

Imperial College London  
Department of Computing

# **Mobile phone data collection and analysis with Open Battery**

by

Alexandros Bentevis

Submitted in partial fulfilment of the requirements for the MSc Degree in  
Advanced Computing of Imperial College London

September 2013



# Abstract

Recently, the battery dissipation of portable smartphone and tablet devices has become a topic of major concern. These devices provide power hungry features that become even more demanding in resources as they improve over time. Many studies focus on battery lifetime prediction, but most of the models proposed are not capable of working in lightweight portable devices.

In this work, we present a battery lifetime prediction model that uses battery related as well as context related information in order to make predictions. The core of the model is an instance-based machine learning engine which uses the distance-weighted  $k$ -NN algorithm to make accurate estimations of battery lifetime based on usage patterns and characteristics. The model does not use statically defined power profiles, but adapts dynamically to the usage characteristics of the user. We also improved an Android-based application called Open Battery, which monitors battery related data and makes them available on-line for academic purposes. The new version of Open Battery is designed to use the prediction model we developed and provide valuable information regarding battery consumption. The last part of this project consists of the statistical analysis of the data collected with Open Battery and the demonstration of our results.



# Acknowledgements

Firstly, I would like to express my gratitude to my family for their continued support throughout the completion of my studies, without whom none of this would have been possible. I would also like to thank Kelly for always believing in me and being by my side and my friends who have encouraged me throughout this year at Imperial College London.

I would also like to thank State Scholarship Foundation for sponsoring my studies this year. The completion of this project was made in the framework of the implementation of the postgraduate program (MSc) that has been co-financed through the Action "State Scholarship Foundation's Grants Programme" following a procedure of individualized evaluation for the academic year 2012-2013", from resources of the operational program "Education and Lifelong Learning" of the European Social Fund and the National Strategic Reference Framework 2007-2013.

I would also like to thank Gareth Jones for his necessary guidance and enthusiasm throughout the development of this project and Tiberiu Chis for providing me many interesting ideas for my academic research.

Finally, I would like to thank my supervisor, Professor Peter Harrison for his constant support, invaluable guidance and fast responses to my emails, in addition to his professional duties at Imperial College London.



Dedicated to my brother Vassilis



# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	Motivation . . . . .	15
1.2	Objectives . . . . .	16
1.3	Thesis Outline . . . . .	16
<b>2</b>	<b>Technical Background</b>	<b>17</b>
2.1	Battery basics . . . . .	17
2.1.1	Electrochemical cells . . . . .	17
2.1.2	Lithium-ion battery . . . . .	19
2.2	Android Operating System . . . . .	20
2.2.1	Android system architecture . . . . .	20
2.2.2	Android power management . . . . .	21
2.2.3	Android application fundamentals . . . . .	22
2.3	Machine learning . . . . .	25
2.3.1	Instance-based learning . . . . .	25
2.3.2	The $k$ -nearest neighbour algorithm . . . . .	25
2.3.3	The distance-weighted $k$ -nearest neighbour algorithm . . . . .	27
2.4	Summary . . . . .	28
<b>3</b>	<b>Related Work on Battery Power Modelling</b>	<b>29</b>
3.1	Battery Analytical Models . . . . .	29
3.1.1	Peukert's law Model . . . . .	29
3.1.2	Kinetic Battery Model (KiBaM) . . . . .	30
3.1.3	Rakhmatov and Vrudhula's Diffusion Model . . . . .	32
3.2	Battery Stochastic Models . . . . .	32
3.2.1	Chiasseroni and Rao . . . . .	33
3.2.2	Fluid Queue Battery Model . . . . .	34
3.2.3	Others . . . . .	35
3.3	Power Model Generation . . . . .	35
3.3.1	Application-level Prediction of Battery Dissipation . . . . .	36
3.3.2	PowerBooster and PowerTutor . . . . .	37
3.3.3	Others . . . . .	38
3.4	Context-aware power management . . . . .	39
3.4.1	Context-aware Battery Management for Mobile Phones (CABMAN) . . . . .	39
3.5	Analysis of battery usage patterns in smartphones . . . . .	40
3.5.1	Kang, Seo and Hong's Personalized Battery Lifetime Prediction . . . . .	41
3.5.2	Others . . . . .	42
3.6	Summary . . . . .	42
<b>4</b>	<b>Open Battery: a battery data analysis software tool</b>	<b>45</b>
4.1	Battery lifetime prediction model based on usage patterns . . . . .	45
4.1.1	Exploiting usage characteristics for battery lifetime prediction . . . . .	45
4.1.2	Model overview . . . . .	46

