ||
dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
        State state = State.HIGH;
        long basePeriod = 1000;
        int nominalFreq = 729600; // changed for Nexus One
        int lowFreq = 691200; // changed for Nexus One
        int curFreq = dvfs.getCPUFrequency();
        int prevEventCount = touchEvents + keyEvents;
        float adjustScreenDown = 0.07F;
        float defaultBrightness = getDefaultScreenBrightness();
        float currentBrightness = defaultBrightness;
        float maxBrightness = defaultBrightness;
        float minBrightness = 0.8F;
        Intent brightnessIntent = new Intent();
        brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
        float totalPowerSavedTime = 0F;
        int cpuUtil = 0;
        long timeSinceStartup = 0;
        int defaultBrightnessPct = (int)defaultBrightness*100;

        int dvfsCountDown = 30000; //Milliseconds
        int screenCountDown = 40000; //Milliseconds
    }
}

```

```

int adjustScreenEveryNms = 3000; //Milliseconds
int adjustDVFSEveryNms = 4000; //Milliseconds
int screenCounter = 0;
int dvfsCounter = 0;
int cpuUtilLowToHighThresh = 70; //%
int cpuUtilHighToLowThresh = 70; //%
int notificationCount = 0;

int resetInterval = 30000; //Milliseconds
int resetCounter = 0;

switch(currentFocusedApp.getAppClassification()) {
    // journal changes
    case VeryLowInteraction:
        basePeriod = 1000;
        resetInterval = 30000;
    case LowInteraction:
        basePeriod = 850;
        resetInterval = 27000;
        break;
    case LowMedInteraction:
        basePeriod = 700;
        resetInterval = 24000;
    case MedInteraction:
        basePeriod = 550;
        resetInterval = 20000;
        break;
    case MedHighInteraction:
        basePeriod = 400;
        resetInterval = 17000;
    case HighInteraction:
        basePeriod = 250;
        resetInterval = 14000;
        break;
    case VeryHighInteraction:
        basePeriod = 100;
        resetInterval = 10000;
}

while (isAlgorithmRunning) {
    //MAY WANT TO CHANGE HOW WE DO STATS.....
    //STATS
    hitCount++;

    if ((touchEvents + keyEvents) > prevEventCount) {
        if (state == State.LOW)
        {
            missCount++; //Missed, so adjust hit
            hitCount--;
        }
        prevEventCount = touchEvents + keyEvents;
    }

    //Update CPU Frequency Minimum Level
    if (dvfsCountDown > 0) {
        dvfsCountDown -= basePeriod;
        dvfsCounter += basePeriod;
        if (dvfsCounter >= adjustDVFSEveryNms) {
            dvfsCounter = 0;
            lowFreq = (int)Math.max((0.7F + 0.3F * (dvfsCountDown /
40000F)) * nominalFreq, 499200);
            // step down levels changed for Nexus One
            if (lowFreq >= 691200) {

```

```

        lowFreq = 691200;
    }
    else if (lowFreq >= 652800) {
        lowFreq = 652800;
    }
    else if (lowFreq >= 614400) {
        lowFreq = 614400;
    }
    else if (lowFreq >= 576000) {
        lowFreq = 576000;
    }
    else if (lowFreq >= 537600) {
        lowFreq = 537600;
    }
    else {
        lowFreq = 499200;
    }
}

//Update Screen Minimum Level
if (screenCountDown > 0) {
    screenCountDown -= basePeriod;
    screenCounter += basePeriod;
    if (screenCounter >= adjustScreenEveryNms) {
        screenCounter = 0;
        currentBrightness = AppUtil.saturate(currentBrightness -
adjustScreenDown, minBrightness, maxBrightness);
        brightnessIntent.putExtra("brightness", currentBrightness);
        sendBroadcast(brightnessIntent);
    }
}

//Reset cpu min freq & screen brightness every reset interval
resetCounter += basePeriod;
if (resetCounter >= resetInterval) {
    resetCounter = 0;

    lowFreq = 691200; // changed for Nexus One

    currentBrightness = defaultBrightness;
    brightnessIntent.putExtra("brightness", currentBrightness);
    sendBroadcast(brightnessIntent);

    dvfsCountDown = 30000; //Milliseconds
    screenCountDown = 40000; //Milliseconds
    screenCounter = 0;
    dvfsCounter = 0;
}

//GET CPU UTILIZATION
cpuUtil = CPUInfo.getCPUUtilizationPct();
avgCpuUtil += cpuUtil;

switch(state) {
    case LOW:
        if (cpuUtil > cpuUtilLowToHighThresh) {
            dvfs.setNominalCPUFrequency();
            state = State.HIGH;
            curFreq = 729600;
        }
        lowCount++;
        break;
}

```

```

        case HIGH:
            if (cpuUtil < cpuUtilHighToLowThresh) {
                dvfs.setCPUFrequency(lowFreq);
                state = State.LOW;
                curFreq = lowFreq;
            }
            highCount++;
            break;
    }

    Thread.sleep(basePeriod);

    totalPowerNominal += PowerModels.getMWCPUPower(nominalFreq, cpuUtil);
    totalPowerNominal +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

    if (curFreq < nominalFreq || currentBrightness < defaultBrightness) {
        totalPowerSavedTime += (basePeriod / 1000F);

        if (currentBrightness < defaultBrightness) {
            totalPowerSaved +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
        }
        if (curFreq < nominalFreq) {
            totalPowerSaved +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
        }
    }

    timeSinceStartup += basePeriod;

    if (notificationCount++ >= 10) {
        postNotification(String.format("[%0.3f      mW,      %0.3f      %%]",
totalPowerSaved, totalSOCSaved) , R.drawable.cpu_icon);
        notificationCount = 0;
    }
}
}

// journal addition
private int selectEGreedyAction( int[] Qstate ) {
    // epsilon greedy algorithm
    double[] qValues = currentFocusedApp.getQValuesAt( Qstate );
    int selectedAction = -1;
    randomEps = false;
    double maxQ = -Double.MAX_VALUE;
    int[] doubleValues = new int[qValues.length];
    int maxDV = 0;

    //Explore
    if ( Math.random() < epsilon ) {
        selectedAction = -1;
        randomEps = true;
    }
    else {
        for( int action = 0 ; action < qValues.length ; action++ ) {

            if( qValues[action] > maxQ ) {
                selectedAction = action;
                maxQ = qValues[action];
                maxDV = 0;
                doubleValues[maxDV] = selectedAction;
            }
        }
    }
}

```

```

    }
    else if( qValues[action] == maxQ ) {
        maxDV++;
        doubleValues[maxDV] = action;
    }
}

if( maxDV > 0 ) {
    int randomIndex = (int) ( Math.random() * ( maxDV + 1 ) );
    selectedAction = doubleValues[ randomIndex ];
}

// Select random action if all qValues == 0 or exploring.
if ( selectedAction == -1 ) {
    // System.out.println( "Exploring ..." );
    selectedAction = (int) (Math.random() * qValues.length);
}

return selectedAction;
}

/* private double getMaxQValue( int[] state, int action ) {

double maxQ = 0;

double[] qValues = policy.getQValuesAt( state );

for( action = 0 ; action < qValues.length ; action++ ) {
    if( qValues[action] > maxQ ) {
        maxQ = qValues[action];
    }
}
return maxQ;
}
*/

// journal addition
private void runQLearningAlgorithm() throws InterruptedException {
    if (dvfs != null && (dvfs.getCurrentGovernor() ==
FrequencyGovernors.Userspace
dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
        long basePeriod = 1000; //Default milliseconds
        int nominalFreq = 729600; // changed for Nexus One
        int lowFreq = 245000; // changed for Nexus One
        int lowNominalFreq = 537600; // changed for Nexus One
        int curFreq = dvfs.getCPUFrequency();
        switch(currentFocusedApp.getAppClassification()) {
            case VeryLowInteraction:
                basePeriod = 1000;
                break;
            case LowInteraction:
                basePeriod = 850;
                break;
            case LowMedInteraction:
                basePeriod = 700;
                break;
            case MedInteraction:
                basePeriod = 550;
                break;
            case MedHighInteraction:
                basePeriod = 400;
                break;

```

```

        case HighInteraction:
            basePeriod = 250;
            break;
        case VeryHighInteraction:
            basePeriod = 100;
            break;
    }
    if (curFreq != nominalFreq) {
        dvfs.setNominalCPUFrequency();
        curFreq = nominalFreq;
    }

    float mean, dev;
    int cpuUtil = 0;
    float adjustScreenDown = 0.1F;
    if (currentFocusedApp.getKeyMean() < currentFocusedApp.getTouchMean()) {
        mean = currentFocusedApp.getKeyMean();
        dev = currentFocusedApp.getKeyStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getTouchMean();
            dev = currentFocusedApp.getTouchStandardDeviation();
        }
    }
    else {
        mean = currentFocusedApp.getTouchMean();
        dev = currentFocusedApp.getTouchStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getKeyMean();
            dev = currentFocusedApp.getKeyStandardDeviation();
        }
    }
    float cov = dev / mean;
    if (cov < 0.5) { //Low variability
        adjustScreenDown = 0.15F;
    }
    else if (cov < 0.75) { //Med variability
        adjustScreenDown = 0.10F;
    }
    else if (cov > 1) { //High variability
        adjustScreenDown = 0.05F;
    }
    int previousTouchKeyEvents;
    long timeSinceStartup = 0;
    Intent brightnessIntent = new Intent();
    brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
    float currentBrightness = getDefaultScreenBrightness();
    float defaultBrightness = getDefaultScreenBrightness();
    int defaultBrightnessPct = (int)defaultBrightness*100;
    int notificationCount = 0;

    double this_Q;
    double max_Q;
    double new_Q;

    // Reset state to start state (nominal, eval interval 0)
    QState[0] = 1;
    QState[1] = 0;

    if (dev > 0 && mean > 0) {
        previousTouchKeyEvents = touchEvents + keyEvents;

        while(isAlgorithmRunning) {
            //for( int i = 0 ; i < epochs ; i++ ) {

```

```

// get action
action = selectEGreedyAction( QState );

// get next state
if (action == 1) {
    if (QState[0] == 0) {
        newQState[0] = 1;    // change to nominal
    }
    else {
        newQState[0] = 0;    // change to below nominal
    }
}
else {
    newQState[0] = QState[0];    // stay the same state
}
if (QState[1] < 10) {
    newQState[1] = QState[1] + 1;    // increment eval interval
count
}
else {
    newQState[1] = QState[1];    // don't increment
}

// calculate reward
// reward is 0 if we don't change state and no touch is received,
// 0 if we do change state and a touch is received,
// -1 for all other scenarios
if ((touchEvents + keyEvents) > previousTouchKeyEvents) {
    newQState[1] = 0;    // touch received, so reset counter

    if (newQState[0] == 0) {    // if predicted the low nominal
state,
        missCount++; // missed
        reward = -1; // predicted incorrectly, so penalize
    }
    else if (newQState[0] == 1) {    // if we predicted the nominal
state,
        hitCount++; // hit
        reward = 2; // predicted correctly, so reward
    }
    previousTouchKeyEvents = touchEvents + keyEvents;
}
else {
    if (newQState[0] == 0) {    // if we predicted the below
nominal state,
        hitCount++; // hit
        reward = 0; // predicted correctly, so don't penalize
    }
    else if (newQState[0] == 1) {
        reward = -1; // predicted incorrectly, so penalize
    }
}

// update screen and CPU freq
if (QState[0] == 0) {
    if (newQState[0] == 0) {
        // stay in below nominal state
        // adjust screen down
        if (currentBrightness > 0.4F) {
adjustScreenDown;
            currentBrightness = currentBrightness -
currentBrightness);
            brightnessIntent.putExtra("brightness",

```

```

        sendBroadcast(brightnessIntent);
    }
}
else {
    // change from below nominal to nominal
    // set brightness to default level
    if (currentBrightness != defaultBrightness) {
        currentBrightness = defaultBrightness;
        brightnessIntent.putExtra("brightness",
currentBrightness);

        sendBroadcast(brightnessIntent);
    }

    // set CPU frequency to nominal frequency
    if (curFreq != nominalFreq) {
        dvfs.setNominalCPUFrequency();
        curFreq = nominalFreq;
    }
}
lowCount++;
}
else {
    if (newQState[0] == 0) {
        // change from nominal to below nominal
        // adjust screen down
        if (currentBrightness > 0.4F) {
            currentBrightness = currentBrightness -
adjustScreenDown;

            brightnessIntent.putExtra("brightness",
currentBrightness);

            sendBroadcast(brightnessIntent);
        }

        // set low CPU frequency
        if (curFreq != lowFreq) {
            dvfs.setLowestCPUFrequency();
            curFreq = lowFreq;
        }
    }
    else {
        // stay in nominal state (do nothing)
    }
    highCount++;
}

this_Q = currentFocusedApp.getQValue( QState, action );
max_Q = currentFocusedApp.getMaxQValue( newQState );

// Calculate new Value for Q
new_Q = this_Q + alpha * ( reward + gamma * max_Q - this_Q );
currentFocusedApp.setQValue( QState, action, new_Q );

// Set state to the new state.
QState[0] = newQState[0];
QState[1] = newQState[1];

synchronized(mRunDVFSAlgorithm) {
    mRunDVFSAlgorithm.wait(basePeriod);
}

//curFreq = dvfs.getCPUFrequency();
totalPowerNominal += PowerModels.getMWCPUPower(nominalFreq,
cpuUtil);

```



```

        totalPowerNominal                                                     +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

        if (curFreq < nominalFreq || currentBrightness <
defaultBrightness) {
            totalPowerSavedTime += (basePeriod / 1000F);

            if (currentBrightness < defaultBrightness) {
                totalPowerSaved                                               +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
            }
            if (curFreq < nominalFreq) {
                totalPowerSaved                                               +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
            }
        }

        timeSinceStartup += basePeriod;

        if (notificationCount++ >= 10) {
            postNotification(String.format("[%].3f      mW,      %.3f      %%]",
totalPowerSaved, totalSOCSaved) , R.drawable.cpu_icon);
            notificationCount = 0;
        }
    }
}
else {
    postNotification("STAT ERROR", R.drawable.cpu_icon);
}
}
}
}

```

```

private void runNormalMDPAlgorithm() throws InterruptedException {
    if (dvfs != null && (dvfs.getCurrentGovernor() == FrequencyGovernors.Userspace
|| dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
        long basePeriod = 1000; //Default milliseconds
        float lowToHighProb = 0.7F;
        float lowStepUpProb = 0.6F;
        float highToLowProb = 0.5F;
        int nominalFreq = 729600; // changed for Nexus One
        int lowFreq = 245000; // changed for Nexus One
        int lowNominalFreq = 537600; // changed for Nexus One
        int curFreq = dvfs.getCPUFrequency();
        switch(currentFocusedApp.getAppClassification()) {
            // journal changes
            case VeryLowInteraction:
                basePeriod = 1000;
                lowToHighProb = 0.75F;
                lowStepUpProb = 0.65F;
                highToLowProb = 0.55F;
            case LowInteraction:
                basePeriod = 850;
                lowToHighProb = 0.7F;
                lowStepUpProb = 0.6F;
                highToLowProb = 0.5F;
                break;
            case LowMedInteraction:
                basePeriod = 700;
                lowToHighProb = 0.65F;
                lowStepUpProb = 0.55F;
                highToLowProb = 0.45F;
            case MedInteraction:
                basePeriod = 550;
        }
    }
}

```

```

        lowToHighProb = 0.6F;
        lowStepUpProb = 0.5F;
        highToLowProb = 0.4F;
        break;
    case MedHighInteraction:
        basePeriod = 400;
        lowToHighProb = 0.55F;
        lowStepUpProb = 0.45F;
        highToLowProb = 0.35F;
    case HighInteraction:
        basePeriod = 250;
        lowToHighProb = 0.5F;
        lowStepUpProb = 0.4F;
        highToLowProb = 0.3F;
    case VeryHighInteraction:
        basePeriod = 100;
        lowToHighProb = 0.45F;
        lowStepUpProb = 0.35F;
        highToLowProb = 0.25F;
        break;
}
float probOfEvent = 0F;
if (curFreq != nominalFreq) {
    dvfs.setNominalCPUFrequency();
    curFreq = nominalFreq;
}
State state = State.HIGH;
float mean, dev;
int cpuUtil = 0;
float adjustScreenDown = 0.1F;
if (currentFocusedApp.getKeyMean() < currentFocusedApp.getTouchMean()) {
    mean = currentFocusedApp.getKeyMean();
    dev = currentFocusedApp.getKeyStandardDeviation();
    if (mean <= 0) {
        mean = currentFocusedApp.getTouchMean();
        dev = currentFocusedApp.getTouchStandardDeviation();
    }
}
else {
    mean = currentFocusedApp.getTouchMean();
    dev = currentFocusedApp.getTouchStandardDeviation();
    if (mean <= 0) {
        mean = currentFocusedApp.getKeyMean();
        dev = currentFocusedApp.getKeyStandardDeviation();
    }
}
float cov = dev / mean;
if (cov < 0.5) { //Low variability
    adjustScreenDown = 0.15F;
}
else if (cov < 0.75) { //Med variability
    adjustScreenDown = 0.10F;
}
else if (cov > 1) { //High variability
    adjustScreenDown = 0.05F;
}
int previousTouchKeyEvents;
long timeSinceLastEvent = 0;
long timeSinceStartup = 0;
Intent brightnessIntent = new Intent();
brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
float currentBrightness = getDefaultScreenBrightness();
float defaultBrightness = getDefaultScreenBrightness();

```

```

int defaultBrightnessPct = (int)defaultBrightness*100;
int notificationCount = 0;
float diff;

if (dev > 0 && mean > 0) {
    previousTouchKeyEvents = touchEvents + keyEvents;

    while(isAlgorithmRunning) {
        //hitCount++; //Predict hit
        //SCREEN
        if ((touchEvents + keyEvents) > previousTouchKeyEvents) {
            if (state == State.LOW) {
                missCount++; //Missed, so adjust hit
                //hitCount--;
            }
            else {
                hitCount++; // CHANGED FOR PERFORMANCE ANALYSIS - NO HIT
                IF NO TOUCH EVENT
            }
            previousTouchKeyEvents = touchEvents + keyEvents;
            //currentBrightness = Math.min(currentBrightness +
adjustScreenUp, defaultBrightness);
            if (currentBrightness != defaultBrightness) {
                currentBrightness = defaultBrightness;
                brightnessIntent.putExtra("brightness",
currentBrightness);
                sendBroadcast(brightnessIntent);
            }
            timeSinceLastEvent = timeSinceStartup;
        }
        else {
            if (state == State.LOW) {
                if (currentBrightness > 0.4F) {
                    currentBrightness = currentBrightness -
adjustScreenDown;
                    brightnessIntent.putExtra("brightness",
currentBrightness);
                    sendBroadcast(brightnessIntent);
                }
            }
        }
        //DVFS
        cpuUtil = CPUInfo.getCPUUtilizationPct();
        avgCpuUtil += cpuUtil;

        diff = timeSinceStartup - timeSinceLastEvent;
        switch (state) {
            case LOW:
                probOfEvent = AppUtil.getNormProb(diff, mean, dev, 100);
                if (probOfEvent > lowToHighProb || diff == 0) {
                    state = State.HIGH;
                    if (curFreq != nominalFreq) {
                        dvfs.setNominalCPUFrequency();
                        curFreq = nominalFreq;
                    }
                    currentBrightness = defaultBrightness;
                    brightnessIntent.putExtra("brightness",
currentBrightness);
                    sendBroadcast(brightnessIntent);
                }
            else if (probOfEvent > lowStepUpProb || cpuUtil > 80) {
                if (curFreq != lowNominalFreq) {
                    dvfs.setLowNominalCPUFrequency();

```