4.1.3	Preliminaries . . . . .	46
4.1.4	Training process . . . . .	48
4.1.5	Prediction algorithm . . . . .	50
4.2	Software Architecture of Open Battery . . . . .	51
4.2.1	The original version of Open Battery . . . . .	51
4.2.2	Goals and requirements . . . . .	52
4.2.3	Open Battery architecture and design . . . . .	52
4.3	Implementation Details . . . . .	55
4.3.1	Basic Android application components of Open Battery . . . . .	55
4.3.2	Prediction Model . . . . .	59
4.3.3	Persistent data management . . . . .	59
4.3.4	Software extensibility . . . . .	61
4.4	Summary . . . . .	63
<b>5</b>	<b>Evaluation and Discussion</b>	<b>65</b>
5.1	Mobile data analysis . . . . .	65
5.1.1	Dataset details . . . . .	65
5.1.2	Analysis results . . . . .	65
5.2	Prediction model Evaluation . . . . .	71
5.2.1	Accuracy . . . . .	71
5.2.2	Performance . . . . .	73
5.3	Software Evaluation . . . . .	75
5.4	Limitations and areas of improvement . . . . .	76
5.5	Summary . . . . .	77
<b>6</b>	<b>Conclusions and Future Work</b>	<b>79</b>
6.1	Conclusions . . . . .	79
6.2	Further Work on Open Battery project . . . . .	80
<b>Appendix A</b>	<b>Technical details of the original Open Battery Android application</b>	<b>87</b>
A.1	Graphical User Interface . . . . .	87
A.2	UML class diagram . . . . .	88
<b>Appendix B</b>	<b>Open Battery User Guide</b>	<b>89</b>
B.1	Installation . . . . .	89
B.2	Open Battery Screens . . . . .	89
B.3	Widget . . . . .	93
<b>Appendix C</b>	<b>Software diagnostics of the new Open Battery application</b>	<b>94</b>
C.1	Software engineering tools . . . . .	94
C.2	Software metrics . . . . .	94
C.3	List of packages and classes . . . . .	95
C.4	Dependency graph . . . . .	96