```

        curFreq = lowNominalFreq;
    }
}
lowCount++;
break;
case HIGH:
    if (diff > 0) {
        probOfEvent = AppUtil.getNormProb(diff, mean, dev,
100);

        if (probOfEvent < highToLowProb && cpuUtil < 80) {
            state = State.LOW;
            if (curFreq != lowFreq) {
                dvfs.setLowestCPUFrequency();
                curFreq = lowFreq;
            }
        }
        highCount++;
        break;
    }

    synchronized(mRunDVFSAlgorithm) {
        mRunDVFSAlgorithm.wait(basePeriod);
    }

    //curFreq = dvfs.getCPUFrequency();
    totalPowerNominal += PowerModels.getMWCPUPower(nominalFreq,
cpuUtil);
    totalPowerNominal +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

    if (curFreq < nominalFreq || currentBrightness <
defaultBrightness) {
        totalPowerSavedTime += (basePeriod / 1000F);

        if (currentBrightness < defaultBrightness) {
            totalPowerSaved +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
        }
        if (curFreq < nominalFreq) {
            totalPowerSaved +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
        }
    }

    timeSinceStartup += basePeriod;

    if (notificationCount++ >= 10) {
        postNotification(String.format("[%0.3f mW, %0.3f %%]",
totalPowerSaved, totalSOCSSaved) , R.drawable.cpu_icon);
        notificationCount = 0;
    }
}
else {
    postNotification("STAT ERROR", R.drawable.cpu_icon);
}
}
}

private void runAdaptiveMDPAlgorithm() throws InterruptedException {

```

```

        if      (dvfs      !=      null      &&      (dvfs.getCurrentGovernor()      ==
FrequencyGovernors.Userspace
dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
    long basePeriod = 1000; //Default milliseconds
    float lowToHighProb = 0.7F;
    float lowStepUpProb = 0.6F;
    float highToLowProb = 0.5F;
    int nominalFreq = 729600;    // changed for Nexus One
    int lowFreq = 245000;        // changed for Nexus One
    int lowNominalFreq = 537600;    // changed for Nexus One
    int curFreq = dvfs.getCPUFrequency();
    switch(currentFocusedApp.getAppClassification()) {
        // journal changes
        case VeryLowInteraction:
            basePeriod = 1000;
            lowToHighProb = 0.75F;
            lowStepUpProb = 0.65F;
            highToLowProb = 0.55F;
        case LowInteraction:
            basePeriod = 850;
            lowToHighProb = 0.7F;
            lowStepUpProb = 0.6F;
            highToLowProb = 0.5F;
            break;
        case LowMedInteraction:
            basePeriod = 700;
            lowToHighProb = 0.65F;
            lowStepUpProb = 0.55F;
            highToLowProb = 0.45F;
        case MedInteraction:
            basePeriod = 550;
            lowToHighProb = 0.6F;
            lowStepUpProb = 0.5F;
            highToLowProb = 0.4F;
            break;
        case MedHighInteraction:
            basePeriod = 400;
            lowToHighProb = 0.55F;
            lowStepUpProb = 0.45F;
            highToLowProb = 0.35F;
        case HighInteraction:
            basePeriod = 250;
            lowToHighProb = 0.5F;
            lowStepUpProb = 0.4F;
            highToLowProb = 0.3F;
        case VeryHighInteraction:
            basePeriod = 100;
            lowToHighProb = 0.45F;
            lowStepUpProb = 0.35F;
            highToLowProb = 0.25F;
            break;
    }
    float probOfEvent = 0F;
    if (curFreq != nominalFreq) {
        dvfs.setNominalCPUFrequency();
        curFreq = nominalFreq;
    }
    State state = State.HIGH;
    float mean, dev;
    int cpuUtil = 0;
    float adjustScreenDown = 0.1F;
    if (currentFocusedApp.getKeyMean() < currentFocusedApp.getTouchMean()) {
        mean = currentFocusedApp.getKeyMean();

```

```

        dev = currentFocusedApp.getKeyStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getTouchMean();
            dev = currentFocusedApp.getTouchStandardDeviation();
        }
    }
    else {
        mean = currentFocusedApp.getTouchMean();
        dev = currentFocusedApp.getTouchStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getKeyMean();
            dev = currentFocusedApp.getKeyStandardDeviation();
        }
    }
    float cov = dev / mean;
    if (cov < 0.5) { //Low variability
        adjustScreenDown = 0.15F;
    }
    else if (cov < 0.75) { //Med variability
        adjustScreenDown = 0.10F;
    }
    else if (cov > 1) { //High variability
        adjustScreenDown = 0.05F;
    }
    int previousTouchKeyEvents;
    long timeSinceLastEvent = 0;
    long timeSinceStartup = 0;
    Intent brightnessIntent = new Intent();
    brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
    float currentBrightness = getDefaultScreenBrightness();
    float defaultBrightness = getDefaultScreenBrightness();
    int defaultBrightnessPct = (int)defaultBrightness*100;
    int notificationCount = 0;
    float diff;

    if (dev > 0 && mean > 0) {
        previousTouchKeyEvents = touchEvents + keyEvents;

        while(isAlgorithmRunning) {
            //hitCount++; //Predict hit
            //SCREEN
            if ((touchEvents + keyEvents) > previousTouchKeyEvents) {
                synchronized(mAverageWindowedEventTime) {
                    if (mAverageWindowedEventTime.size() >
(EVENT_AVERAGE_WINDOW / 2)) {
                        mean = AppUtil.getMean(mAverageWindowedEventTime);
                        dev =
AppUtil.getStandardDeviation(mAverageWindowedEventTime);
                    }
                }

                if (state == State.LOW) {
                    missCount++; //Missed, so adjust hit
                    //hitCount--;
                }
                else {
                    hitCount++; // CHANGED FOR PERFORMANCE ANALYSIS - NO HIT
IF NO TOUCH EVENT
                }
                previousTouchKeyEvents = touchEvents + keyEvents;
                //currentBrightness = Math.min(currentBrightness +
adjustScreenUp, defaultBrightness);
                if (currentBrightness != defaultBrightness) {

```

```

        currentBrightness = defaultBrightness;
        brightnessIntent.putExtra("brightness",
currentBrightness);
        sendBroadcast(brightnessIntent);
    }
    timeSinceLastEvent = timeSinceStartup;
}
else {
    if (state == State.LOW) {
        if (currentBrightness > 0.4F) {
            currentBrightness = currentBrightness -
adjustScreenDown;
            brightnessIntent.putExtra("brightness",
currentBrightness);
            sendBroadcast(brightnessIntent);
        }
    }
}
//DVFS
cpuUtil = CPUInfo.getCPUUtilizationPct();
avgCpuUtil += cpuUtil;

diff = timeSinceStartup - timeSinceLastEvent;
switch (state) {
    case LOW:
        probOfEvent = AppUtil.getNormProb(diff, mean, dev, 100);
        if (probOfEvent > lowToHighProb || diff == 0) {
            state = State.HIGH;
            if (curFreq != nominalFreq) {
                dvfs.setNominalCPUFrequency();
                curFreq = nominalFreq;
            }
            currentBrightness = defaultBrightness;
            brightnessIntent.putExtra("brightness",
currentBrightness);
            sendBroadcast(brightnessIntent);
        }
        else if (probOfEvent > lowStepUpProb || cpuUtil > 80) {
            if (curFreq != lowNominalFreq) {
                dvfs.setLowNominalCPUFrequency();
                curFreq = lowNominalFreq;
            }
        }
        lowCount++;
        break;
    case HIGH:
        if (diff > 0) {
            probOfEvent = AppUtil.getNormProb(diff, mean, dev,
100);
            if (probOfEvent < highToLowProb && cpuUtil < 80) {
                state = State.LOW;
                if (curFreq != lowFreq) {
                    dvfs.setLowestCPUFrequency();
                    curFreq = lowFreq;
                }
            }
        }
        highCount++;
        break;
}

synchronized(mRunDVFSAlgorithm) {
    mRunDVFSAlgorithm.wait(basePeriod);
}

```

```

    }

    //curFreq = dvfs.getCPUFrequency();

    totalPowerNominal    +=    PowerModels.getMWCPUPower(nominalFreq,
cpuUtil);
    totalPowerNominal    +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

    if    (curFreq    <    nominalFreq    ||    currentBrightness    <
defaultBrightness) {
        totalPowerSavedTime += (basePeriod / 1000F);

        if (currentBrightness < defaultBrightness) {
            totalPowerSaved    +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
        }
        if (curFreq < nominalFreq) {
            totalPowerSaved    +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
        }
    }

    timeSinceStartup += basePeriod;

    if (notificationCount++ >= 10) {
        postNotification(String.format("[%0.3f    mW,    %0.3f    %%]",
totalPowerSaved, totalSOCSaved) , R.drawable.cpu_icon);
        notificationCount = 0;
    }
}
else {
    postNotification("STAT ERROR", R.drawable.cpu_icon);
}
}
}

```

```

private void runMovingAverageAlgorithm() throws InterruptedException {
    if (dvfs != null && (dvfs.getCurrentGovernor() == FrequencyGovernors.Userspace
|| dvfs.setFrequencyGovernor(FrequencyGovernors.Userspace))) {
        long basePeriod = 1000; //Default milliseconds
        float lowToHighProb = 0.7F;
        float lowStepUpProb = 0.6F;
        float highToLowProb = 0.5F;
        int nominalFreq = 729600; // changed for Nexus One
        int lowFreq = 245000; // changed for Nexus One
        int lowNominalFreq = 537600; // changed for Nexus One
        int curFreq = dvfs.getCPUFrequency();
        switch(currentFocusedApp.getAppClassification()) {
            // journal changes
            case VeryLowInteraction:
                basePeriod = 1000;
                lowToHighProb = 0.75F;
                lowStepUpProb = 0.65F;
                highToLowProb = 0.55F;
            case LowInteraction:
                basePeriod = 850;
                lowToHighProb = 0.7F;
                lowStepUpProb = 0.6F;
                highToLowProb = 0.5F;
                break;
            case LowMedInteraction:

```



```

        basePeriod = 700;
        lowToHighProb = 0.65F;
        lowStepUpProb = 0.55F;
        highToLowProb = 0.45F;
    case MedInteraction:
        basePeriod = 550;
        lowToHighProb = 0.6F;
        lowStepUpProb = 0.5F;
        highToLowProb = 0.4F;
        break;
    case MedHighInteraction:
        basePeriod = 400;
        lowToHighProb = 0.55F;
        lowStepUpProb = 0.45F;
        highToLowProb = 0.35F;
    case HighInteraction:
        basePeriod = 250;
        lowToHighProb = 0.5F;
        lowStepUpProb = 0.4F;
        highToLowProb = 0.3F;
    case VeryHighInteraction:
        basePeriod = 100;
        lowToHighProb = 0.45F;
        lowStepUpProb = 0.35F;
        highToLowProb = 0.25F;
        break;
    }
    float probOfEvent = 0F;
    if (curFreq != nominalFreq) {
        dvfs.setNominalCPUFrequency();
        curFreq = nominalFreq;
    }
    State state = State.HIGH;
    float mean, dev;
    int cpuUtil = 0;
    float adjustScreenDown = 0.1F;
    if (currentFocusedApp.getKeyMean() < currentFocusedApp.getTouchMean()) {
        mean = currentFocusedApp.getKeyMean();
        dev = currentFocusedApp.getKeyStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getTouchMean();
            dev = currentFocusedApp.getTouchStandardDeviation();
        }
    }
    else {
        mean = currentFocusedApp.getTouchMean();
        dev = currentFocusedApp.getTouchStandardDeviation();
        if (mean <= 0) {
            mean = currentFocusedApp.getKeyMean();
            dev = currentFocusedApp.getKeyStandardDeviation();
        }
    }
    float cov = dev / mean;
    if (cov < 0.5) { //Low variability
        adjustScreenDown = 0.15F;
    }
    else if (cov < 0.75) { //Med variability
        adjustScreenDown = 0.10F;
    }
    else if (cov > 1) { //High variability
        adjustScreenDown = 0.05F;
    }
    int previousTouchKeyEvents;

```

```

long timeSinceLastEvent = 0;
long timeSinceStartup = 0;
Intent brightnessIntent = new Intent();
brightnessIntent.setAction(SCREENBRIGHTNESS_CHANGE_EVENT);
float currentBrightness = getDefaultScreenBrightness();
float defaultBrightness = getDefaultScreenBrightness();
int defaultBrightnessPct = (int)defaultBrightness*100;
int notificationCount = 0;
float diff;
int timeAvgWindow = (int)Math.max(1, (mean/basePeriod)*0.5F);
int windowCounter = 0;

if (dev > 0 && mean > 0) {
    previousTouchKeyEvents = touchEvents + keyEvents;

    while(isAlgorithmRunning) {
        //hitCount++; //Predict hit
        //SCREEN
        if ((touchEvents + keyEvents) > previousTouchKeyEvents) {
            if (state == State.LOW) {
                missCount++; //Missed, so adjust hit
                //hitCount--;
            }
            else {
                hitCount++; // CHANGED FOR PERFORMANCE ANALYSIS - NO HIT
            }
        }
        previousTouchKeyEvents = touchEvents + keyEvents;
        //currentBrightness = Math.min(currentBrightness +
adjustScreenUp, defaultBrightness);
        if (currentBrightness != defaultBrightness) {
            currentBrightness = defaultBrightness;
            brightnessIntent.putExtra("brightness",
currentBrightness);
            sendBroadcast(brightnessIntent);
        }
        timeSinceLastEvent = timeSinceStartup;
    }
    else {
        if (state == State.LOW) {
            if (currentBrightness > 0.4F) {
                currentBrightness = currentBrightness -
adjustScreenDown;
                brightnessIntent.putExtra("brightness",
currentBrightness);
                sendBroadcast(brightnessIntent);
            }
        }
    }
}
//DVFS
cpuUtil = CPUInfo.getCPUUtilizationPct();
avgCpuUtil += cpuUtil;

diff = timeSinceStartup - timeSinceLastEvent;
switch (state) {
    case LOW:
        probOfEvent = AppUtil.getNormProb(diff, mean, dev, 100);
        if (probOfEvent > lowToHighProb || diff == 0) {
            state = State.HIGH;
            if (curFreq != nominalFreq) {
                dvfs.setNominalCPUFrequency();
                curFreq = nominalFreq;
            }
        }
    }
}

```

```

        currentBrightness = defaultBrightness;
        brightnessIntent.putExtra("brightness",
currentBrightness);
        sendBroadcast(brightnessIntent);
    }
    else if (probOfEvent > lowStepUpProb || cpuUtil > 80) {
        if (curFreq != lowNominalFreq) {
            dvfs.setLowNominalCPUFrequency();
            curFreq = lowNominalFreq;
        }
        lowCount++;
        break;
    case HIGH:
        if (diff > 0) {
            probOfEvent = AppUtil.getNormProb(diff, mean, dev,
100);

            if (probOfEvent < highToLowProb && cpuUtil < 80) {
                state = State.LOW;
                if (curFreq != lowFreq) {
                    dvfs.setLowestCPUFrequency();
                    curFreq = lowFreq;
                }
            }
            highCount++;
            break;
        }

        synchronized(mRunDVFSAlgorithm) {
            mRunDVFSAlgorithm.wait(basePeriod);
        }

        //curFreq = dvfs.getCPUFrequency();

        totalPowerNominal += PowerModels.getMWCPUPower(nominalFreq,
cpuUtil);
        totalPowerNominal +=
PowerModels.getMWScreenPower(defaultBrightnessPct);

        if (curFreq < nominalFreq || currentBrightness <
defaultBrightness) {
            totalPowerSavedTime += (basePeriod / 1000F);

            if (currentBrightness < defaultBrightness) {
                totalPowerSaved +=
PowerModels.getMWScreenPowerSavings(defaultBrightness, currentBrightness);
            }
            if (curFreq < nominalFreq) {
                totalPowerSaved +=
PowerModels.getMWCPUPowerSavings(nominalFreq, curFreq, cpuUtil);
            }
        }

        timeSinceStartup += basePeriod;

        if (notificationCount++ >= 10) {
            postNotification(String.format("[%.3f mW, %.3f %%]",
totalPowerSaved, totalSOCSaved) , R.drawable.cpu_icon);
            notificationCount = 0;
        }
    }
}

```

```

        if (mTimeAverageWindowedEventTime.size() >= EVENT_AVERAGE_WINDOW)
    {
        mTimeAverageWindowedEventTime.remove(0);
        mTimeAverageWindowedEventTime.add((float)diff);
    }
    else {
        mTimeAverageWindowedEventTime.add((float)diff);
    }

    if (windowCounter++ >= timeAvgWindow) {
        float windowMean =
AppUtil.getMean(mTimeAverageWindowedEventTime);

        switch(currentFocusedApp.getAppClassification()) {
            // journal changes
            case VeryLowInteraction:
                if (mean > (windowMean+1000)) {
                    mean = Math.max(1000, mean-1000);
                }
                else if (mean < (windowMean-1000)) {
                    mean = mean+1000;
                }
                break;
            case LowInteraction:
                if (mean > (windowMean+850)) {
                    mean = Math.max(850, mean-850);
                }
                else if (mean < (windowMean-850)) {
                    mean = mean+850;
                }
                break;
            case LowMedInteraction:
                if (mean > (windowMean+700)) {
                    mean = Math.max(700, mean-700);
                }
                else if (mean < (windowMean-700)) {
                    mean = mean+700;
                }
                break;
            case MedInteraction:
                if (mean > (windowMean+550)) {
                    mean = Math.max(500, mean-500);
                }
                else if (mean < (windowMean-500)) {
                    mean = mean+500;
                }
                break;
            case MedHighInteraction:
                if (mean > (windowMean+400)) {
                    mean = Math.max(400, mean-400);
                }
                else if (mean < (windowMean-400)) {
                    mean = mean+400;
                }
                break;
            case HighInteraction:
                if (mean > (windowMean+250)) {
                    mean = Math.max(250, mean-250);
                }
                else if (mean < (windowMean-250)) {
                    mean = mean+250;
                }
                break;
        }
    }
}

```

```

        case VeryHighInteraction:
            if (mean > (windowMean+100)) {
                mean = Math.max(100, mean-100);
            }
            else if (mean < (windowMean-100)) {
                mean = mean+100;
            }
            break;
        }
        windowCounter = 0;
    }
}
else {
    postNotification("STAT ERROR", R.drawable.cpu_icon);
}
}

private void setScreenOffDVFSLevel() {
    int nominalFreq = 528000;
    int curFreq = nominalFreq;
    int lowNominalFreq = 245760;
    if (isAlgorithmRunning) {
        stopDVFSAlgorithm();
    }
    if (dvfs != null && dvfs.getCurrentGovernor() ==
FrequencyGovernors.Userspace) {
        if (curFreq != lowNominalFreq) {
            dvfs.setLowNominalCPUFrequency();
            curFreq = lowNominalFreq;
        }
    }
}

private void setScreenOnDVFSLevel() {
    int nominalFreq = 528000;
    int curFreq = nominalFreq;

    if (dvfs != null && dvfs.getCurrentGovernor() ==
FrequencyGovernors.Userspace && !isAlgorithmRunning) {
        ArrayList<Integer> freq = dvfs.getFrequencyScaleModes();
        if (freq != null && freq.size() > 0) {
            if (curFreq != nominalFreq) {
                dvfs.setNominalCPUFrequency();
                curFreq = nominalFreq;
            }
        }
        else {
            dvfs.setCPUFrequency(dvfs.getMaxCPUFrequency());
            curFreq = dvfs.getMaxCPUFrequency();
        }
    }
}

private float getDefaultScreenBrightness() {
    float temp;
    try {
        int brightness = (int)
android.provider.Settings.System.getFloat(getContentResolver(),
android.provider.Settings.System.SCREEN_BRIGHTNESS);
        temp = brightness / 255F;
        if (temp < 0.4F) {

```

```

        temp = 0.4F;
    }
    return temp;
}
catch (SettingNotFoundException e) {
    return 1F;
}
}

/*
 * naiveNumEventClassifier classifies apps after a specified number
 * of events by taking the average of all time between events
 * received thus far
 */
public void naiveNumEventClassifier() {
    int i;
    long totalTime = 0;
    long average = 0;

    if (touchEvents + keyEvents < NUM_EVENTS_CLASSIFIER) {
        return;
    }
    else {
        for (i = 0; i < deltaTouchEventTimeList.size(); i++) {
            totalTime += deltaTouchEventTimeList.get(i);
        }
        for (i = 0; i < deltaKeyEventTimeList.size(); i++) {
            totalTime += deltaKeyEventTimeList.get(i);
        }

        average = totalTime / (deltaTouchEventTimeList.size() +
deltaKeyEventTimeList.size());

        if (average <= 2000) {
            // classify as very high interaction

currentFocusedApp.classifyApplication(AppClassification.VeryHighInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
            Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
        }
        else if (average <= 3000) {
            // classify as high interaction

currentFocusedApp.classifyApplication(AppClassification.HighInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
            Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
        }
        else if (average <= 4000) {
            // classify as medium-high interaction

currentFocusedApp.classifyApplication(AppClassification.MedHighInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));

```

```

                Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
                AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
            }
            else if (average <= 5000) {
                // classify as medium interaction

currentFocusedApp.classifyApplication(AppClassification.MedInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
                Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
                AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
            }
            else if (average <= 6000) {
                // classify as low-medium interaction

currentFocusedApp.classifyApplication(AppClassification.LowMedInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
                Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
                AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
            }
            else {
                // classify as low interaction

currentFocusedApp.classifyApplication(AppClassification.LowInteraction);

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
                Toast.makeText(getBaseContext(), String.format("App classified as
%s!",
                AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
            }
        }
    }

    public void bayesianClassifier(float mean, float sd, long dT) {
        String meanLevel = "";
        String sdLevel = "";
        String etLevel = "";

        // posterior probabilities
        float postProbVeryFast = 0F; // journal addition
        float postProbFast = 0F;
        float postProbMedFast = 0F; // journal addition
        float postProbMed = 0F;
        float postProbSlowMed = 0F; // journal addition
        float postProbSlow = 0F;
        float postProbVerySlow = 0F; // journal addition
        // prior probabilities
        float priorProbVeryFast = 0F; // journal addition
        float priorProbFast = 0F;
        float priorProbMedFast = 0F; // journal addition
        float priorProbMed = 0F;
        float priorProbSlowMed = 0F; // journal addition
        float priorProbSlow = 0F;
        float priorProbVerySlow = 0F; // journal addition
        // class likelihoods
        float likelihoodMeanVeryFast = 0F; // journal addition

```

```

float likelihoodSDVeryFast = 0F;    // journal addition
float likelihoodMeanFast = 0F;
float likelihoodSDFast = 0F;
float likelihoodMeanMedFast = 0F;   // journal addition
float likelihoodSDMedFast = 0F;     // journal addition
float likelihoodMeanMed = 0F;
float likelihoodSDMed = 0F;
float likelihoodMeanSlowMed = 0F;   // journal addition
float likelihoodSDSlowMed = 0F;     // journal addition
float likelihoodMeanSlow = 0F;
float likelihoodSDSlow = 0F;
float likelihoodMeanVerySlow = 0F;  // journal addition
float likelihoodSDVerySlow = 0F;    // journal addition

if (mean > 0 && sd > 0 && dT > 0) {
    if (mean <= 1000) {
        meanLevel = "very fast";
    }
    else if (mean <= 2000) {
        meanLevel = "fast";
    }
    else if (mean <= 3000) {
        meanLevel = "med fast";
    }
    else if (mean <= 4000) {
        meanLevel = "med";
    }
    else if (mean <= 5000) {
        meanLevel = "slow med";
    }
    else if (mean <= 6000) {
        meanLevel = "slow";
    }
    else {
        meanLevel = "very slow";
    }

    if (sd <= 1000) {
        sdLevel = "very low";
    }
    else if (sd <= 1500) {
        sdLevel = "low";
    }
    else if (sd <= 2000) {
        sdLevel = "low med";
    }
    else if (sd <= 2500) {
        sdLevel = "med";
    }
    else if (sd <= 3000) {
        sdLevel = "med high";
    }
    else if (sd <= 3500) {
        sdLevel = "high";
    }
    else {
        sdLevel = "very high";
    }

    if (dT <= 1000) {
        etLevel = "very fast";
    }
    else if (dT <= 2000) {

```



```

        etLevel = "fast";
    }
    else if (dT <= 3000) {
        etLevel = "med fast";
    }
    else if (dT <= 4000) {
        etLevel = "med";
    }
    else if (dT <= 5000) {
        etLevel = "slow med";
    }
    else if (dT <= 6000) {
        etLevel = "slow";
    }
    else {
        etLevel = "very slow";
    }
}

if (meanLevel.compareTo("") != 0 && sdLevel.compareTo("") != 0 &&
etLevel.compareTo("") != 0) {
    priorProbVeryFast = currentFocusedApp.findTTProb("very fast");
    priorProbFast = currentFocusedApp.findTTProb("fast");
    priorProbMedFast = currentFocusedApp.findTTProb("med fast");
    priorProbMed = currentFocusedApp.findTTProb("med");
    priorProbSlowMed = currentFocusedApp.findTTProb("slow med");
    priorProbSlow = currentFocusedApp.findTTProb("slow");
    priorProbVerySlow = currentFocusedApp.findTTProb("very slow");

    likelihoodMeanVeryFast = currentFocusedApp.findMeanAndTTProb(meanLevel,
"very fast");
    likelihoodSDVeryFast = currentFocusedApp.findSDAndTTProb(sdLevel, "very
fast");
    likelihoodMeanFast = currentFocusedApp.findMeanAndTTProb(meanLevel,
"fast");
    likelihoodSDFast = currentFocusedApp.findSDAndTTProb(sdLevel, "fast");
    likelihoodMeanMedFast = currentFocusedApp.findMeanAndTTProb(meanLevel,
"med fast");
    likelihoodSDMedFast = currentFocusedApp.findSDAndTTProb(sdLevel, "med
fast");
    likelihoodMeanMed = currentFocusedApp.findMeanAndTTProb(meanLevel, "med");
    likelihoodSDMed = currentFocusedApp.findSDAndTTProb(sdLevel, "med");
    likelihoodMeanSlowMed = currentFocusedApp.findMeanAndTTProb(meanLevel,
"slow med");
    likelihoodSDSlowMed = currentFocusedApp.findSDAndTTProb(sdLevel, "slow
med");
    likelihoodMeanSlow = currentFocusedApp.findMeanAndTTProb(meanLevel,
"slow");
    likelihoodSDSlow = currentFocusedApp.findSDAndTTProb(sdLevel, "slow");
    likelihoodMeanVerySlow = currentFocusedApp.findMeanAndTTProb(meanLevel,
"very slow");
    likelihoodSDVerySlow = currentFocusedApp.findSDAndTTProb(sdLevel, "very
slow");

    postProbVeryFast = priorProbVeryFast * likelihoodMeanVeryFast *
likelihoodSDVeryFast;
    postProbFast = priorProbFast * likelihoodMeanFast * likelihoodSDFast;
    postProbMedFast = priorProbMedFast * likelihoodMeanMedFast *
likelihoodSDMedFast;
    postProbMed = priorProbMed * likelihoodMeanMed * likelihoodSDMed;
    postProbSlowMed = priorProbSlowMed * likelihoodMeanSlowMed *
likelihoodSDSlowMed;
    postProbSlow = priorProbSlow * likelihoodMeanSlow * likelihoodSDSlow;

```

```

        postProbVerySlow = priorProbVerySlow * likelihoodMeanVerySlow *
likelihoodSDVerySlow;

        if (Float.isNaN(postProbVeryFast))
            postProbVeryFast = 0;
        if (Float.isNaN(postProbFast))
            postProbFast = 0;
        if (Float.isNaN(postProbMedFast))
            postProbMedFast = 0;
        if (Float.isNaN(postProbMed))
            postProbMed = 0;
        if (Float.isNaN(postProbSlowMed))
            postProbSlowMed = 0;
        if (Float.isNaN(postProbSlow))
            postProbSlow = 0;
        if (Float.isNaN(postProbVerySlow))
            postProbVerySlow = 0;

        if (postProbVeryFast >= postProbFast && postProbVeryFast >=
postProbMedFast
            && postProbVeryFast >= postProbMed && postProbVeryFast >=
postProbSlowMed
            && postProbVeryFast >= postProbSlow && postProbVeryFast >=
postProbVerySlow) {
            // probability of an event happening fast is high
            // so classify app as high interaction

currentFocusedApp.classifyApplication(AppClassification.VeryHighInteraction);
        }
        else if (postProbFast >= postProbVeryFast && postProbFast >=
postProbMedFast
            && postProbFast >= postProbMed && postProbFast >= postProbSlowMed
            && postProbFast >= postProbSlow && postProbFast >=
postProbVerySlow) {
            // probability of an event happening fast is high
            // so classify app as high interaction

currentFocusedApp.classifyApplication(AppClassification.HighInteraction);
        }
        else if (postProbMedFast >= postProbVeryFast && postProbMedFast >=
postProbFast
            && postProbMedFast >= postProbMed && postProbMedFast >=
postProbSlowMed
            && postProbMedFast >= postProbSlow && postProbMedFast >=
postProbVerySlow) {
            // probability of an event happening fast is high
            // so classify app as high interaction

currentFocusedApp.classifyApplication(AppClassification.MedHighInteraction);
        }
        else if (postProbMed >= postProbVeryFast && postProbMed >= postProbFast
            && postProbMed >= postProbMedFast && postProbMed >=
postProbSlowMed
            && postProbMed >= postProbSlow && postProbMed >= postProbVerySlow)
        {
currentFocusedApp.classifyApplication(AppClassification.MedInteraction);
        }
        else if (postProbSlowMed >= postProbVeryFast && postProbSlowMed >=
postProbFast
            && postProbSlowMed >= postProbMedFast && postProbSlowMed >=
postProbMed

```

```

        && postProbSlowMed >= postProbSlow && postProbSlowMed >=
postProbVerySlow) {
currentFocusedApp.classifyApplication(AppClassification.LowMedInteraction);
    }
    else if (postProbSlow >= postProbVeryFast && postProbSlow >= postProbFast
        && postProbSlow >= postProbMedFast && postProbSlow >= postProbMed
        && postProbSlow >= postProbSlowMed && postProbSlow >=
postProbVerySlow) {
        // probability of an event happening slow is high
        // so classify app as low interaction

currentFocusedApp.classifyApplication(AppClassification.LowInteraction);
    }
    else if (postProbVerySlow >= postProbVeryFast && postProbVerySlow >=
postProbFast
        && postProbVerySlow >= postProbMedFast && postProbVerySlow >=
postProbMed
        && postProbVerySlow >= postProbSlowMed && postProbVerySlow >=
postProbSlow) {
currentFocusedApp.classifyApplication(AppClassification.VeryLowInteraction);
    }

    currentFocusedApp.addTrainingSetEntry(meanLevel, sdLevel, etLevel);

    if (previousClassification != AppClassification.NotClassified) {
        if (currentFocusedApp.getAppClassification() ==
previousClassification) {
            unchangedClassCount++;
            if (unchangedClassCount > 5) {
                // set app trained with current classification

currentFocusedApp.setTrainedWithClassification(currentFocusedApp.getAppClassification(
));
                Toast.makeText(getBaseContext(), String.format("App classified
as %s!", AppClassification.toString(currentFocusedApp.getAppClassification())),
Toast.LENGTH_LONG).show();
            }
        }
        else {
            unchangedClassCount = 0;
        }
    }

    previousClassification = currentFocusedApp.getAppClassification();

    //Toast.makeText(getBaseContext(), String.format("app: %s\nmean: %.1f\nsd:
%.1f\ntt: %d\nmeanLevel: %s\nsdLevel: %s\nttLevel: %s\ntrainingSet size:
%d\npostProbHigh: %.1f\npostProbMed: %.1f\npostProbLow: %.1f\nunchangedClassCount:
%d\nclassification: %s",
        //
        currentFocusedApp.getName(), mean, sd, et,
meanLevel, sdLevel, etLevel, currentFocusedApp.getTrainingSet().size(), postProbFast,
postProbMed, postProbSlow, unchangedClassCount,
currentFocusedApp.getAppClassification().toString()), Toast.LENGTH_LONG).show();

    }
}
}

```

## A.22 ContextLogger.java (Strategy 2)

```
package csu.research;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.AlertDialog;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.location.Location;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.SystemClock;
import android.util.Log;
import android.view.KeyEvent;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;
import android.widget.Chronometer;
import android.widget.TextView;
import android.widget.Toast;
import android.widget.ToggleButton;

import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.LinkedList;

public class ContextLogger extends Activity {
    private final int OPTIONS_GROUP_DEFAULT = 0;
    private final int OPTIONS_GROUP_TEST = 1;
    private final int OPTIONS_MENU_SHOWLOCATIONS = 1;
    private LocationDatabase mLocDatabase;
    private TextView mMainTextView;
    Button mStartButton;
    Button mStopButton;
    private PendingIntent mAlarmSender;
    private boolean mLoggingActive;
    private int mLoggingInterval; // Logging interval for periods of no user
interaction
    private int mLoggingInterval2; // Logging interval when screen is on
BroadcastReceiver mScreenReceiver;

    /** Called when the activity is first created. */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // VARIABLES
        mLoggingActive = false;
        mLoggingInterval = 10 * 60 * 1000; // 10 minutes
        mLoggingInterval2 = 60 * 1000; // 1 minute

        // UI ELEMENTS
        mMainTextView = (TextView)findViewById(R.id.DisplayTextView);
```

```

registerForContextMenu(mMainTextView);
mStartButton = (Button)findViewById(R.id.StartButton);
mStartButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        startLogging();
    }
});
mStopButton = (Button)findViewById(R.id.StopButton);
mStopButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        stopLogging();
    }
});

// Check if an alarm is already scheduled
Intent i = new Intent(ContextLogger.this, ContextLoggerService.class);
mAlarmSender = PendingIntent.getService(ContextLogger.this, 0, i,
PendingIntent.FLAG_NO_CREATE);

if (mAlarmSender != null) {
    // Service running (alarm already scheduled)
    mLoggingActive = true;
    mMainTextView.setText("Logging active!");
    mStartButton.setEnabled(false);

//         // Initialize screen broadcast receiver
//         IntentFilter filter = new IntentFilter(Intent.ACTION_SCREEN_ON);
//         filter.addAction(Intent.ACTION_SCREEN_OFF);
//         mScreenReceiver = new ScreenBroadcastReceiver();
//         registerReceiver(mScreenReceiver, filter);
}
else {
    mLoggingActive = false;
    // Create an IntentSender that will launch our service, to be scheduled
    // with the alarm manager.
    mAlarmSender = PendingIntent.getService(ContextLogger.this, 0, i, 0);
    mMainTextView.setText("Logging disabled...");
    mStopButton.setEnabled(false);
}
}

@Override
protected void onDestroy() {
    stopLogging();
    super.onDestroy();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(OPTIONS_GROUP_TEST, OPTIONS_MENU_SHOWLOCATIONS, 0, "Show Locations");
    return super.onCreateOptionsMenu(menu);
}

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    menu.setGroupVisible(OPTIONS_GROUP_DEFAULT, true);
    menu.setGroupVisible(OPTIONS_GROUP_TEST, true);
    return super.onPrepareOptionsMenu(menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {

```

```

case OPTIONS_MENU_SHOWLOCATIONS:
    // Check for database and get lats and longs here

    if (loadDatabase()) {
        Intent intent = new Intent(this, LocationsViewer.class);
        LinkedList<Location> locations;

        locations = mLocDatabase.getLocations();
        int numLocations = mLocDatabase.getNumberOfLocations();
        double[] lats = new double[numLocations];
        double[] longs = new double[numLocations];

        for (int i = 0; i < numLocations; i++) {
            lats[i] = locations.get(i).getLatitude();
            longs[i] = locations.get(i).getLongitude();
        }

        intent.putExtra("latitudes", lats);
        intent.putExtra("longitudes", longs);

        startActivity(intent);
    }
    else {
        Toast.makeText(getApplicationContext(), "Locations N/A",
Toast.LENGTH_LONG).show();
    }
    return true;
}
return super.onOptionsItemSelected(item);
}