# List of Figures

2.1	Schematic representation of an electrochemical cell [23] . . . . .	17
2.2	Characteristics of an ideal battery: Constant voltage and constant capacity. [33] . . . . .	18
2.3	Non-Ideal battery properties [28] . . . . .	19
2.4	Schematic representation of a lithium-ion battery [43] . . . . .	19
2.5	Distribution of Android versions [16] . . . . .	21
2.6	Android layered architecture diagram [17] . . . . .	22
2.7	Android power management architecture . . . . .	23
2.8	Differences between the Manhattan distance and the Euclidean distance . . . . .	26
2.9	The hamming distance between two binary vectors . . . . .	27
3.1	The two-tank model of KiBaM [23] . . . . .	30
3.2	Schematic representation of the voltage model [30] . . . . .	31
3.3	The basic Markov chain battery model by Chiasseroni and Rao [7] . . . . .	33
3.4	The extended Markov chain battery model by Chiasseroni and Rao [8] . . . . .	34
3.5	Sample trace from a fluid queue [20] . . . . .	35
3.6	Comparison of battery drain rates for all benchmarks [27] . . . . .	36
3.7	Wi-Fi interface power states according to PowerBooster [45] . . . . .	37
3.8	Power model construction [45] . . . . .	37
3.9	CABMAN system architecture [41] . . . . .	39
3.10	Architecture of the personalized prediction system of Kang, Seo and Hong [25] . . . . .	41
4.1	Prediction model using a learning engine based on usage patterns . . . . .	46
4.2	An example of a device state in Open Battery prediction model . . . . .	47
4.3	The construction of the state consumption table . . . . .	49
4.4	The construction of the pattern table with parameters $pSize=3$ and $fSize=2$ . . . . .	50
4.5	The Three-Tier software architecture of Open Battery . . . . .	53
4.6	The Model-View-Controller software paradigm . . . . .	53
4.7	The design of Open Battery Android application . . . . .	54
4.8	The activity sequence of Open Battery . . . . .	56
4.9	The widget updating process. . . . .	58
4.10	The SQLite local database . . . . .	60
4.11	The PostgreSQL remote database . . . . .	61
4.12	The DAO implementation of two different types of storage . . . . .	61
5.1	Our dataset regarding battery consumption . . . . .	66
5.2	Proportion of time spent charging/discharging . . . . .	67
5.3	How the device is being charged . . . . .	68
5.4	Battery level at start of charging periods . . . . .	68
5.5	Battery level during the day at start of charging periods . . . . .	69
5.6	Frequencies of device states during charging and discharging periods . . . . .	69
5.7	A simple transition diagram with merged device states . . . . .	70
5.8	Spent time in each merged operational state . . . . .	71
5.9	Error vs parameter $k$ for $k$ -NN and distance-weighted $k$ -NN algorithms . . . . .	72
5.10	Error vs Size of dataset . . . . .	73

5.11	The error for different values of $pSize$ and $fSize$ . . . . .	74
5.12	Time complexity of the instance-based learning engine . . . . .	74
5.13	Battery consumption vs Logging rate . . . . .	75
5.14	CPU usage vs Logging rate . . . . .	76
5.15	Memory heap vs Dataset size . . . . .	77
A.1	The GUI of the original Open Battery application . . . . .	87
A.2	UML diagram of the original Open Battery Android application . . . . .	88
B.1	The Splash Screen of Open Battery . . . . .	89
B.2	The Home Screen of Open Battery . . . . .	90
B.3	The Application Process Screen of Open Battery . . . . .	90
B.4	The History Screen of Open Battery . . . . .	91
B.5	The Statistics Screen of Open Battery . . . . .	92
B.6	The Preferences Screen of Open Battery . . . . .	93
B.7	The Open Battery widget . . . . .	93
C.1	The packages of Open Battery tool . . . . .	95
C.2	The package dependency graph . . . . .	96

# List of Tables

2.1	Usage share of the different Android platform versions on June 3, 2013 [16] . . . . .	21
2.2	Different types of wake locks in Android OS . . . . .	22
3.1	Possible states of a mobile device according to Kang, Seo and Hong . . . . .	41
4.1	Different subsystems and their states according to Open Battery prediction model .	47
5.1	The merged operational states of a portable device . . . . .	70
5.2	Parameters used for our experiments . . . . .	72
C.1	Software metrics of Open Battery tool . . . . .	94



# Chapter 1

## Introduction

Smartphones are one of the fastest growing technological types of devices in the current mobile networks. Google has recently announced in Google I/O 2013 [18] that there have been over 900 million Android device activations to date. If we compare that to 100 million in 2011, and 400 million in 2012, it is obvious that the circulation of Android devices has increased immensely.

Their rapidly expanding appeal on a vaster consuming target group has steadily created the need on personalized services, as well as customizing the resource management of smartphones, according to individual customers requirements. Android smartphones and tablet devices offer several power consuming hardware components (Wi-Fi, 3G, GPS etc) and the application developers are exploiting them in order to provide revolutionary and high level user experience. However, the battery life has not increased at the same rate to support the power demand [11].

This fact implicates the necessity of usage pattern analysis, using real log data from real smartphones users. The studies realized towards this goal are basically aiming on achieving the optimization of battery usage, the best possible accuracy of predictions of battery lifetime and the detection of abnormal processes and intrusions, through user identification data [26].