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (mLoggingActive && keyCode == KeyEvent.KEYCODE_BACK) {
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setMessage(String.format("Logging is currently active!\n" +
            "Please press CANCEL, then the HOME button if you would " +
            "like the logger to run in the background. Otherwise, press " +
            "EXIT to stop logging and close the application.));
        builder.setCancelable(true);
        builder.setTitle("EXIT WARNING!");
        builder.setPositiveButton("Save/Exit", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                stopLogging(); //IMPLEMENT ADDITIONAL EXIT CODE IF NEEDED!
                ContextLogger.this.finish();
            }
        });

        builder.setNegativeButton("EXIT", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                ContextLogger.this.finish();
            }
        });

        builder.setNegativeButton("CANCEL", new DialogInterface.OnClickListener()
{
            public void onClick(DialogInterface dialog, int which) {
                dialog.cancel();
            }
        });

        builder.show();
    }
}

```

```

        return false;
    }
    else {
        return super.onKeyDown(keyCode, event);
    }
}

public void startLogging() {
    mStartButton.setEnabled(false);
    mStopButton.setEnabled(true);

    mMainTextView.setText("Logging active!");

    // We want the alarm to go off immediately
    long firstTime = SystemClock.elapsedRealtime();

    Toast.makeText(this, R.string.logging_started,
        Toast.LENGTH_SHORT).show();

    // Schedule the alarm with the faster logging interval (because the screen is
on)
    AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);
    am.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP, firstTime,
        mLoggingInterval2, mAlarmSender);

    // // Initialize screen broadcast receiver
    // IntentFilter filter = new IntentFilter(Intent.ACTION_SCREEN_ON);
    // filter.addAction(Intent.ACTION_SCREEN_OFF);
    // mScreenReceiver = new ScreenBroadcastReceiver();
    // registerReceiver(mScreenReceiver, filter);

    mLoggingActive = true;
}

public void stopLogging() {
    mStartButton.setEnabled(true);
    mStopButton.setEnabled(false);

    mMainTextView.setText("Logging disabled...");

//    unregisterReceiver(mScreenReceiver);

    // Cancel the alarm.
    AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);
    am.cancel(mAlarmSender);

    Toast.makeText(this, R.string.logging_stopped,
        Toast.LENGTH_SHORT).show();

    mLoggingActive = false;
}

private boolean loadDatabase() {
    try {
        if
(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath = String.format("%s/ContextLogger/Locations.ser",
Environment.getExternalStorageDirectory());
            if (new File(filePath).exists()) {
                FileInputStream fis = new FileInputStream(filePath);
                ObjectInputStream in = new ObjectInputStream(fis);
                mLocDatabase = (LocationDatabase)in.readObject();
                in.close();
            }
        }
    }
}

```

```

        return true;
    }
}
mLocDatabase = new LocationDatabase();
return false;
}
catch (Exception e) {
    mLocDatabase = new LocationDatabase();
    return false;
}
}

private boolean deleteDatabase() {
    try {
        if
(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath = String.format("%s/ContextLogger",
Environment.getExternalStorageDirectory());
            if (new File(filePath).exists()) {
                mLocDatabase.deleteDatabase(filePath);
                mLocDatabase.deleteExport(filePath);
                return true;
            }
        }
        return false;
    }
    catch (Exception e) {
        return false;
    }
}
}
}

```

### A.23 ContextLoggerService.java (Strategy 2)

```

package csu.research;

import android.app.ActivityManager;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.app.ActivityManager.RunningAppProcessInfo;
import android.app.ActivityManager.RunningServiceInfo;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.location.Location;
import android.location.LocationManager;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.wifi.WifiManager;
import android.os.Binder;
import android.os.Environment;
import android.os.IBinder;
import android.os.Parcel;

```



```

import android.os.PowerManager;
import android.os.RemoteException;
import android.os.SystemClock;
import android.provider.Settings;
import android.provider.Settings.SettingNotFoundException;
import android.telephony.TelephonyManager;
import android.util.Log;
import android.view.Display;
import android.view.WindowManager;
import android.widget.Toast;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

/**
 * This is an example of implementing an application service that will run in
 * response to an alarm, allowing us to move long duration work out of an intent
 * receiver.
 *
 */
public class ContextLoggerService extends Service implements AccelerometerListener {
    private final String TAG = "ContextLoggerService";
    private LocationDatabase mLocDatabase;
    private NotificationManager mNM;
    private WifiManager mWM;
    private ConnectivityManager mCM;
    private TelephonyManager mTM;
    private LocationManager mLM;
    private ActivityManager mAM;
    private PackageManager mPackM;
    private PowerManager mPM;
    private AudioManager mAudioM;
    private AccelerometerManager mAccM;
    private NetInfo mNetInfo;
    private BatteryInfo mBattInfo;
    private SensorInfo mSensorInfo;
    private CpuInfo mCpuInfo;
    private LocationInfo mLocInfo;
    private boolean mLocationCheck;
    private boolean mMoving = false;
    private int mScreenBrightness;

    @Override
    public void onCreate() {
        // SERVICES
        mNM = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        mWM = (WifiManager) getSystemService(WIFI_SERVICE);
        mCM = (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);
        mTM = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        mLM = (LocationManager) getSystemService(LOCATION_SERVICE);
        mAM = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        mPM = (PowerManager) getSystemService(Context.POWER_SERVICE);
        mPackM = getPackageManager();
        mAudioM = (AudioManager) getSystemService(AUDIO_SERVICE);
        mAccM = new AccelerometerManager(getApplicationContext());
    }

```

```

// CLASSES
mNetInfo = new NetInfo(mWM, mCM, mTM);
mBattInfo = new BatteryInfo(getApplicationContext());
mSensorInfo = new SensorInfo(getApplicationContext());
mCpuInfo = new CpuInfo();
mLocInfo = new LocationInfo(mLM);

// DATABASE
if (!loadDatabase()) {
    Toast.makeText(getApplicationContext(), "New database created!",
Toast.LENGTH_LONG).show();
}

// VARIABLES

// show the icon in the status bar
showNotification();

//if (!mPM.isScreenOn()) {
    mLocationCheck = mLocInfo.checkForLocation();
//}

// Start accelerometer
if (mAccM.isSupported()) {
    mAccM.startListening(this);
}

IntentFilter filt = new IntentFilter();
filt.addAction(Intent.ACTION_SCREEN_OFF);

// Start up the thread running the service. Note that we create a
// separate thread because the service normally runs in the process's
// main thread, which we don't want to block.
Thread thr = new Thread(null, mTask, "Context Logger service");
thr.start();

// The the user we started
// Toast.makeText(this, R.string.alarm_service_started,
// Toast.LENGTH_SHORT).show();
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // We want this service to continue running until it is explicitly
    // stopped, so return sticky
    return START_STICKY;
}

@Override
public void onDestroy() {
    // Cancel the notification -- we use the same ID that we had used to
    // start it
    mNM.cancel(R.string.iconized);

    if (mAccM.isListening()) {
        mAccM.stopListening();
    }

    mSensorInfo.unregisterListeners();

    // Tell the user we stopped.
// Toast.makeText(this, R.string.alarm_service_stopped,

```

```

//          Toast.LENGTH_SHORT).show();

        if (!saveDatabase()) {
            Toast.makeText(getApplicationContext(), "Error occurred saving location
database!", Toast.LENGTH_LONG);
        }
    }

/**
 * The function that runs in our worker thread
 */
Runnable mTask = new Runnable() {
    public void run() {
        String foregroundApp = "";
        String outputString = "";
        long oldWifiTraffic;
        long newWifiTraffic;
        long wifiTrafficDiff;
        Calendar calendar = Calendar.getInstance();

        File log = new File(String.format("%s/ContextLogger",
Environment.getExternalStorageDirectory()),
String.format("ContextLoggerLog_%d.txt", calendar.get(Calendar.DAY_OF_MONTH)));

        try {
            BufferedWriter out = new BufferedWriter(new FileWriter(
                log.getAbsolutePath(), log.exists()));

            mCpuInfo.getCPUStats();

//          foregroundApp = getForegroundAppName();

            /*
            * ATTRIBUTES:
            * Day of Week
            * Time of Day (Minutes)
            * Latitude
            * Longitude
            * GPS Satellite Count (measure of GPS signal strength)
            * Number of Wifi APs Available
            * WiFi RSSI (measure of WiFi signal strength)
            * CDMA Data Signal Strength
            * EVDO Data Signal Strength
            * GSM Data Signal Strength
            * Call State
            * Battery Level
            * Battery Status
            * CPU Utilization
            * Context Switches
            * Processes Created
            * Processes Running
            * Processes Blocked
            * Screen On?
            * User Moving?
            * Ambient Light
            *
            * TARGET VARIABLES:
            * Data Needed?
            * Coarse Location Needed?
            * Fine Location Needed?
            */

```

```

/*
 * MODIFY LOGGER TO LOG START/STOP OF INTERACTION SESSION
 * DURING INTERACTION SESSION
 */

    mScreenBrightness = (int)
    android.provider.Settings.System.getFloat(getContentResolver(),
    android.provider.Settings.System.SCREEN_BRIGHTNESS);

    // get original traffic so difference can be calculated
    oldWifiTraffic = mNetInfo.readWifiTraffic();

    int minuteOfDay = (calendar.get(Calendar.HOUR_OF_DAY) * 60) +
    calendar.get(Calendar.MINUTE);

    if (mLocationCheck == false) {
        // wait for 20 seconds.
        long endTime = System.currentTimeMillis() + 20 * 1000;
        while (System.currentTimeMillis() < endTime);

        outputString = String.format("%d,%d,%s,%s",
            calendar.get(Calendar.DAY_OF_WEEK),
            minuteOfDay,
            "?", // latitude not available
            "?"); // longitude not available
    }
    else {
        Location loc = mLocInfo.getCurrentLocation();

        // Check for best location for 20 seconds.
        long endTime = System.currentTimeMillis() + 20 * 1000;
        while (loc == null && System.currentTimeMillis() < endTime) {
            loc = mLocInfo.getCurrentLocation();
            //Log.d(TAG, "loc = null");
        }

        mLocInfo.removeUpdates();

        outputString = String.format("%d,%d",
            calendar.get(Calendar.DAY_OF_WEEK),
            minuteOfDay);

        if (loc != null) {
            // add location to database
            mLocDatabase.addLocation(loc);

            outputString = outputString.concat(String.format(",%.6f,%.6f",
                loc.getLatitude(),
                loc.getLongitude()));
        }
        else {
            outputString = outputString.concat(",?,?");
        }
    }

    if (mLocInfo.isGpsEnabled()) {
        outputString = outputString.concat(String.format(",%d",
            mLocInfo.getNumSatellites()));
    }
    else {
        outputString = outputString.concat(",?");
    }
}

```

```

    if (mNetInfo.isWifiEnabled()) {
        outputString = outputString.concat(String.format(",%d",
            mNetInfo.getNumAPs()));
    }
    else {
        outputString = outputString.concat(",?");
    }

    if (mNetInfo.isWifiConnected()) {
        outputString = outputString.concat(String.format(",%d",
            mNetInfo.getWifiSignalLevel()));
    }
    else {
        outputString = outputString.concat(",?");
    }

    // Get network data signal strengths
    outputString = outputString.concat(String.format(",%d,%d,%d",
        mNetInfo.getCdmaSigStrength(),
        mNetInfo.getEvdoSigStrength(),
        mNetInfo.getGsmSigStrength()));

    outputString
outputString.concat(String.format(",%d,%d,%d,%.1f,%d,%d,%d,%d,%d,%d",
        mNetInfo.getCallState(),
        mBattInfo.getBatteryLevel(),
        mBattInfo.getBatteryStatus(),
        mCpuInfo.getUtilizationPct(),
        mCpuInfo.getContextSwitches(),
        mCpuInfo.getProcessesCreated(),
        mCpuInfo.getProcessesRunning(),
        mCpuInfo.getProcessesBlocked(),
        mPM.isScreenOn() ? 1 : 0,
        mMoving ? 1 : 0));

    if (mSensorInfo.getAmbientLight() == -1) {
        outputString = outputString.concat(",?");
    }
    else {
        outputString
        = outputString.concat(String.format(",%d",
mSensorInfo.getAmbientLight()));
    }

    // Data Needed?
    newWifiTraffic = mNetInfo.readWifiTraffic();

    if (oldWifiTraffic != -1 && newWifiTraffic != -1) {
        wifiTrafficDiff = newWifiTraffic - oldWifiTraffic;
    }
    else {
        wifiTrafficDiff = 0;
    }

    if (wifiTrafficDiff > 5 ||
        mNetInfo.isNetworkDataActive() == true ||
        internetBeingUsed()) {
        outputString = outputString.concat(",1");
    }
    else {
        outputString = outputString.concat(",0");
    }
}

```

```

// Coarse Location Needed?
if (coarseLocBeingUsed()) {
    outputString = outputString.concat(",1");
}
else {
    outputString = outputString.concat(",0");
}

// Fine Location Needed?
if (fineLocBeingUsed()) {
    outputString = outputString.concat(",1\n");
}
else {
    outputString = outputString.concat(",0\n");
}

//
outputString.concat(String.format(",%d,%d,%.1f,%.1f,%.1f,%d,%d,%d,%d,%s,%d,%d,%d", =
//
//      mBattInfo.getBatteryLevel(),
//      mBattInfo.getBatteryStatus(),
//      mBattInfo.getBatteryTemperatureInDegF(),
//      mBattInfo.getBatteryVoltageInVolts(),
//      mCpuInfo.getUtilizationPct(),
//      mCpuInfo.getContextSwitches(),
//      mCpuInfo.getProcessesCreated(),
//      mCpuInfo.getProcessesRunning(),
//      mCpuInfo.getProcessesBlocked(),
//      foregroundApp == null ? "?" : foregroundApp,
//      mPM.isScreenOn() ? 1 : 0,
//      mScreenBrightness,
//      mMoving ? 1 : 0));

//
//      if (mSensorInfo.getProximity() == -1) {
//          outputString = outputString.concat(",?");
//      }
//      else {
//          outputString = outputString.concat(String.format(",%d",
mSensorInfo.getProximity()));
//      }

//
//      if (mSensorInfo.getAzimuth() == -1) {
//          outputString = outputString.concat(",?");
//      }
//      else {
//          outputString = outputString.concat(String.format(",%d",
mSensorInfo.getAzimuth()));
//      }

//
//      if (mSensorInfo.getPitch() == -181) {
//          outputString = outputString.concat(",?");
//      }
//      else{
//          outputString = outputString.concat(String.format(",%d",
mSensorInfo.getPitch()));
//      }

//
//      if (mSensorInfo.getRoll() == -91) {
//          outputString = outputString.concat(",?");
//      }
//      else {
//          outputString = outputString.concat(String.format(",%d",
mSensorInfo.getRoll()));
//      }

```

```

//          // get call status
//          outputStream = outputStream.concat(String.format("%d",
mNetInfo.getCallStatus()));

//          // get DSP (music on?) status
//          if (mAudioM.isMusicActive() == true) {
//              outputStream = outputStream.concat(String.format(",1\n"));
//          }
//          else {
//              outputStream = outputStream.concat(String.format(",0\n"));
//          }

        out.write(outputString);
        out.close();
    }
    catch (IOException e) {
        Log.e("ContextLoggerService", "Exception appending to log file", e);
    } catch (SettingNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

//          // sleep for 30 seconds.
//          long endTime = System.currentTimeMillis() + 15 * 1000;
//          while (System.currentTimeMillis() < endTime) {
//              synchronized (mBinder) {
//                  try {
//                      mBinder.wait(endTime - System.currentTimeMillis());
//                  } catch (Exception e) {
//                  }
//              }
//          }

        // Done with our work... stop the service!
        ContextLoggerService.this.stopSelf();
    }
};

/**
 * AccelerometerListener callback methods
 */
public void onShake(float force) {}

public void onAccelerationChanged(float x, float y, float z) {}

public void onMove(float metric) {
    mMoving = true;
}

public void onStationary() {
    mMoving = false;
}

// private boolean isRunningService(String processname){
//     if(processname==null)
//         return false;
// }
// RunningServiceInfo service;
// List <RunningServiceInfo> l = mAM.getRunningServices(9999);
// Iterator <RunningServiceInfo> i = l.iterator();
// while(i.hasNext()){

```

```

//         service = i.next();
//         if(service.process.equals(processname))
//             return true;
//     }
//
//     return false;
// }
//
// private String getForegroundAppName() {
//     RunningAppProcessInfo result=null, info=null;
//
//     List <RunningAppProcessInfo> l = mAM.getRunningAppProcesses();
//     Iterator <RunningAppProcessInfo> i = l.iterator();
//     while(i.hasNext()){
//         info = i.next();
//         if(info.importance == RunningAppProcessInfo.IMPORTANCE_FOREGROUND
//             && !isRunningService(info.processName)){
//             result=info;
//             break;
//         }
//     }
//     if (result != null) {
//         return result.processName;
//     }
//     else {
//         return null;
//     }
// }

private boolean fineLocBeingUsed() {
    RunningAppProcessInfo info=null;

    List <RunningAppProcessInfo> l = mAM.getRunningAppProcesses();
    Iterator <RunningAppProcessInfo> i = l.iterator();
    while(i.hasNext()){
        info = i.next();
        if(info.importance == RunningAppProcessInfo.IMPORTANCE_FOREGROUND){
            if
(mPackM.checkPermission(android.Manifest.permission.ACCESS_FINE_LOCATION,
info.processName) == PackageManager.PERMISSION_GRANTED
                && !info.processName.equals("com.android.phone")
                && !info.processName.equals("com.android.systemui")
                && !info.processName.equals("csu.research")) {
                    return true;
                }
            }
        }
    }
    return false;
}

private boolean coarseLocBeingUsed() {
    RunningAppProcessInfo info=null;

    List <RunningAppProcessInfo> l = mAM.getRunningAppProcesses();
    Iterator <RunningAppProcessInfo> i = l.iterator();
    while(i.hasNext()){
        info = i.next();
        if(info.importance == RunningAppProcessInfo.IMPORTANCE_FOREGROUND){
            if
(mPackM.checkPermission(android.Manifest.permission.ACCESS_COARSE_LOCATION,
info.processName) == PackageManager.PERMISSION_GRANTED
                && !info.processName.equals("com.android.phone")
                && !info.processName.equals("com.android.systemui"))

```



```

                && !info.processName.equals("csu.research")) {
                    return true;
                }
            }
        }
        return false;
    }

private boolean internetBeingUsed() {
    RunningAppProcessInfo info=null;

    List <RunningAppProcessInfo> l = mAM.getRunningAppProcesses();
    Iterator <RunningAppProcessInfo> i = l.iterator();
    while(i.hasNext()){
        info = i.next();
        if(info.importance == RunningAppProcessInfo.IMPORTANCE_FOREGROUND){
            if (mPackM.checkPermission(android.Manifest.permission.INTERNET,
info.processName) == PackageManager.PERMISSION_GRANTED
                && !info.processName.equals("com.android.phone")
                && !info.processName.equals("com.android.systemui")
                && !info.processName.equals("csu.research")) {
                    return true;
                }
            }
        }
        return false;
    }

public LocationDatabase getLocationDatabase() {
    return mLocDatabase;
}

private boolean saveDatabase() {
    try {
        if
(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath = String.format("%s/ContextLogger",
Environment.getExternalStorageDirectory());
            if (!new File(filePath).exists()) {
                new File(filePath).mkdirs();
            }
            if (new File(filePath).canWrite()) {
                return mLocDatabase.save(filePath);
            }
        }
        return false;
    }
    catch (Exception e) {
        return false;
    }
}

private boolean loadDatabase() {
    try {
        if
(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            String filePath = String.format("%s/ContextLogger/Locations.ser",
Environment.getExternalStorageDirectory());
            if (new File(filePath).exists()) {
                FileInputStream fis = new FileInputStream(filePath);
                ObjectInputStream in = new ObjectInputStream(fis);
                mLocDatabase = (LocationDatabase)in.readObject();
            }
        }
    }
}

```