In order to enhance the understanding of how and where the energy is used and distributed throughout the several utilities and applications of a smartphone, several studies have been realized, analysing the motif of power consumption performed by certain devices under multiple usage scenarios, which differ according to the individual's needs and preferences.

The implications on battery life and energy usage caused by specific users' behaviour, such as battery charging habits and customizing aiming to diminish energy consumption when needed (i.e. alteration of the default brightness level), established a gradually emerging searching field, known as Human-Battery Interaction (HBI), which has provided astonishing findings referring to real usage patterns [38].

In this project we plan to improve an existing application for Android platform called Open Battery that logs battery data and make it available on-line for academic research purposes and create a model to predict future battery performance. The model uses a machine learning engine that recognizes usage patterns and makes estimations based on user-specific characteristics. The data collected are published on-line under the Open Data Commons Public Domain Dedication and License (PDDL) [12], which allows free access for purposes of redistribution and re-evaluation.

### 1.1 Motivation

**Improvement of models regarding battery usage** Several models have been proposed in previous studies, processing several monitored data and proposing techniques to predict battery consumption. Although, the feasibility of these systems has been proved in certain occasions, the

need of acquiring the highest possible accuracy and performance of prediction algorithms should not be considered a trivial issue.

**The need of free battery data for academic purposes** One of the basic obstacles that have occasionally shifted the progress of previous studies is the fact that the collection of user data lies under the protection of privacy agreements. As a result, the data is restricted and forbidden to be shared outside the named researchers original proposal. Thus, because of the inevitability that different devices and different operating scenarios will display different behaviour, the fact that an innovating, more efficient way of collecting real users data, easily accessed for further academic or commercial usage is been strongly emphasized.

## 1.2 Objectives

**Research on models for predicting battery consumption.** In literature there are many approaches regarding battery modelling and prediction of battery consumption. These models will be studied and evaluated in order to develop a technique for prediction of battery drain in a commercial application like Open Battery.

**Rewrite the Open Battery application introducing new features.** The Open Battery is a project that has to be rewritten in order to improve its performance and its usability. The Graphical User Interface (GUI) of the Android application will be redesigned, new methods of data representation, analysis and visualization will be introduced and a cache model will be designed in order to improve the overall behaviour of the system. Moreover, a prediction model will be implemented, in order to make an estimation on when the battery will be empty or fully charged.

**Demonstrate the application and results regarding battery consumption of Android smartphone devices.** The improved Open Battery application will be used in Android devices in order to demonstrate its functionality. We will also use the battery data to visualize the energy consumption of the handsets and give conclusions according to the data retrieved regarding smartphone usage patterns.

## 1.3 Thesis Outline

We start by defining in a greater depth what a Lithium-ion (Li-ion) battery used in smartphone devices is and by presenting some related technical concepts and details of Android Operating System (OS) and machine learning methods in Chapter 2. This is important in order to better understand the context of our work and to present the theoretical background that will be useful for the rest of this project. It will also help in better understanding concepts presented in the next chapter. In Chapter 3 we present the related work of other researchers in this field, discussing in more depth the advantages and disadvantages of existing battery performance prediction models and systems. Some of the ideas presented in this section are used to form the basis of this project. In Chapter 4 we present the formal description of the battery lifetime prediction model we developed and we present the design of the Open Battery system describing its software components and the interaction between them. We also discuss the implementation details of each of these components and any technical issues that arise through the development of Open Battery. In Chapter 5 we present the evaluation of our work pointing out the limitations of our approach, providing qualitative and quantitative analysis regarding the accuracy and performance of our prediction model after running a number of experiments and presenting some valuable statistics from the analysis of our data. Finally, in Chapter 6 we conclude this work and consider directions for further work in this project.