```

        in.close();
        return true;
    }
    }
    mLocDatabase = new LocationDatabase();
    return false;
}
catch (Exception e) {
    mLocDatabase = new LocationDatabase();
    return false;
}
}

@Override
public IBinder onBind(Intent intent) {
    return mBinder;
}

/**
 * Show a notification while this service is running.
 */
private void showNotification() {
    // In this sample, we'll use the same text for the ticker and the
    // expanded notification
    CharSequence text = getText(R.string.app_name);

    // Set the icon, scrolling text and timestamp
    Notification notification = new Notification(R.drawable.plan_48,
        text, System.currentTimeMillis());

    Intent notifyIntent = new Intent(this, ContextLogger.class);

    notifyIntent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP
Intent.FLAG_ACTIVITY_SINGLE_TOP);

    // The PendingIntent to launch our activity if the user selects this
    // notification
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
        notifyIntent, 0);

    // Set the info for the views that show in the notification panel.
    notification.setLatestEventInfo(this, getText(R.string.iconized),
        text, contentIntent);

    // Send the notification.
    // We use a layout id because it is a unique number. We use it later to
    // cancel.
    mNM.notify(R.string.iconized, notification);
}

/**
 * This is the object that receives interactions from clients. See
 * RemoteService for a more complete example.
 */
private final IBinder mBinder = new Binder() {
    @Override
    protected boolean onTransact(int code, Parcel data, Parcel reply,
        int flags) throws RemoteException {
        return super.onTransact(code, data, reply, flags);
    }
};
}

```

## A.24 ContextPrediction.py (Strategy 2)

```
#####
# IMPORTS
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d as plt3
import neuralNet as nn
import random
import gradientDescent as gd
import mvSom as mvSom
import kNearestNeighbor as knn
import time

#####
# PARAMETERS
trainingFraction = 0.8
nModels = 1
nHiddenUnits = 18

SYNTHETIC_USER = 0
OVERWRITE_STATES = 0
PARTITION_NN_INPUTS = 0
RUN_LLRLR = 0
RUN_NN = 0
RUN_MVSOM = 1
RUN_KNN = 0
RUN_QDA_LDA = 0
RUN_QDA = 0
PLOT_ERRORS = 0
PLOT_LOC_VS_TIME = 0

LOCATION_PRECISION = 3
LOCATION_THRESHOLD = 60 #minutes (Lump all locations where time spent is < x min)

REAL_USER_INPUT_FILE = '/home/brad/Dropbox/CS 545/Final Assignment/ContextLoggerData/BradsMom3/ContextLoggerLog_BradsMom3.csv'
SYNTHETIC_USER_INPUT_FILE = '/home/brad/Dropbox/CS 545/Final Assignment/Synthetic User Stuff/TheBusyTraveler_SyntheticUser_1.csv'

#####
# FUNCTIONS

#Returns radial location distance precision
def getDistanceRadius(units='m', disp=True):
    #Default units = meters
    if (LOCATION_PRECISION == 6):
        rad = 0.78224
    elif (LOCATION_PRECISION == 5):
        rad = 13.85617
    elif (LOCATION_PRECISION == 4):
        rad = 139.8210749
    elif (LOCATION_PRECISION == 3):
        rad = 1399.436513
    elif (LOCATION_PRECISION == 2):
        rad = 13992.21037
    elif (LOCATION_PRECISION == 1):
        rad = 139578.8669
    else:
        rad = 1226391.488
```

```

#Convert
if (units == 'ft'):
    rad *= 3.2808399
elif (units == 'mi'):
    rad *= 0.000621371192
elif (units == 'km'):
    rad *= 0.001
elif (units == 'in'):
    rad *= 39.3700787
else: #If we get here, than input units were not defined
    units = 'm'
#Print
if (disp):
    print 'Radial Location Precision (%d) = %.6f %s' \
          % (LOCATION_PRECISION, rad, units)

return rad

#Plot Location Count Distribution
def plotLocationDistribution(loccounts):
    #Plot all key location distributions
    plt.figure()
    title = 'Unique Location Distribution\n(%s)' % \
            ('Radial Accuracy = %.5f %s' % (getDistanceRadius(units='mi', disp=False),
'mi'))
    plt.suptitle(title, fontweight='bold', fontsize=14)
    plt.pie(loccounts, autopct='%1.1f%%', colors=('r', 'g', 'b', 'y', 'w', 'm', 'c'),
shadow=True)
    #Plot threshold based locations
    (threshlocs, _) = np.where(loccounts >= LOCATION_THRESHOLD)
    plt.figure()
    title = 'Unique Location Distribution\nTime Spent >= %d Minutes\n(Radial Accuracy
= %.5f %s)' % \
            (LOCATION_THRESHOLD, getDistanceRadius(units='mi', disp=False), 'mi')
    plt.suptitle(title, fontweight='bold', fontsize=14)
    plt.pie(loccounts[threshlocs], autopct='%1.1f%%', colors=('r', 'g', 'b', 'y', 'w',
'm', 'c'), shadow=True)

#Plot Location vs Time
def plotLocationVsTime(time, locmap, loccounts, color='g*'):
    plt.figure()
    plt.plot(time, locmap, color)
    plt.suptitle('Unique Location vs Time', fontweight='bold', fontsize=14)
    plt.xlabel('Time')
    plt.ylabel('Unique Location')
    plt.grid()
    plt.figure()
    plt.bar(range(loccounts.shape[0]), loccounts)

#Plot Targets vs Location
def plotTargetsVsLocation(X, T, K, loccol=2, legend=[]):
    plt.figure()
    plt.plot(X[:,loccol], T, 'r*')
    plt.ylim(0, 9)
    plt.suptitle('Target vs Unique Location', fontweight='bold', fontsize=14)
    plt.xlabel('Unique Location')
    plt.ylabel('States')
    plt.yticks(K, legend, rotation=17)
    plt.grid()

def plotScreenOnVsLocation(X, loccol=2, sccol=16, color='r*'):
    plt.figure()
    indicescreenon = np.where(X[:,sccol] == 1)[0]

```

```

indicescreenoff = np.where(X[:,sccol] == 0)[0]
unqLocs = np.unique(X[:,loccol]).reshape((-1,1))
histon = np.zeros(unqLocs.shape)
histoff = np.zeros(unqLocs.shape)
idx = 0
for loc in unqLocs:
    locindices = set(np.where(X[:,loccol] == loc)[0])
    histon[idx,0] = len(set(indicescreenon).intersection(locindices))
    histoff[idx,0] = len(set(indicescreenoff).intersection(locindices))
    idx += 1

p1 = plt.bar(unqLocs, histoff, color='b')
p2 = plt.bar(unqLocs, histon, color='r', bottom=histoff)
plt.suptitle('Screen Usage vs Unique Location', fontweight='bold', fontsize=14)
plt.xlabel('Unique Location')
plt.ylabel('Screen Usage')
plt.legend( (p1[0], p2[0]), ('Off', 'On') )
plt.grid()

#Gets number of unique locations and respective distributions based on a given
precision
def uniqueLocations(X, cols=(2,3), disp=True, fwrite=True):
    locs = np.around(X[:,cols], decimals=LOCATION_PRECISION)
    unique = locs[0,:]
    #Get number of unique locations
    for i in range(1, X.shape[0]):
        if (not np.any(unique == locs[i,:])):
            unique = np.vstack((unique, locs[i,:]))

    locmap = np.zeros((X.shape[0], 1))
    loccounts = np.zeros((unique.shape[0], 1))
    for i in range(unique.shape[0]):
        (indices, _) = np.where(locs == unique[i,:])
        loccounts[i] = indices.size
        locmap[indices,:] = i

    if (fwrite):
        f = open(NameOfUniqueLocFile, 'w')
        f.write('Latitude,Longitude\n')
    #If disp == True
    if (disp):
        print 'Number of unique locations = %d out of %d' % (unique.shape[0],
X.shape[0])
        print 'Percent of unique locations = %.1f %%' % float(unique.shape[0] *
float(100) / X.shape[0])
        undefinedLocs = np.where(locs == np.array([0, 0]))[0].shape[0]
        print 'Number of undefined locations = %d (%.1f %%)' % (undefinedLocs,
float(undefinedLocs) * 100 / locs.shape[0])
        for i in range(unique.shape[0]):
            print '%.4f %.4f - %d' % (unique[i,0], unique[i,1], loccounts[i])
            if (fwrite):
                f.write('%%.4f,%.4f\n' % (unique[i,0], unique[i,1]))

    if (fwrite):
        f.close()

    return (unique, locmap, loccounts)

#Converts Latitude and Longitude to unique location integer
def convertLocations(X, locmap, loccols=(2,3)):
    return np.hstack((X[:,loccols[0]], np.hstack((locmap, X[:,(loccols[1]+1):])))

# Print Defined/Undefined States

```

```

def printStates(T):
    states = np.unique(targetToStates(T))
    if (states is None or states.size == 0):
        print 'No Available States'
    else:
        #State 0, 0, 0
        if np.any(states == 1):
            print 'Data = 0, Coarse = 0, Fine = 0'
        #State 0, 0, 1
        if np.any(states == 2):
            print 'Data = 0, Coarse = 0, Fine = 1'
        #State 0, 1, 0
        if np.any(states == 3):
            print 'Data = 0, Coarse = 1, Fine = 0'
        #State 0, 1, 1
        if np.any(states == 4):
            print 'Data = 0, Coarse = 1, Fine = 1'
        #State 1, 0, 0
        if np.any(states == 5):
            print 'Data = 1, Coarse = 0, Fine = 0'
        #State 1, 0, 1
        if np.any(states == 6):
            print 'Data = 1, Coarse = 0, Fine = 1'
        #State 1, 1, 0
        if np.any(states == 7):
            print 'Data = 1, Coarse = 1, Fine = 0'
        #State 1, 1, 1
        if np.any(states == 8):
            print 'Data = 1, Coarse = 1, Fine = 1'

def printStateDistributions(T):
    T = targetToStates(T)
    unq = np.unique(T).reshape((-1,1))
    hist = np.zeros(unq.shape)
    idx = 0
    for state in unq:
        hist[idx,0] = np.where(T == state)[0].size
        print 'State %d = %d of %d (%.2f %%)' % (state, hist[idx,0].astype(int),
T.shape[0], \
                                                    hist[idx,0] * float(100) /
T.shape[0])
        idx += 1

def getStateNames(T):
    names = []
    states = np.unique(targetToStates(T))
    if (states is None or states.size == 0):
        print 'No Available States'
    else:
        #State 0, 0, 0
        if np.any(states == 1):
            names.append('D = 0, C = 0, F = 0')
        #State 0, 0, 1
        if np.any(states == 2):
            names.append('D = 0, C = 0, F = 1')
        #State 0, 1, 0
        if np.any(states == 3):
            names.append('D = 0, C = 1, F = 0')
        #State 0, 1, 1
        if np.any(states == 4):
            names.append('D = 0, C = 1, F = 1')
        #State 1, 0, 0
        if np.any(states == 5):

```

```

        names.append('D = 1, C = 0, F = 0')
#State 1, 0, 1
if np.any(states == 6):
    names.append('D = 1, C = 0, F = 1')
#State 1, 1, 0
if np.any(states == 7):
    names.append('D = 1, C = 1, F = 0')
#State 1, 1, 1
if np.any(states == 8):
    names.append('D = 1, C = 1, F = 1')

return names

#Horizontally stack LDA/QDA discriminants
def appendDiscriminant(disc, dn):
    if (disc is None):
        disc = dn
    else:
        disc = np.hstack((disc, dn))
    return disc

# Targets to states
# State 1 = 0, 0, 0
# State 2 = 0, 0, 1
# State 3 = 0, 1, 0
# State 4 = 0, 1, 1
# State 5 = 1, 0, 0
# State 6 = 1, 0, 1
# State 7 = 1, 1, 0
# State 8 = 1, 1, 1
def targetToStates(T):
    if (T.shape[1] == 3):
        return np.dot(T, np.array([4, 2, 1]).reshape(3,1)) + 1
    elif (T.shape[1] == 2):
        return np.dot(T, np.array([2, 1]).reshape(2,1)) + 1
    else:
        return (T + 1).reshape(-1,1)

def createStateMap(T):
    map = []
    if np.any(T == 1):
        map.append(1)
    if np.any(T == 2):
        map.append(2)
    if np.any(T == 3):
        map.append(3)
    if np.any(T == 4):
        map.append(4)
    if np.any(T == 5):
        map.append(5)
    if np.any(T == 6):
        map.append(6)
    if np.any(T == 7):
        map.append(7)
    if np.any(T == 8):
        map.append(8)
    return map

def reduceStateSpace(T, data=True, coarseloc=True, fineloc=False):
    Tnew = np.zeros((T.shape[0], 1))
    if (data):
        Tnew = np.hstack((Tnew, T[:, :1]))
    if (coarseloc):

```

```

        Tnew = np.hstack((Tnew, T[:,1:2]))
    if (fineloc):
        Tnew = np.hstack((Tnew, T[:,2:]))
    return Tnew[:,1:]

# Partition into training and testing sets
def partitionSets(X, T, trainFraction):
    nSamples = X.shape[0]
    allI = xrange(nSamples) # indices for all data rows
    nTrain = int(round(nSamples*trainFraction)) # number of training samples
    trainI = list(set(random.sample(allI,nTrain))) # row indices for training samples
    testI = list(set(allI).difference(set(trainI))) # row indices for testing samples
    # use lists of indices to select appropriate rows
    Xtrain = X[trainI,:]
    Ttrain = T[trainI,:]
    Xtest = X[testI,:]
    Ttest = T[testI,:]
    return ((Xtrain, Xtest), (Ttrain, Ttest))

def makeStandardize(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    def standardize(origX):
        return (origX - means) / stds
    def unstandardize(stdX):
        return stds * stdX + means
    return (standardize, unstandardize)

# Add constant column of 1's
def addOnes(A):
    return np.hstack((np.ones((A.shape[0],1)),A))

def makeIndicatorVars(T):
    # Make sure T is two-dimensional. Should be nSamples x 1.
    if T.ndim == 1:
        T = T.reshape((-1,1))
    return (T == np.unique(T)).astype(int)

def getMuAndSigma(X):
    if (X.shape[0] == 0):
        return (0,0)
    if X.shape[1] == 1:
        mu = np.mean(X[0],axis=0)
    else:
        mu = np.mean(X,axis=0).reshape(-1,1)
    Sigma = np.cov(X.T)
    return (mu,Sigma)

def discQDA(X, mu, Sigma, prior):
    # Assumes X is already standardized
    Xc = np.asarray(X) - np.asarray(mu).reshape(1,-1)
    if Sigma.size == 1:
        Sigma = np.asarray(Sigma).reshape((1,1))
    SigmaInv = np.linalg.inv(Sigma)
    return -0.5 * np.log(np.linalg.det(Sigma)) \
        - 0.5 * np.sum(np.dot(Xc,SigmaInv) * Xc, axis=1) \
        + np.log(prior)

def discLDA(X, mu, Sigma, prior):
    # Assumes X is already standardized
    mu = np.asarray(mu).reshape(1,-1)
    if Sigma.size == 1:
        Sigma = np.asarray(Sigma).reshape((1,1))

```



```

SigmaInv = np.linalg.inv(Sigma)
return np.dot(np.dot(mu,SigmaInv), X.T) - 0.5 * np.dot(np.dot(mu,SigmaInv), mu.T)
+ np.log(prior)

# For multiple samples, for any dimension, including 1
def normald(X, mu=None, sigma=None):
    """ normald:
        X contains samples, one per row, NxN.
        mu is mean vector, Dx1.
        sigma is covariance matrix, DxN. """
    d = X.shape[1]
    if np.any(mu == None):
        mu = np.zeros((d,1))
    if np.any(sigma == None):
        sigma = np.eye(d)
    if d == 1:
        detSigma = sigma
        sigmaI = 1.0/sigma
    else:
        detSigma = np.linalg.det(sigma)
        sigmaI = np.linalg.inv(sigma)
    normConstant = 1.0 / np.sqrt((2*np.pi)**d * detSigma)
    diffv = X - mu.T # change column vector mu to be row vector
    return normConstant * np.exp(-0.5 * np.sum(np.dot(diffv, sigmaI) * diffv,
axis=1))[:,np.newaxis]

def percentCorrect(Tpred,T):
    (res, _) = np.where(Tpred == T)
    return res.size * float(100) / T.shape[0]

def falsePosNeg(Tpred,T):
    (pos, _) = np.where(Tpred == 1)
    notPos = 0
    for i in pos:
        if T[i] == 0:
            notPos += 1

    (neg, _) = np.where(Tpred == 0)
    notNeg = 0
    for i in neg:
        if T[i] == 1:
            notNeg += 1

    falsePos = notPos * float(100) / T.shape[0]
    falseNeg = notNeg * float(100) / T.shape[0]
    return (falsePos, falseNeg)

def g(X,beta):
    fs = np.exp(np.dot(X, beta)) # N x K-1
    denom = 1 + np.sum(fs,axis=1)
    denom = denom.reshape(-1,1)
    gs = fs / denom
    return np.hstack((gs,1/denom))

def packBeta(beta):
    return beta.flatten()

def unpackBeta(beta):
    # Assumes XtrainStd1 and TtrainInd are defined in calling context
    return beta.reshape( (XtrainStd1.shape[1],TtrainInd.shape[1]-1) )

def objectivef(beta):
    # Assumes XtrainStd1 and TtrainInd are defined in calling context

```

```

    betaUnpacked = unpackBeta(beta)
    gs = g(XtrainStd1,betaUnpacked)
    return -np.mean(TtrainInd * np.log(gs))

def gradf(beta):
    # Assumes XtrainStd1 and TtrainInd are defined in calling context
    betaUnpacked = unpackBeta(beta)
    gs = g(XtrainStd1,betaUnpacked)
    betaNew = -np.dot(XtrainStd1.T, TtrainInd[:, :-1] - gs[:, :-1])
    return packBeta(betaNew)

# setup
plt.ioff() # turn off interactive mode to view...

def missingFloatIsNan(s):
    if s == '?':
        return np.nan
    else:
        return float(s)

def missingFloatIsNegInf(s):
    if s == '?':
        return float(-1000)
    else:
        return float(s)

def missingStringIsNan(s):
    if s == '?':
        return np.nan
    else:
        return s

def missingFloatIsZero(s):
    if s == '?':
        return float(0)
    else:
        return s

def available(s):
    if s == '?':
        return float(0)
    else:
        return float(1)

NameOfUniqueLocFile = '/home/brad/Dropbox/CS 545/Final Assignment/Unique
Locations/UniqueLocs.csv'
NameOfOutputStatesFile = '/home/brad/Dropbox/CS 545/Final Assignment/OutputStates.csv'

if SYNTHETIC_USER:
    inputNames = [
        'Bias',
        'Day',
        'Time',
        'Location',
        'Satellites',
        'WiFi RSSI',
        'Network SS',
    ]

    outputNames = [
        'Data?',
        'Coarse Location?',
        'Fine Location?'
    ]

```

```

    ]

    inputData = np.loadtxt(SYNTHETIC_USER_INPUT_FILE, skiprows=1, delimiter = ',',
        usecols=(0,1,2,3,4,5,6,7,8,9))

    X = inputData[:, :7]
    T = inputData[:, 7:]
else:
    inputNames = [
        'Bias',
        'Day',
        'Time',
        'Location',
        'Satellites',
        'WiFi APs',
        'WiFi RSSI',
        'Network Signal Strength',
        'Call State',
        'Battery Level',
        'Battery Status',
        'CPU Utilization',
        'Context Switches',
        'Processes Created',
        'Processes Running',
        'Processes Blocked',
        'Screen On?',
        'Device Moving?',
        'Ambient Light'
    ]

    outputNames = [
        'Data?',
        'Coarse Location?',
        'Fine Location?'
    ]

    inputData = np.loadtxt(REAL_USER_INPUT_FILE, skiprows=1, delimiter = ',', \
        usecols=(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21), \
        converters={6: missingFloatIsNegInf,
                    2: missingFloatIsNan,
                    3: missingFloatIsNan,
                    4: missingFloatIsZero,
                    5: missingFloatIsZero,
                    18: missingFloatIsZero})

    X = inputData[:, :19]
    T = inputData[:, 19:]

    # remove rows with location = 0,0
    notNans = np.isnan(X) == False
    goodRowsMask = notNans.all(axis=1)
    X = X[goodRowsMask, :]
    T = T[goodRowsMask, :]

T = reduceStateSpace(T, data=True, coarseloc=True, fineloc=True)

stateNames = getStateNames(T)

#Get unique locations
(uniqueLoc, locmap, loccounts) = uniqueLocations(X)

```

```

#Convert Locations
X = convertLocations(X, locmap)

# Partition data into training and testing sets
((Xtrain, Xtest), (Ttrain, Ttest)) = partitionSets(X, T, trainingFraction)
nTrain = float(Xtrain.shape[0])
nTrainAttr = Xtrain.shape[1]
nTest = float(Xtest.shape[0])

nAttrs = X.shape[1]
nSamples = X.shape[0]

#Standardize
(standardize, _) = makeStandardize(Xtrain)
XtrainStd = standardize(Xtrain)
XtestStd = standardize(Xtest)

Ttrain = targetToStates(Ttrain)
Ttest = targetToStates(Ttest)

if OVERWRITE_STATES:
    for i in xrange(Ttrain.shape[0]):
        if random.random() > 0.3:
            if Xtrain[i,15] == 0:
                Ttrain[i] = 1
    for i in xrange(Ttest.shape[0]):
        if random.random() > 0.3:
            if Xtest[i,15] == 0:
                Ttest[i] = 1

statesMap = createStateMap(Ttrain)

#Convert to indicator vars (e.g. 1,2,3 --> 1,0,0 - 0,1,0 - 0,0,1)
TtrainInd = makeIndicatorVars(Ttrain)

if RUN_LLRL:
    XtrainStd1 = addOnes(XtrainStd)
    XtestStd1 = addOnes(XtestStd)

    beta = np.zeros((XtrainStd1.shape[1],TtrainInd.shape[1]-1))

    # Steepest descent for LLR
    # alpha = 0.001
    # for step in range(1000) :
    #     gs = g(XtrainStd1,beta)
    #     beta = beta + alpha * np.dot(XtrainStd1.T, TtrainInd[:, :-1] - gs[:, :-1])
    #     likelihoodPerSample = np.exp(np.sum(TtrainInd * np.log(gs)) /
XtrainStd1.shape[0])
    #     print "Step",step," l =",likelihoodPerSample
    #     logregOutput = g(XtrainStd1,beta)
    #     predictedTrain = np.argmax(logregOutput,axis=1).reshape(-1,1)
    #     print "Percent correct: Train %.2f" % percentCorrect(predictedTrain,Ttrain-1)
    #     logregOutput = g(XtestStd1,beta)
    #     predictedTest = np.argmax(logregOutput,axis=1).reshape(-1,1)

    # SCG for LLR
    gs = g(XtrainStd1,beta)
    scgresult = gd.scg(packBeta(beta),
                        objectivef,
                        gradf,
                        nIterations = 1000,
                        xPrecision = 1.e-10,

```

```

        fPrecision = 1.e-10,
        ftracep=True)
scgresultx = scgresult['x']
scgresultxUnpacked = unpackBeta(scgresultx)
reason = scgresult['reason']
errorTrace = scgresult['ftrace']
xTrace = scgresult['xtrace']
numberOfIterations = len(errorTrace)
print "SCG stopped after",numberOfIterations,"iterations:",reason

startTime = time.time()

# use SCG result on testing data
logregOutput = g(XtestStd1,scgresultxUnpacked)
llrPredictedTest = np.argmax(logregOutput,axis=1).reshape(-1,1)

llrTime = time.time() - startTime

# Map predicted states back to actual states
for i in xrange(0, Ttest.shape[0]):
    llrPredictedTest[i] = statesMap[llrPredictedTest[i]]

llrPercentCorrectTest = percentCorrect(llrPredictedTest,Ttest)
print "LLR percent correct: Test %.2f" % llrPercentCorrectTest
print "LLR time: %.5f" % llrTime

outputFile = open(NameOfOutputStatesFile, 'w')
outputFile.write('LLR\n')
outputFile.write('Actual,Predicted\n')
for i in xrange(Ttest.shape[0]):
    outputFile.write('%d,%d\n' % (Ttest[i], llrPredictedTest[i]))
outputFile.close()

if RUN_NN:
#####
# NONLINEAR (NEURAL NETWORK) LOGISTIC REGRESSION

if PARTITION_NN_INPUTS:
    # Partition input data into temporal, spatial, and device subsets
    XtemporalTrain = Xtrain[:,(0,1)]
    XspatialTrain = Xtrain[:,(2,3,4,5,6,16,17)]
    XdeviceTrain = Xtrain[:,(7,8,9,10,11,12,13,14,15)]
    XtemporalTest = Xtest[:,(0,1)]
    XspatialTest = Xtest[:,(2,3,4,5,6,16,17)]
    XdeviceTest = Xtest[:,(7,8,9,10,11,12,13,14,15)]

    nTemporalInputs = XtemporalTrain.shape[1]
    nSpatialInputs = XspatialTrain.shape[1]
    nDeviceInputs = XdeviceTrain.shape[1]
    nOutputs = TtrainInd.shape[1] # nOutputs = number of classes

    # Initialize neural network
    temporalNnet = nn.NeuralNetClassifier(nTemporalInputs,nHiddenUnits,nOutputs)

    # Train neural network
    temporalNnet.train(XtemporalTrain,TtrainInd,nTrain,weightPrecision=1.e-
8,errorPrecision=1.e-8,nIterations=2000)
    print                    "SCG                    stopped
after",temporalNnet.getNumberOfIterations(),"iterations:",temporalNnet.reason

    nnXtrace = temporalNnet.getXTrace()

    # Use neural network on test data

```

```

        (nnTemporalPredictedTest,Ztest)          =          temporalNnet.use(XtemporalTest,
allOutputs=True)

# Map predicted states back to actual states
for i in xrange(0, Ttest.shape[0]):
    nnTemporalPredictedTest[i] = statesMap[nnTemporalPredictedTest[i]]

nnTemporalPercentCorrectTest = percentCorrect(nnTemporalPredictedTest,Ttest)

# Initialize neural network
spatialNnet = nn.NeuralNetClassifier(nSpatialInputs,nHiddenUnits,nOutputs)

# Train neural network
spatialNnet.train(XspatialTrain,TtrainInd,nTrain,weightPrecision=1.e-
8,errorPrecision=1.e-8,nIterations=2000)
print                                "SCG                                stopped
after",spatialNnet.getNumberOfIterations(),"iterations:",spatialNnet.reason

nnXtrace = spatialNnet.getXTrace()

# Use neural network on test data
(nnSpatialPredictedTest,Ztest)          =          spatialNnet.use(XspatialTest,
allOutputs=True)

# Map predicted states back to actual states
for i in xrange(0, Ttest.shape[0]):
    nnSpatialPredictedTest[i] = statesMap[nnSpatialPredictedTest[i]]

nnSpatialPercentCorrectTest = percentCorrect(nnSpatialPredictedTest,Ttest)

# Initialize neural network
deviceNnet = nn.NeuralNetClassifier(nDeviceInputs,nHiddenUnits,nOutputs)

# Train neural network
deviceNnet.train(XdeviceTrain,TtrainInd,nTrain,weightPrecision=1.e-
8,errorPrecision=1.e-8,nIterations=2000)
print                                "SCG                                stopped
after",deviceNnet.getNumberOfIterations(),"iterations:",deviceNnet.reason

nnXtrace = deviceNnet.getXTrace()

# Use neural network on test data
(nnDevicePredictedTest,Ztest) = deviceNnet.use(XdeviceTest, allOutputs=True)

# Map predicted states back to actual states
for i in xrange(0, Ttest.shape[0]):
    nnDevicePredictedTest[i] = statesMap[nnDevicePredictedTest[i]]

nnDevicePercentCorrectTest = percentCorrect(nnDevicePredictedTest,Ttest)

print "Temporal NN percent correct: Test %.2f" % nnTemporalPercentCorrectTest
print "Spatial NN percent correct: Test %.2f" % nnSpatialPercentCorrectTest
print "Device NN percent correct: Test %.2f" % nnDevicePercentCorrectTest

temporalError = temporalNnet.getErrorTrace()
spatialError = spatialNnet.getErrorTrace()
deviceError = deviceNnet.getErrorTrace()
minError = min(temporalError[-1],spatialError[-1],deviceError[-1])
if (minError == temporalError[-1]):
    nnMinErrorPredictedTest = nnTemporalPredictedTest
elif (minError == spatialError[-1]):

```

```

        nnMinErrorPredictedTest = nnSpatialPredictedTest
    elif (minError == deviceError[-1]):
        nnMinErrorPredictedTest = nnDevicePredictedTest

    # Perform Voting
    nnPredictedTest = np.zeros(Ttest.shape)
    for i in xrange(0, Ttest.shape[0]):
        if (nnTemporalPredictedTest[i] == nnSpatialPredictedTest[i]):
            nnPredictedTest[i] = nnTemporalPredictedTest[i]
        elif (nnTemporalPredictedTest[i] == nnDevicePredictedTest[i]):
            nnPredictedTest[i] = nnTemporalPredictedTest[i]
        elif (nnSpatialPredictedTest[i] == nnDevicePredictedTest[i]):
            nnPredictedTest[i] = nnSpatialPredictedTest[i]
        else:
            # Choose output with lowest error (or highest percent correct???)
            nnPredictedTest[i] = nnMinErrorPredictedTest[i]

    nnPercentCorrectTest = percentCorrect(nnPredictedTest, Ttest)
    print "Combined NN percent correct: Test %.2f" % nnPercentCorrectTest

nInputs = Xtrain.shape[1]
nOutputs = TtrainInd.shape[1] # nOutputs = number of classes

nnetVs = np.zeros((nModels, nAttrs+1))
nnetTestPctCorrect = np.zeros((nModels, 1))

# Bootstrap to
for booti in xrange(nModels):
    # Initialize neural network
    nnet = nn.NeuralNetClassifier(nInputs, nHiddenUnits, nOutputs)

    # Train neural network
    nnet.train(Xtrain, TtrainInd, nTrain, weightPrecision=1.e-8, errorPrecision=1.e-
8, nIterations=2000)
    print                                     "SCG                                     stopped
after", nnet.getNumberOfIterations(), "iterations:", nnet.reason
    # sum the absolute values of the hidden unit weights for each input attribute
    nnetVs[booti, :] = np.sum(np.abs(nnet.getV()), axis=1).reshape(1, -1)

    nnXtrace = nnet.getXTrace()

    startTime = time.time()

    # Use neural network on test data
    (nnPredictedTest, Ztest) = nnet.use(Xtest, allOutputs=True)

    # Map predicted states back to actual states
    for i in xrange(0, Ttest.shape[0]):
        nnPredictedTest[i] = statesMap[nnPredictedTest[i]]

    nnTime = time.time() - startTime

    nnPercentCorrectTest = percentCorrect(nnPredictedTest, Ttest)
    print "NN percent correct: Test %.2f" % nnPercentCorrectTest
    nnetTestPctCorrect[booti, 0] = nnPercentCorrectTest
    print "NN time: %.5f" % nnTime

    outputFile = open(NameOfOutputStatesFile, 'w')
    outputFile.write('NN\n')
    outputFile.write('Actual, Predicted\n')
    for i in xrange(Ttest.shape[0]):
        outputFile.write('%d,%d\n' % (Ttest[i], nnPredictedTest[i]))
    outputFile.close()

```

```

#####Resample#####
# Partition data into training and testing sets
((Xtrain, Xtest), (Ttrain, Ttest)) = partitionSets(X, T, trainingFraction)
nTrain = float(Xtrain.shape[0])
nTrainAttr = Xtrain.shape[1]
nTest = float(Xtest.shape[0])

nAttrs = X.shape[1]
nSamples = X.shape[0]

#Standardize
(standardize, _) = makeStandardize(Xtrain)
XtrainStd = standardize(Xtrain)
XtestStd = standardize(Xtest)

Ttrain = targetToStates(Ttrain)
Ttest = targetToStates(Ttest)
#####Resample#####

print 'Average Test Percent Correct over %d models = %.2f %%' % (nModels,
np.mean(nnetTestPctCorrect))

plt.figure()
plt.subplot(2,1,1)
nnet.plotErrors()

plt.subplot(2,1,2)
plt.plot(nnXtrace)
plt.ylabel("Weights")
plt.xlabel("Iterations")

plt.figure()
plt.boxplot(nnetVs)
plt.xlabel("Attribute")
plt.xticks(np.arange(1, nAttrs+2),inputNames,rotation=17)
plt.title('Distribution of Weight Values')
plt.show()

if RUN_MVSOM:
#####
# MISSING-VALUES SELF-ORGANIZING MAP (MVSOM)
som = mvsom.SOM(20,20,nAttrs)
som.learn(XtrainStd,Ttrain,epochs=10000)

startTime = time.time()

somPredictedTest = som.use(XtestStd).astype('int')

somTime = time.time() - startTime

somPercentCorrectTest = percentCorrect(somPredictedTest,Ttest)
# (somFalsePos, somFalseNeg) = falsePosNeg(somPredictedTest,Ttest)
print "SOM percent correct: Test %.2f" % somPercentCorrectTest
# print "False Pos: %.2f, False Neg: %.2f" % (somFalsePos, somFalseNeg)
print "SOM time: %.5f" % somTime

outputFile = open(NameOfOutputStatesFile, 'w')
outputFile.write('MVSOM\n')
outputFile.write('Actual,Predicted\n')
for i in xrange(Ttest.shape[0]):
    outputFile.write('%d,%d\n' % (Ttest[i], somPredictedTest[i]))
outputFile.close()

```



```

if RUN_KNN:
    kNN = knn.kNN()
    kNN.train(XtrainStd, Ttrain, 30)

    startTime = time.time()

    knnPredictedTest = kNN.classifySet(XtestStd)

    knnTime = time.time() - startTime

    knnPercentCorrectTest = percentCorrect(knnPredictedTest, Ttest)
#   (knnFalsePos, knnFalseNeg) = falsePosNeg(knnPredictedTest, Ttest)
    print "KNN percent correct: Test %.2f" % knnPercentCorrectTest
#   print "False Pos: %.2f, False Neg: %.2f" % (knnFalsePos, knnFalseNeg)
    print "KNN time: %.5f" % knnTime

    outputFile = open(NameOfOutputStatesFile, 'w')
    outputFile.write('KNN\n')
    outputFile.write('Actual, Predicted\n')
    for i in xrange(Ttest.shape[0]):
        outputFile.write('%d,%d\n' % (Ttest[i], knnPredictedTest[i]))
    outputFile.close()

if RUN_QDA_LDA:
    #####-----QDA-----#####
    #Class Attributes
    c1train = Xtrain[np.where(Ttrain == 1)[0], :].reshape(-1, nTrainAttr)
    c2train = Xtrain[np.where(Ttrain == 2)[0], :].reshape(-1, nTrainAttr)
    c3train = Xtrain[np.where(Ttrain == 3)[0], :].reshape(-1, nTrainAttr)
    c4train = Xtrain[np.where(Ttrain == 4)[0], :].reshape(-1, nTrainAttr)
    c5train = Xtrain[np.where(Ttrain == 5)[0], :].reshape(-1, nTrainAttr)
    c6train = Xtrain[np.where(Ttrain == 6)[0], :].reshape(-1, nTrainAttr)
    c7train = Xtrain[np.where(Ttrain == 7)[0], :].reshape(-1, nTrainAttr)
    c8train = Xtrain[np.where(Ttrain == 8)[0], :].reshape(-1, nTrainAttr)

    #Class likelihood
    #p(Class = k) = Nk / N
    prior1 = c1train.shape[0] / nTrain
    prior2 = c2train.shape[0] / nTrain
    prior3 = c3train.shape[0] / nTrain
    prior4 = c4train.shape[0] / nTrain
    prior5 = c5train.shape[0] / nTrain
    prior6 = c6train.shape[0] / nTrain
    prior7 = c7train.shape[0] / nTrain
    prior8 = c8train.shape[0] / nTrain

    #Use QDA discriminant to calculate probability
    (mu1, Sigma1) = getMuAndSigma(standardize(c1train))
    (mu2, Sigma2) = getMuAndSigma(standardize(c2train))
    (mu3, Sigma3) = getMuAndSigma(standardize(c3train))
    (mu4, Sigma4) = getMuAndSigma(standardize(c4train))
    (mu5, Sigma5) = getMuAndSigma(standardize(c5train))
    (mu6, Sigma6) = getMuAndSigma(standardize(c6train))
    (mu7, Sigma7) = getMuAndSigma(standardize(c7train))
    (mu8, Sigma8) = getMuAndSigma(standardize(c8train))

    disc = None

    if RUN_QDA:
        #Classes
        K = np.unique(Ttrain)

```

```

startTime = time.time()