# Chapter 2

## Technical Background

This chapter presents the basics regarding battery lifetime and especially of Lithium-ion batteries which are used in smartphone and tablet devices. It also presents an overview of Android OS and application development basics as well as background knowledge of the  $k$ -nearest neighbour machine learning algorithm that is used in our prediction model which will be discussed in more depth in Chapter 4.

### 2.1 Battery basics

#### 2.1.1 Electrochemical cells

A battery consists of one or more electrochemical cells connected in series or in parallel. In these cells, a electrochemical reaction converts stored chemical energy into electrical energy. A cell is composed of an anode holding charged ions, a cathode holding discharged ions and the electrolyte separating the two electrodes. An electrochemical cell is illustrated in Figure 2.1

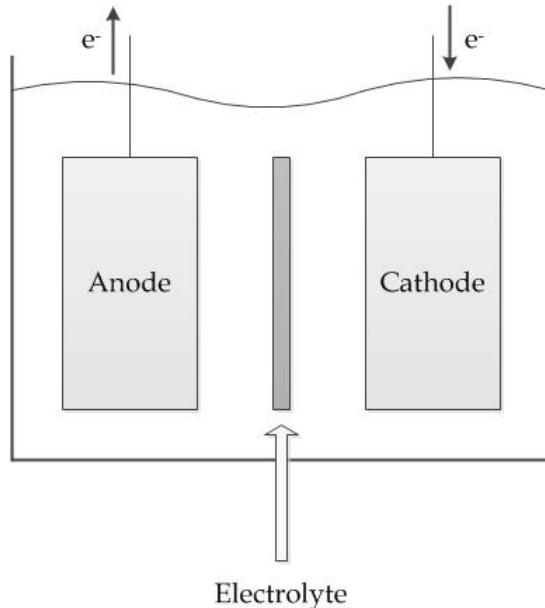
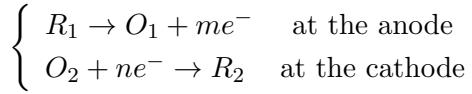


Figure 2.1: Schematic representation of an electrochemical cell [23]

During the discharge of the battery two reactions take place, one at the anode and one at the cathode of the electrochemical cell. More specifically, at the anode an oxidation reaction takes place, where a reductant ( $R_1$ ) donates  $m$  electrons which are released into the connected circuit. On the other hand, on the cathode a reduction reaction occurs, where  $m$  electrons are accepted by an oxidant ( $O_2$ ) [23]. The reactions during battery discharge are:



### Ideal battery properties

Among the many properties that a battery has, the two most important regarding battery usage are voltage and capacity. The voltage is expressed in volts ( $V$ ), while the charge capacity of the battery is typically given in terms of Amp-hours ( $Ah$ ) or milliAmp-hours( $mAh$ ) and is called the battery's  $C$  rating. Under ideal circumstances, a battery has a constant voltage throughout a discharge which drops instantaneously to zero when the battery becomes empty and has constant capacity as shown in Figure 2.2.

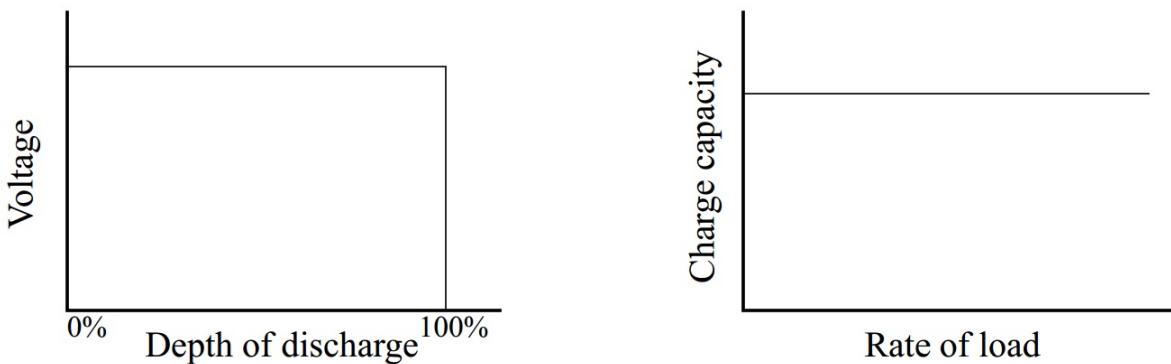


Figure 2.2: Characteristics of an ideal battery: Constant voltage and constant capacity. [33]

For constant loads, the calculation of the battery's lifetime is easy, given by the equation:

$$L = \frac{C}{I} \quad (2.1)$$

where  $L$  is the lifetime of the battery,  $C$  is its capacity and  $I$  is the discharge current. However, due to some properties that are present in real circumstances this equation does not hold for real batteries. These properties are discussed next.

### Non-Ideal battery properties

While ideally the voltage and the capacity of the battery remain constant, in real conditions both vary widely. The two effects that make battery performance sensitive to discharge profile are the *Rate Capacity Effect* and the *Recovery Effect*. It is very important to mention that these effects occur for all types of batteries but the magnitude of their impact on the battery's lifetime depends on each battery type, making the formal modelling of batteries a very complicated process.

**Rate Capacity Effect** The Rate Capacity Effect is the phenomenon where the voltage slowly drops while the battery is discharging and the capacity is lower for high discharge currents. This behaviour is illustrated in Figure 2.3a which shows the capacity as a function of the discharge rate which is given in terms of  $C$  rating. In this figure we can see the loss of effective capacity with increasing high discharge rates for a typical NiCd battery.

**Recovery effect** During periods of time when little or no discharge occur the battery actually recovers the capacity that it lost in periods of high discharge. The degree of the battery recovery depends on the discharge profile, the length and distribution of the idle periods and the details regarding the battery construction [40]. This effect of non-linearity is shown in Figure 2.3b.

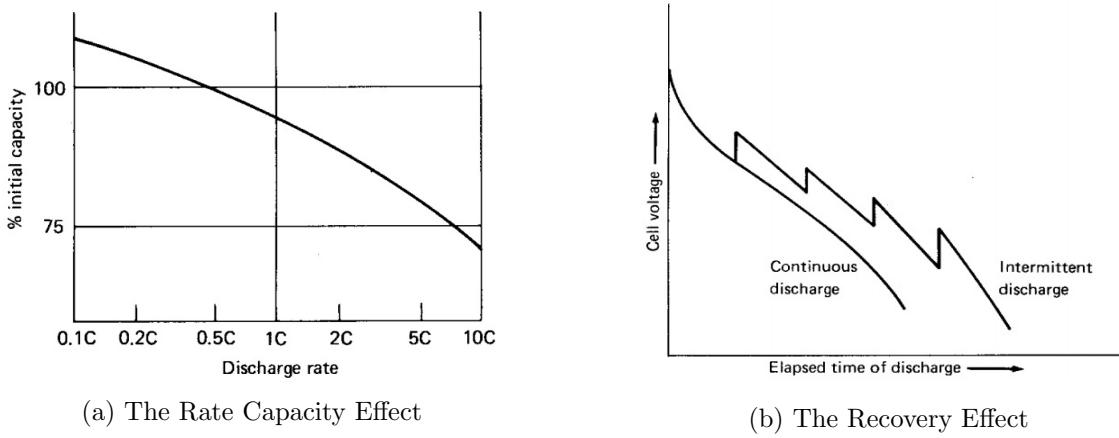
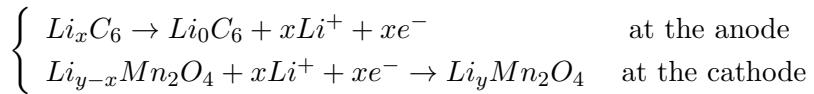


Figure 2.3: Non-Ideal battery properties [28]