# QDA Discriminants
if np.any(K == 1):
    d1 = discQDA(XtestStd, mu1, Sigma1, prior1).reshape(-1, 1)
    disc = appendDiscriminant(disc, d1)
if np.any(K == 2):
    d2 = discQDA(XtestStd, mu2, Sigma2, prior2).reshape(-1, 1)
    disc = appendDiscriminant(disc, d2)
if np.any(K == 3):
    d3 = discQDA(XtestStd, mu3, Sigma3, prior3).reshape(-1, 1)
    disc = appendDiscriminant(disc, d3)
if np.any(K == 4):
    d4 = discQDA(XtestStd, mu4, Sigma4, prior4).reshape(-1, 1)
    disc = appendDiscriminant(disc, d4)
if np.any(K == 5):
    d5 = discQDA(XtestStd, mu5, Sigma5, prior5).reshape(-1, 1)
    disc = appendDiscriminant(disc, d5)
if np.any(K == 6):
    d6 = discQDA(XtestStd, mu6, Sigma6, prior6).reshape(-1, 1)
    disc = appendDiscriminant(disc, d6)
if np.any(K == 7):
    d7 = discQDA(XtestStd, mu7, Sigma7, prior7).reshape(-1, 1)
    disc = appendDiscriminant(disc, d7)
if np.any(K == 8):
    d8 = discQDA(XtestStd, mu8, Sigma8, prior8).reshape(-1, 1)
    disc = appendDiscriminant(disc, d8)

# Predicted class from QDA
qdaPredictedTest = np.argmax(disc, axis=1).reshape(-1, 1)

qdaTime = time.time() - startTime

qdaPercentCorrectTest = percentCorrect(K[qdaPredictedTest,:],Ttest)
print "QDA percent correct: Test %.2f" % qdaPercentCorrectTest
print "QDA time: %.5f" % qdaTime

outputFile = open(NameOfOutputStatesFile, 'w')
outputFile.write('QDA\n')
outputFile.write('Actual,Predicted\n')
for i in xrange(Ttest.shape[0]):
    outputFile.write('%d,%d\n' % (Ttest[i], K[qdaPredictedTest[i]]))
outputFile.close()
else:
#Classes
K = np.unique(Ttrain)

Sigma = 0
#Sigma
if np.any(K == 1):
    Sigma += Sigma1 * prior1
if np.any(K == 2):
    Sigma += Sigma2 * prior2
if np.any(K == 3):
    Sigma += Sigma3 * prior3
if np.any(K == 4):
    Sigma += Sigma4 * prior4
if np.any(K == 5):
    Sigma += Sigma5 * prior5
if np.any(K == 6):
    Sigma += Sigma6 * prior6
if np.any(K == 7):
    Sigma += Sigma7 * prior7

```

```

if np.any(K == 8):
    Sigma += Sigma8 * prior8

startTime = time.time()

# LDA Discriminants
if np.any(K == 1):
    d1 = discLDA(XtestStd, mu1, Sigma, prior1).reshape(-1, 1)
    disc = appendDiscriminant(disc, d1)
if np.any(K == 2):
    d2 = discLDA(XtestStd, mu2, Sigma, prior2).reshape(-1, 1)
    disc = appendDiscriminant(disc, d2)
if np.any(K == 3):
    d3 = discLDA(XtestStd, mu3, Sigma, prior3).reshape(-1, 1)
    disc = appendDiscriminant(disc, d3)
if np.any(K == 4):
    d4 = discLDA(XtestStd, mu4, Sigma, prior4).reshape(-1, 1)
    disc = appendDiscriminant(disc, d4)
if np.any(K == 5):
    d5 = discLDA(XtestStd, mu5, Sigma, prior5).reshape(-1, 1)
    disc = appendDiscriminant(disc, d5)
if np.any(K == 6):
    d6 = discLDA(XtestStd, mu6, Sigma, prior6).reshape(-1, 1)
    disc = appendDiscriminant(disc, d6)
if np.any(K == 7):
    d7 = discLDA(XtestStd, mu7, Sigma, prior7).reshape(-1, 1)
    disc = appendDiscriminant(disc, d7)
if np.any(K == 8):
    d8 = discLDA(XtestStd, mu8, Sigma, prior8).reshape(-1, 1)
    disc = appendDiscriminant(disc, d8)

# Predicted class from LDA
ldaPredictedTest = np.argmax(disc, axis=1).reshape(-1, 1)

ldaTime = time.time() - startTime

ldaPercentCorrectTest = percentCorrect(K[ldaPredictedTest,:],Ttest)
print "LDA percent correct: Test %.2f" % ldaPercentCorrectTest
print "LDA time: %.5f" % ldaTime

outputFile = open(NameOfOutputStatesFile, 'w')
outputFile.write('LDA\n')
outputFile.write('Actual,Predicted\n')
for i in xrange(Ttest.shape[0]):
    outputFile.write('%d,%d\n' % (Ttest[i], K[ldaPredictedTest[i]]))
outputFile.close()

if PLOT_LOC_VS_TIME:
    rad = getDistanceRadius(units='mi')
    time = (X[:,0] - 1) * 1440 + X[:,1]
    #plotLocationVsTime(time, locmap, loccounts)

    #plotTargetsVsLocation(X, targetToStates(T), K, legend=stateNames)

    printStateDistributions(T)
    plotScreenOnVsLocation(X, sccol=X.shape[1]-1)
    #plotLocationDistribution(loccounts)
    printStates(T)

plt.show()

#if PLOT_ERRORS:
#    plt.figure()

```

```

# ind = np.arange(3) # the x locations for the groups
# width = 0.35 # the width of the bars: can also be len(x) sequence
# correct = (nnPercentCorrectTest,somPercentCorrectTest,knnPercentCorrectTest)
# falseP = (nnFalsePos,somFalsePos,knnFalsePos)
# falseN = (nnFalseNeg,somFalseNeg,knnFalseNeg)
#
# falseNbottom =
(nnPercentCorrectTest+nnFalsePos,somPercentCorrectTest+somFalsePos,knnPercentCorrectTe
st+knnFalsePos)
# p1 = plt.bar(ind, correct, width, color='b')
# p2 = plt.bar(ind, falseP, width, color='g', bottom=correct)
# p3 = plt.bar(ind, falseN, width, color='r', bottom=falseNbottom)
#
# plt.ylabel('Percent')
# plt.title('WiFi Availability Prediction Results')
# plt.xticks(ind+width/2., ('NN', 'SOM', 'KNN') )
# plt.yticks(np.arange(60,101,10))
# plt.ylim(60,100)
# plt.legend( (p1[0], p2[0], p3[0]), ('Correct', 'False Pos.', 'False Neg.'),
loc='lower right' )
# plt.show()

raw_input('...')

```

## A.25 kNearestNeighbor.py (Strategy 2)

```

#!/usr/bin/env python
# This code is part of the Biopython distribution and governed by its
# license. Please see the LICENSE file that should have been included
# as part of this package.
"""
This module provides code for doing k-nearest-neighbors classification.

k Nearest Neighbors is a supervised learning algorithm that classifies
a new observation based the classes in its surrounding neighborhood.

Glossary:
distance The distance between two points in the feature space.
weight The importance given to each point for classification.

Classes:
kNN Holds information for a nearest neighbors classifier.

Functions:
train Train a new kNN classifier.
calculate Calculate the probabilities of each class, given an observation.
classify Classify an observation into a class.

Weighting Functions:
equal_weight Every example is given a weight of 1.
"""

import numpy as np

class kNN:
    """Holds information necessary to do nearest neighbors classification.

    Members:
    classes Set of the possible classes.

```

```

xs      List of the neighbors.
ys      List of the classes that the neighbors belong to.
k       Number of neighbors to look at.

"""
def __init__(self):
    """kNN()"""
    self.classes = set()
    self.xs = []
    self.ys = []
    self.k = None

def equal_weight(self, x, y):
    """equal_weight(x, y) -> 1"""
    # everything gets 1 vote
    return 1

def train(self, xs, ys, k, typecode=None):
    """train(xs, ys, k) -> kNN

    Train a k nearest neighbors classifier on a training set. xs is a
    list of observations and ys is a list of the class assignments.
    Thus, xs and ys should contain the same number of elements. k is
    the number of neighbors that should be examined when doing the
    classification.

    """
    self.classes = set(ys.flat)
    self.xs = np.asarray(xs, typecode)
    self.ys = ys.flat
    self.k = k

def calculate(self, x, weight_fn=equal_weight, distance_fn=None):
    """calculate(knn, x[, weight_fn][, distance_fn]) -> weight dict

    Calculate the probability for each class. knn is a kNN object. x
    is the observed data. weight_fn is an optional function that
    takes x and a training example, and returns a weight. distance_fn
    is an optional function that takes two points and returns the
    distance between them. If distance_fn is None (the default), the
    Euclidean distance is used. Returns a dictionary of the class to
    the weight given to the class.

    """
    x = np.asarray(x)

    order = [] # list of (distance, index)
    if distance_fn:
        for i in range(len(self.xs)):
            dist = distance_fn(x, self.xs[i])
            order.append((dist, i))
    else:
        # Default: Use a fast implementation of the Euclidean distance
        temp = np.zeros(len(x))
        # Predefining temp allows reuse of this array, making this
        # function about twice as fast.
        for i in range(len(self.xs)):
            temp[:] = x - self.xs[i]
            dist = np.sqrt(np.dot(temp, temp))
            order.append((dist, i))
    order.sort()

    # first 'k' are the ones I want.

```

```

weights = {} # class -> number of votes
for k in self.classes:
    weights[k] = 0.0
for dist, i in order[:self.k]:
    klass = self.ys[i]
    weights[klass] = weights[klass] + weight_fn(self, x, self.xs[i])

return weights

def classify(self, x, weight_fn=equal_weight, distance_fn=None):
    """classify(knn, x[, weight_fn][, distance_fn]) -> class

    Classify an observation into a class. If not specified, weight_fn will
    give all neighbors equal weight. distance_fn is an optional function
    that takes two points and returns the distance between them. If
    distance_fn is None (the default), the Euclidean distance is used.
    """
    weights = self.calculate(x, weight_fn=weight_fn, distance_fn=distance_fn)

    most_class = None
    most_weight = None
    for klass, weight in weights.items():
        if most_class is None or weight > most_weight:
            most_class = klass
            most_weight = weight
    return most_class

def classifySet(self, data):
    predictions = np.zeros((data.shape[0],1))
    for i in xrange(data.shape[0]):
        predictions[i,0] = self.classify(data[i,:])
    return predictions

```

## A.26 mvsom.py (Strategy 2)

```

# -----
# Missing Value Self-Organizing Map
# Brad Donohoo
# September 26, 2011
# -----

# -----
# IMPORTS
# -----
import numpy as np

class SOM:
    ''' Self-organizing map '''
    def __init__(self, *args):
        ''' Initialize som '''
        self.refVectors = np.zeros(args)
        self.predictions = np.zeros((args[0],args[1]))
        self.reset()

    def reset(self):
        ''' Reset weights '''
        self.refVectors = np.random.random(self.refVectors.shape)
        self.predictions = np.random.random(self.predictions.shape)

# Subtract function that handles missing values
def subtract(self, data):

```

```

    sub = self.refVectors - data;
    # Convert all nans to 0s
    sub = np.nan_to_num(sub)
    return sub

# Returns a euclidean distance from the passed data to all nodes on the SOM grid
def euclidean(self, data):
    return np.sqrt((self.subtract(data)**2).sum(axis=-1))

def fromdistance(self, fn, shape, center=None, dtype=float):
    def distance(*args):
        d = 0
        for i in range(len(shape)):
            d += ((args[i]-center[i])/float(max(1,shape[i]-1)))**2
        return np.sqrt(d)/np.sqrt(len(shape))
    if center == None:
        center = np.array(list(shape))/2
    return fn(np.fromfunction(distance,shape,dtype=dtype))

def Gaussian(self,shape,center,sigma=0.5):
    ''' '''
    def g(x):
        return np.exp(-x**2/sigma**2)
    return self.fromdistance(g,shape,center)

# def learn(self, samples, epochs=10000, sigma=(10, 0.01), lrate=(0.5,0.005)):
def learn(self, X, T, epochs=10000):
    ''' Learn samples '''
#     sigma_i, sigma_f = sigma
#     lrate_a, lrate_b = lrate

    scalingParam = 0.0 # c
    maxDataBMUdist = 0.0
    initialNhRadius = 1 # Half the size of the SOM grid

    for i in xrange(epochs):
        # Adjust learning rate and neighborhood
#         t = i/float(epochs)
#         lrate = lrate_a * math.exp(-(lrate_b * i))
#         sigma = sigma_i*(sigma_f/float(sigma_i))**t

        # Get random sample
        index = np.random.randint(0,X.shape[0])
        data = X[index]
        pred = T[index]

        # Calculate Euclidean distance from data to nodes on the grid
        gridDataDist = self.euclidean(data)

        # Get index of BMU, or nearest node (minimum Euclidean distance)
        BMUindex = np.unravel_index(np.argmin(gridDataDist), gridDataDist.shape)

        # Get distance from new data sample to BMU
        dataBMUdist = gridDataDist[BMUindex]

        # Find max distance from new sample to BMU thus far
        if maxDataBMUdist < dataBMUdist:
            maxDataBMUdist = dataBMUdist

        scalingParam = dataBMUdist / maxDataBMUdist # E(i)
        lrate = scalingParam # lr(i)

        # Get distance from BMU to nodes on the grid

```

```

        gridBMUdist = self.euclidean(self.refVectors[BMUindex])

        #  $nh(d,i) = e^{-(d^2) / (E(i)*c)}$ 
        nh = np.exp( -( gridBMUdist**2) / (scalingParam * initialNhRadius) ) )

        # Generate a Gaussian centered on winner
#         nh = self.Gaussian(gridBMUdist.shape, BMUindex)
#         nh = np.nan_to_num(nh)

        # Move nodes towards sample
        delta = self.subtract(data)
#         for n in xrange(self.refVectors.shape[-1]):
#             #  $m_n(i+1) = m_n(i) + lr(i) * nh(d,i) * (x(i) - m_n(i))$ 
#             self.refVectors[... ,n] -= np.dot(lrate, np.dot(nh, delta[... ,n]))
#             self.predictions -= np.dot(lrate, np.dot(nh, (self.predictions - pred)))
        for n in xrange(self.refVectors.shape[-1]):
            self.refVectors[... ,n] -= lrate * nh * delta[... ,n]
            self.predictions -= lrate * nh * (self.predictions - pred)

def use(self, data):
    predictions = np.zeros((data.shape[0],1))
    for i in xrange(data.shape[0]):
        # Calculate euclidean distance
        dist = self.euclidean(data[i,:])

        # Get index of nearest node (minimum euclidean distance)
        BMUindex = np.unravel_index(np.argmin(dist), dist.shape)

        # Get prediction from BMU
        predictions[i,0] = round(self.predictions[BMUindex])
    return predictions

# # Example: 2d uniform distribution (1d)
# # -----
# print 'One-dimensional SOM over two-dimensional uniform square'
# som = SOM(100,2)
# samples = np.random.random((10000,2))
# learn(som, samples)
#
# # Example: 2d uniform distribution (2d)
# # -----
# print 'Two-dimensional SOM over two-dimensional uniform square'
# som = SOM(10,10,2)
# samples = np.random.random((10000,2))
# learn(som, samples)
# # Example: 2d non-uniform distribution (2d)
# # -----
# print 'Two-dimensional SOM over two-dimensional non-uniform disc'
# som = SOM(10,10,2)
# samples = np.random.normal(loc=.5, scale=.2, size=(10000,2))
# learn(som, samples)
#
# # Example: 2d non-uniform disc distribution (2d)
# # -----
# print 'Two-dimensional SOM over two-dimensional non-uniform ring'
# som = SOM(10,10,2)
# angles = np.random.random(10000)*2*np.pi
# radius = 0.25+np.random.random(10000)*.25
# samples = np.zeros((10000,2))
# samples[:,0] = 0.5+radius*np.cos(angles)
# samples[:,1] = 0.5+radius*np.sin(angles)
# learn(som, samples)

```



## A.27 neuralNet.py (Strategy 2)

```
import numpy as np
import matplotlib.pyplot as plt
import gradientDescent as gd

class NeuralNet:
    """ Neural network with one hidden layer.
    For nonlinear regression (prediction of real-valued outputs)
    net = NeuralNet(ni,nh,no)          # ni is number of attributes each sample,
                                      # nh is number of hidden units,
                                      # no is number of output components
    net.train(X,T,                    # X is nSamples x ni, T is nSamples x no
              nIterations=1000,      # maximum number of SCG iterations
              weightPrecision=1e-8,  # SCG terminates when weight change
magnitudes fall below this,
              errorPrecision=1e-8)   # SCG terminates when objective function
change magnitudes fall below this
    Y,Z = net.use(Xtest,allOutputs=True) # Y is nSamples x no, Z is nSamples x nh
    """
    def __init__(self,ni,nh,no):
        # Initialize weights to uniformly distributed values
        # between small normally-distributed between -0.1 and 0.1
        self.V = np.random.uniform(-0.1,0.1,size=(1+ni,nh))
        self.W = np.random.uniform(-0.1,0.1,size=(1+nh,no))
        self.ni,self.nh,self.no = ni,nh,no
        self.standardize = None
        self.standardizeT = None
        self.unstandardizeT = None

    def train(self,X,T,nSamples,
              nIterations=100,weightPrecision=0.0001,errorPrecision=0.0001):
        if self.standardize is None:
            self.standardize,_ = self.makeStandardize(X)
        X = self.standardize(X)
        X1 = self.addOnes(X)

        if self.standardizeT is None:
            self.standardizeT,self.unstandardizeT = self.makeStandardize(T)
        T = self.standardizeT(T)

        # Local functions used by gradientDescent.scg()
        def pack(V,W):
            return np.hstack((V.flat,W.flat))
        def unpack(w):
            self.V[:] = w[:self.ni+1].reshape( (self.ni+1, self.nh) )
            self.W[:] = w[self.ni+1:].reshape( (self.nh+1, self.no) )
        def objectiveF(w):
            unpack(w)
            Y = np.dot( self.addOnes(np.tanh(np.dot(X1,self.V))), self.W )
            return 0.5 * np.mean((Y - T)**2)
        def gradF(w):
            unpack(w)
            Z = np.tanh(np.dot(X1,self.V))
            Z1 = self.addOnes(Z)
            Y = np.dot( Z1, self.W )
            error = (Y - T) / (nSamples * self.no)
            dV = np.dot( X1.T, np.dot(error,self.W[1:,:].T) * (1-Z**2) )
            dW = np.dot( Z1.T, error )
            return pack(dV,dW)

        scgresult = gd.scg(pack(self.V,self.W), objectiveF, gradF,
```

```

        xPrecision = weightPrecision,
        fPrecision = errorPrecision,
        nIterations = nIterations,
        ftracep=True)

    unpack(scgresult['x'])
    self.reason = scgresult['reason']
    self.errorTrace = scgresult['ftrace']
    self.xTrace = scgresult['xtrace']
    self.numberOfIterations = len(self.errorTrace)

def getNumberOfIterations(self):
    return self.numberOfIterations

def getXTrace(self):
    return self.xTrace

def getErrorTrace(self):
    return self.errorTrace

def use(self,X,allOutputs=False):
    X = self.standardize(X)
    X1 = self.addOnes(X)
    Z = np.tanh(np.dot(X1,self.V))
    Z1 = self.addOnes(Z)
    Y = self.unstandardizeT( np.dot( Z1, self.W ) )
    return (Y,Z) if allOutputs else Y

def plotErrors(self):
    plt.plot(self.errorTrace)
    plt.ylabel("RMSE")
#     plt.xlabel("Iterations")

def addOnes(self,A):
    return np.hstack((np.ones((A.shape[0],1)),A))

def makeStandardize(self,X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)

def draw(self, inputNames = None, outputNames = None, gray = False):
    def isOdd(x):
        return x % 2 != 0