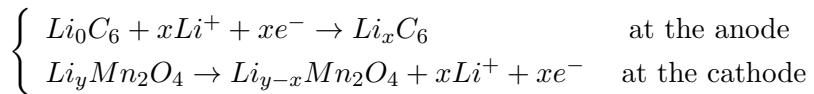
### 2.1.2 Lithium-ion battery

Lithium-ion batteries are rechargeable batteries and they are considered as the fastest growing battery systems. They are widely used in high-performance devices such as laptop computers, digital cameras, mobile phones and lately in smartphone and tablet devices due to their high energy density, light weight, long service lifetimes and low-discharge currents. The schematic representation of a lithium-ion battery is shown in Figure 2.4. As we can see from the figure it consists of the positive electrode  $Li_xC_6$ , the negative electrode  $Li_yMn_2O_4$  and a plasticized electrolyte [43]. A variety of substances are used in lithium batteries, but a common combination is a lithium cobalt oxide cathode and a carbon anode.

When discharging, the electrochemical reactions that take place are:



On the other hand during charging, the electrochemical reactions that take place are:



As we can observe from these formulas, lithium ion de-inserts from solid particles of the anode and inserts into solid particles of the cathode during discharging . This process is reversed during the charge process.

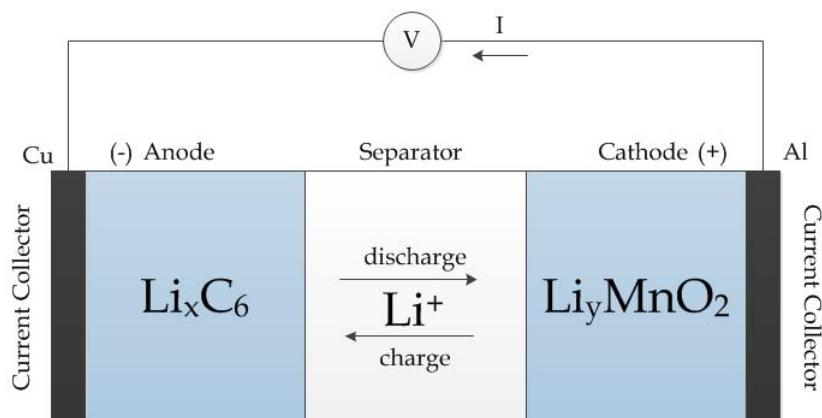


Figure 2.4: Schematic representation of a lithium-ion battery [43]

## Rechargeable battery properties

Lithium-ion battery is a rechargeable type of battery which has properties that are not applicable for the basic battery model discussed earlier in this chapter. During the charging process, a rechargeable battery may regain its capacity lost during discharging. Generally, a discharge process and the charge process following it are called a *cycle*. The capacity of a lithium-ion battery tends to decrease when the number of cycles increases resulting in a phenomenon called *Cycle Ageing Phenomena*. Moreover, the battery's temperature seems to have a major influence on the impact of cycle aging. Unfortunately, high battery temperature results in low life cycle of the battery making an accurate prediction of battery lifetime even more complicated than the prediction in non-rechargeable batteries [43]. According to a wide research on a total of 85 commercial lithium-ion batteries [19] their capacities tend to decrease by 10-40% during the first 450 cycles. Therefore, the lack of knowledge regarding the temperature and the cycle life of the battery makes lifetime predictions inaccurate in real lithium-ion batteries.

## 2.2 Android Operating System

The Lithium-ion batteries that were described in the previous section are used nowadays in high performance portable devices such as smartphones and tablet computers. These devices have several power hungry features, such as colorful Organic Light-Emitting Diode (OLED) screens, Wi-Fi and cellular connectivity and GPS. Taking consideration the major popularity of these portable devices, their battery consumption is a major issue and many researchers have tried to investigate how to predict it and minimize it.

Android is an open-source Linux-based operating system originally designed for mobile devices, such as smartphone and tablet computers, originally developed by Android Inc. In August 2005, Google acquired Android Inc. and the first Android based smartphone was released in August 2008 running Android version 1.0. Android-based devices gained increasing popularity during the last five years dominating nearly 70% of smartphone market, according to [29]. During the same period more than 10 new versions of the system have been released, as shown in Table 2.1. The version distribution of active Android devices is illustrated in Figure 2.5 showing that devices running Android OS versions prior to 2.3 tend to disappear nowadays. Its high usage around the globe as well as its well-developed Software Development Kit (SDK) and the adaptability of the system to different portable devices made the decision of which platform will Open Battery be developed an easy choice.