    W = [self.V, self.W]
    nLayers = 2

    # calculate xlim and ylim for whole network plot
    # Assume 4 characters fit between each wire
    # -0.5 is to leave 0.5 spacing before first wire
    xlim = max(map(len,inputNames))/4.0 if inputNames else 1
    ylim = 0

    for li in range(nLayers):
        ni,no = W[li].shape #no means number outputs this layer
        if not isOdd(li):
            ylim += ni + 0.5
        else:

```

```

        xlim += ni + 0.5

ni,no = W[nLayers-1].shape #no means number outputs this layer
if isOdd(nLayers):
    xlim += no + 0.5
else:
    ylim += no + 0.5

# Add space for output names
if outputNames:
    if isOdd(nLayers):
        ylim += 0.25
    else:
        xlim += round(max(map(len,outputNames))/4.0)

ax = plt.gca()

x0 = 1
y0 = 0 # to allow for constant input to first layer
# First Layer
if inputNames:
    #addx = max(map(len,inputNames))*0.1
    y = 0.55
    for n in inputNames:
        y += 1
        ax.text(x0-len(n)*0.2, y, n)
        x0 = max([1,max(map(len,inputNames))/4.0])

for li in range(nLayers):
    Wi = W[li]
    ni,no = Wi.shape
    if not isOdd(li):
        # Odd layer index. Vertical layer. Origin is upper left.
        # Constant input
        ax.text(x0-0.2, y0+0.5, '1')
        for li in range(ni):
            ax.plot((x0,x0+no-0.5), (y0+li+0.5, y0+li+0.5),color='gray')
        # output lines
        for li in range(no):
            ax.plot((x0+1+li-0.5, x0+1+li-0.5), (y0, y0+ni+1),color='gray')
        # cell "bodies"
        xs = x0 + np.arange(no) + 0.5
        ys = np.array([y0+ni+0.5]*no)
        ax.scatter(xs,ys,marker='v',s=1000,c='gray')
        # weights
        if gray:
            colors = np.array(["black","gray"])[(Wi.flat >= 0)+0]
        else:
            colors = np.array(["red","green"])[(Wi.flat >= 0)+0]
        xs = np.arange(no)+ x0+0.5
        ys = np.arange(ni)+ y0 + 0.5
        aWi = abs(Wi)
        aWi = aWi / np.max(aWi) * 50
        coords = np.meshgrid(xs,ys)
        #ax.scatter(coords[0],coords[1],marker='o',s=2*np.pi*aWi**2,c=colors)
        ax.scatter(coords[0],coords[1],marker='s',s=aWi**2,c=colors)
        y0 += ni + 1
        x0 += -1 ## shift for next layer's constant input
    else:
        # Even layer index. Horizontal layer. Origin is upper left.
        # Constant input
        ax.text(x0+0.5, y0-0.2, '1')
        # input lines

```

```

for li in range(ni):
    ax.plot((x0+li+0.5, x0+li+0.5), (y0,y0+no-0.5),color='gray')
# output lines
for li in range(no):
    ax.plot((x0, x0+ni+1), (y0+li+0.5, y0+li+0.5),color='gray')
# cell "bodies"
xs = np.array([x0 + ni + 0.5]*no)
ys = y0 + 0.5 + np.arange(no)
ax.scatter(xs,ys,marker='>',s=1000,c='gray')
# weights
Wiflat = Wi.T.flatten()
if gray:
    colors = np.array(["black","gray"])[(Wiflat >= 0)+0]
else:
    colors = np.array(["red","green"])[(Wiflat >= 0)+0]
xs = np.arange(ni)+x0 + 0.5
ys = np.arange(no)+y0 + 0.5
coords = np.meshgrid(xs,ys)
aWi = abs(Wiflat)
aWi = aWi / np.max(aWi) * 50
#ax.scatter(coords[0],coords[1],marker='o',s=2*np.pi*aWi**2,c=colors)
ax.scatter(coords[0],coords[1],marker='s',s=aWi**2,c=colors)
x0 += ni + 1
y0 -= 1 ##shift to allow for next layer's constant input

# Last layer output labels
if outputNames:
    if isOdd(nLayers):
        x = x0+1.5
        for n in outputNames:
            x += 1
            ax.text(x, y0+0.5, n)
    else:
        y = y0+0.6
        for n in outputNames:
            y += 1
            ax.text(x0+0.2, y, n)
ax.axis([0,xlim, ylim,0])
ax.axis('off')

class NeuralNetClassifier(NeuralNet):

    def g(self,Y):
        fs = np.exp(Y) # N x K-1
        denom = 1 + np.sum(fs,axis=1)
        denom = denom.reshape(-1,1)
        gs = fs / denom
        return gs
#    return np.hstack((gs,1/denom))

    def train(self,X,TI,nSamples,
              nIterations=100,weightPrecision=0.0001,errorPrecision=0.0001):
        if self.standardize is None:
            self.standardize,_ = self.makeStandardize(X)
        X = self.standardize(X)
        X1 = self.addOnes(X)

        # NOTE: TI does not need to be (should not be) standardized

        # Local functions used by gradientDescent.scg()
        def pack(V,W):
            return np.hstack((V.flat,W.flat))
        def unpack(w):

```

```

        self.V[:] = w[(self.ni+1)*self.nh].reshape( (self.ni+1, self.nh) )
        self.W[:] = w[(self.ni+1)*self.nh:].reshape( (self.nh+1, self.no) )
def objectiveF(w):
    unpack(w)
    Y = np.dot( self.addOnes(np.tanh(np.dot(X1,self.V))), self.W )
    gs = self.g(Y)
    ll = -np.sum(TI * np.log(gs)) # log likelihood
#     return 0.5 * np.mean((Y - T)**2)
    return ll
def gradF(w):
    unpack(w)
    Z = np.tanh(np.dot(X1,self.V))
    Z1 = self.addOnes(Z)
    Y = np.dot( Z1, self.W )
    gs = self.g(Y)
    error = (TI - gs) / (nSamples * self.no)
    dV = -np.dot( X1.T, np.dot(error,self.W[1:,:].T) * (1-(Z**2)) )
    dW = -np.dot( Z1.T, error )
#     error = (Y - T) / (nSamples * self.no)
#     dV = np.dot( X1.T, np.dot(error,self.W[1:,:].T) * (1-Z**2) )
#     dW = np.dot( Z1.T, error )
    return pack(dV,dW)

scgresult = gd.scg(pack(self.V,self.W), objectiveF, gradF,
                  xPrecision = weightPrecision,
                  fPrecision = errorPrecision,
                  nIterations = nIterations,
                  ftracep=True)

unpack(scgresult['x'])
self.reason = scgresult['reason']
self.errorTrace = scgresult['ftrace']
self.xTrace = scgresult['xtrace']
self.numberofIterations = len(self.errorTrace)

def use(self,X,allOutputs=False):
    def g(Y):
        fs = np.exp(Y) # N x K-1
        denom = 1 + np.sum(fs,axis=1)
        denom = denom.reshape(-1,1)
        gs = fs / denom
        return np.hstack((gs,1/denom))

    X = self.standardize(X)
    X1 = self.addOnes(X)
    Z = np.tanh(np.dot(X1,self.V))
    Z1 = self.addOnes(Z)
    # Y does not need to be unstandardized
    Y = np.dot( Z1, self.W )
    gs = self.g(Y)
    predictedClasses = np.argmax(gs,axis=1).reshape(-1,1)
    return (predictedClasses,Z) if allOutputs else predictedClasses

def getV(self):
    return self.V

```

## A.28 gradientDescent.py (Strategy 2)

```

### by Chuck Anderson, for CS545
### http://www.cs.colostate.edu/~anderson/cs545
### You may use, but please credit the source.

```

```

#####
### Steepest descent

from copy import copy
import numpy as np
import sys
from math import sqrt, ceil

floatPrecision = sys.float_info.epsilon

def steepest(x, f, gradf, *fargs, **params):
    """steepest:
    Example:
    def parabola(x,xmin,s):
        d = x - xmin
        return np.dot( np.dot(d.T, s), d)
    def parabolaGrad(x,xmin,s):
        d = x - xmin
        return 2 * np.dot(s, d)
    center = np.array([5,5])
    S = np.array([[5,4],[4,5]])
    firstx = np.array([-1.0,2.0])
    r = steepest(firstx, parabola, parabolaGrad, center, S,
        stepsize=0.01,xPrecision=0.001, nIterations=1000)
    print('Optimal: point',r[0],'f',r[1])"""

    stepsize= params.pop("stepsize",0.1)
    evalFunc = params.pop("evalFunc",lambda x: "Eval "+str(x))
    nIterations = params.pop("nIterations",1000)
    xPrecision = params.pop("xPrecision",0.001 * np.mean(x))
    fPrecision = params.pop("fPrecision",0.001 * np.mean(f(x,*fargs)))
    xtracep = params.pop("xtracep",False)
    ftracep = params.pop("ftracep",False)

    xtracep = True
    ftracep = True

    i = 1
    if xtracep:
        xtrace = np.zeros((nIterations+1,len(x)))
        xtrace[0,:] = x
    else:
        xtrace = None
    oldf = f(x,*fargs)
    if ftracep:
        ftrace = np.zeros(nIterations+1)
        ftrace[0] = f(x,*fargs)
    else:
        ftrace = None

    while i <= nIterations:
        g = gradf(x,*fargs)
        newx = x - stepsize * g
        newf = f(newx,*fargs)
        if (i % (nIterations/10)) == 0:
            print "Steepest: Iteration",i,"Error",evalFunc(newf)
        if xtracep:
            xtrace[i,:] = newx
        if ftracep:
            ftrace[i] = newf
        if np.any(newx == np.nan) or newf == np.nan:

```

```

        raise ValueError("Error: Steepest descent produced newx that is NaN.
Stepsize may be too large.")
    if np.any(newx==np.inf) or newf==np.inf:
        raise ValueError("Error: Steepest descent produced newx that is NaN.
Stepsize may be too large.")
    if max(abs(newx - x)) < xPrecision:
        return {'x':newx, 'f':newf, 'nIterations':i, 'xtrace':xtrace[:i,:],
'ftrace':ftrace[:i],
                'reason':"limit on x precision"}
    if abs(newf - oldf) < fPrecision:
        return {'x':newx, 'f':newf, 'nIterations':i, 'xtrace':xtrace[:i,:],
'ftrace':ftrace[:i],
                'reason':"limit on f precision"}
    x = newx
    oldf = newf
    i += 1

    return {'x':newx, 'f':newf, 'nIterations':i, 'xtrace':xtrace[:i,:],
'ftrace':ftrace[:i], 'reason':"did not converge"}

```

```

#####
### Scaled Conjugate Gradient algorithm from
### "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning"
### by Martin F. Moller
### Neural Networks, vol. 6, pp. 525-533, 1993
###
### Adapted by Chuck Anderson from the Matlab implementation by Nabney
### as part of the netlab library.
###
### Call as scg() to see example use.

```

```

def scg(x, f,gradf, *fargs, **params):
    """scg:
    Example:
    def parabola(x,xmin,s):
        d = x - xmin
        return np.dot( np.dot(d.T, s), d)
    def parabolaGrad(x,xmin,s):
        d = x - xmin
        return 2 * np.dot(s, d)
    center = np.array([5,5])
    S = np.array([[5,4],[4,5]])
    firstx = np.array([-1.0,2.0])
    r = scg(firstx, parabola, parabolaGrad, center, S,
            xPrecision=0.001, nIterations=1000)
    print('Optimal: point',r[0],'f',r[1])"""

    print x
    print params
    evalFunc = params.pop("evalFunc",lambda x: "Eval "+str(x))
    nIterations = params.pop("nIterations",1000)
    xPrecision = params.pop("xPrecision",0.001 * np.mean(x))
    fPrecision = params.pop("fPrecision",0.001 * np.mean(f(x,*fargs)))
    xtracep = params.pop("xtracep",False)
    ftracep = params.pop("ftracep",False)

    xtracep = True
    ftracep = True

### from Nabney's netlab matlab library

```

```

nvars = len(x)
sigma0 = 1.0e-4
fold = f(x, *fargs)
fnow = fold
gradnew = gradf(x, *fargs)
gradold = copy(gradnew)
d = -gradnew          # Initial search direction.
success = True        # Force calculation of directional derivs.
nsuccess = 0          # nsuccess counts number of successes.
beta = 1.0            # Initial scale parameter.
betamin = 1.0e-15     # Lower bound on scale.
betamax = 1.0e100     # Upper bound on scale.
j = 1                 # j counts number of iterations.

if xtracep:
    xtrace = np.zeros((nIterations+1,len(x)))
    xtrace[0,:] = x
else:
    xtrace = None
if ftracep:
    ftrace = np.zeros(nIterations+1)
    ftrace[0] = fold
else:
    ftrace = None

### Main optimization loop.
while j <= nIterations:

    # Calculate first and second directional derivatives.
    if success:
        mu = np.dot(d, gradnew)
        if mu==np.nan: print "mu is NaN"
        if mu >= 0:
            d = -gradnew
            mu = np.dot(d, gradnew)
            kappa = np.dot(d, d)
            if kappa < floatPrecision:
                return {'x':x, 'f':fnow, 'nIterations':j, 'xtrace':xtrace[:j,:],
'ftrace':ftrace[:j]},
                    'reason':"limit on machine precision"}
            sigma = sigma0/sqrt(kappa)
            xplus = x + sigma * d
            gplus = gradf(xplus, *fargs)
            theta = np.dot(d, gplus - gradnew)/sigma

        ## Increase effective curvature and evaluate step size alpha.
        delta = theta + beta * kappa
        if delta is np.nan: print "delta is NaN"
        if delta <= 0:
            delta = beta * kappa
            beta = beta - theta/kappa
        alpha = -mu/delta

        ## Calculate the comparison ratio.
        xnew = x + alpha * d
        fnew = f(xnew, *fargs)
        Delta = 2 * (fnew - fold) / (alpha*mu)
        if Delta is not np.nan and Delta >= 0:
            success = True
            nsuccess += 1
            x = xnew
            fnow = fnew
        else:

```



```

        success = False
        fnow = fold
    if xtracep:
        xtrace[j,:] = x
    if ftracep:
        ftrace[j] = fnew

    if j % ceil(nIterations/10) == 0:
        print "SCG: Iteration",j,"fValue",evalFunc(fnow),"Scale",beta

    if success:
        ## Test for termination

        ##print(c(max(abs(alpha*d)),max(abs(fnew-fold))))

        if max(abs(alpha*d)) < xPrecision:
            return {'x':x, 'f':fnow, 'nIterations':j, 'xtrace':xtrace[:j,:],
'ftrace':ftrace[:j],
                    'reason':"limit on x Precision"}
        elif abs(fnew-fold) < fPrecision:
            return {'x':x, 'f':fnow, 'nIterations':j, 'xtrace':xtrace[:j,:],
'ftrace':ftrace[:j],
                    'reason':"limit on f Precision"}
        else:
            ## Update variables for new position
            fold = fnew
            gradold = gradnew
            gradnew = gradf(x, *fargs)
            #print "gradold",gradold
            #print "gradnew",gradnew
            ## If the gradient is zero then we are done.
            if np.dot(gradnew, gradnew) == 0:
                return {'x':x, 'f':fnow, 'nIterations':j, 'xtrace':xtrace[:j,:],
'ftrace':ftrace[:j],
                        'reason':"zero gradient"}

            ## Adjust beta according to comparison ratio.
            if Delta is np.nan or Delta < 0.25:
                beta = min(4.0*beta, betamax)
            elif Delta > 0.75:
                beta = max(0.5*beta, betamin)

            ## Update search direction using Polak-Ribiere formula, or re-start
            ## in direction of negative gradient after nparams steps.
            if nsuccess == nvars:
                d = -gradnew
                nsuccess = 0
            elif success:
                gamma = np.dot(gradold - gradnew, gradnew/mu)
                #print "gamma",gamma
                d = gamma * d - gradnew
                #print "end d",d
            j += 1

            ## If we get here, then we haven't terminated in the given number of
            ## iterations.

            ##print("Did not converge.")
            return {'x':x, 'f':fnow, 'nIterations':j, 'xtrace':xtrace[:j,:],
'ftrace':ftrace[:j],
                    'reason':"did not converge"}

```

## ABBREVIATIONS

3D	3-Dimensional
3G/4G	3 <sup>rd</sup> and 4 <sup>th</sup> Generation (of cellular mobile networks)
AP	Access Point
API	Application Programming Interface
ARM	Advanced RISC Machine
AURA	Application and User interaction Aware (framework)
CHBL	Change Blindness
CPU	Central Processing Unit
D(V)FS	Dynamic (Voltage) Frequency Scaling
EDGE	Enhanced Data rates for GSM Evolution
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSM	Global System for Mobile communications
HBLP	High Burst Long Pause
HTC	High Tech Computer corporation
KNN	K-Nearest Neighbor

LBA	Location-Based Application
LCD	Liquid Crystal Display
LDA	Linear Discriminant Analysis
LLR	Linear Logistic Regression
MDP	Markov Decision Process
MEMS	MicroElectroMechanical System
MHz/GHz	MegaHertz/GigaHertz
MVSOM	Missing Values Self-Organizing Map
mW	milliWatts
NN	Neural Network
OLED	Organic Light-Emitting Diode
OS	Operating System
PC	Personal Computer
PCA	Principle Component Analysis
PCB	Printed Circuit Board
PCM	Phase Change Memory
PDA	Personal Digital Assistant

QoS	Quality of Service
RAM	Random Access Memory
RIM	Research In Motion
RISC	Reduced Instruction Set
RSSI	Received Signal Strength Indicator
SCG	Scaled Conjugate Gradient
SDK	Software Development Kit
SMS	Short Message Service
SVM	Support Vector Machine
UI	User Interface
UMTS	Universal Mobile Telecommunications System
VM	Virtual Machine
VRL	Variable Rate Logging
WiFi	Wireless Fidelity