In this section we discuss some fundamental technical details of the architecture of Android OS as well as application development in this platform that are essential for understanding topics regarding Open Battery implementation covered in Chapter 4.

### 2.2.1 Android system architecture

The Android OS can be considered as a software stack of layers, where each layer is a group of different components, as shown in Figure 2.6. This stack consists of the Linux kernel, basic libraries, an application framework and built-in applications.

The basic layer of the stack is the *Linux kernel*. Android OS is built on top of the Linux kernel version 3.x (version 2.6 prior to Android 4.0) with some further architectural changes made by Google outside the typical Linux kernel development cycle. Although it is built on top of the Linux kernel, Android is not identical to Linux, since it does not include the full set of standard Linux utilities. The kernel is responsible for interacting with the hardware of the device and contains all the essential hardware drivers.

The *Libraries* layer is built on top of the Linux kernel layer and consists of many essential na-

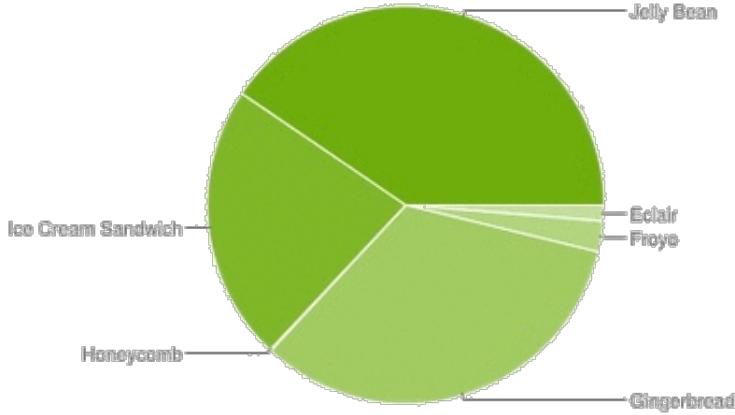


Figure 2.5: Distribution of Android versions [16]

Version	Codename	Release date	API level	Distribution
4.2.x	Jelly Bean	November 13, 2012	17	4.0%
4.1.x	Jelly Bean	July 9, 2012	16	29.0%
4.0.x	Ice Cream Sandwich	December 16, 2011	15	25.6%
3.2	Honeycomb	July 15, 2011	13	0.1%
3.1	Honeycomb	May 10, 2011	12	0%
2.3.3 - 2.3.7	Gingerbread	February 9, 2011	10	36.4%
2.3 - 2.3.2	Gingerbread	December 6, 2010	9	0.1%
2.2	Froyo	May 20, 2010	8	3.2%
2.0 - 2.1	Eclair	October 26, 2009	7	1.5%
1.6	Donut	September 15, 2009	4	0.1%
1.5	Cupcake	April 30, 2009	3	0%
1.1	-	February 9, 2009	2	0%
1.0	-	September 23, 2008	1	0%

Table 2.1: Usage share of the different Android platform versions on June 3, 2013 [16]

tive libraries written in C or C++. An important native library related to this project is SQLite, which is a lightweight transactional database engine used in Android OS as a back end for data storage purposes.

Android layered architecture provides a powerful *Application Framework* for application development. Android applications are written in the Java programming language, while the GUI components are preferably declared in lightweight sets of Extensible Markup Language (XML) resource files. The Android SDK tools are responsible for compiling the code into an Android package, which is a file with an .apk extension. The code in an .apk file is considered to be one single application and Android-based devices use this file in order to install the current application in the internal memory of the device.

### 2.2.2 Android power management

One of the changes that Google made to the original Linux kernel was the addition of Android's power driver to the kernel 2.6.33. The new driver was added considering that the portable devices

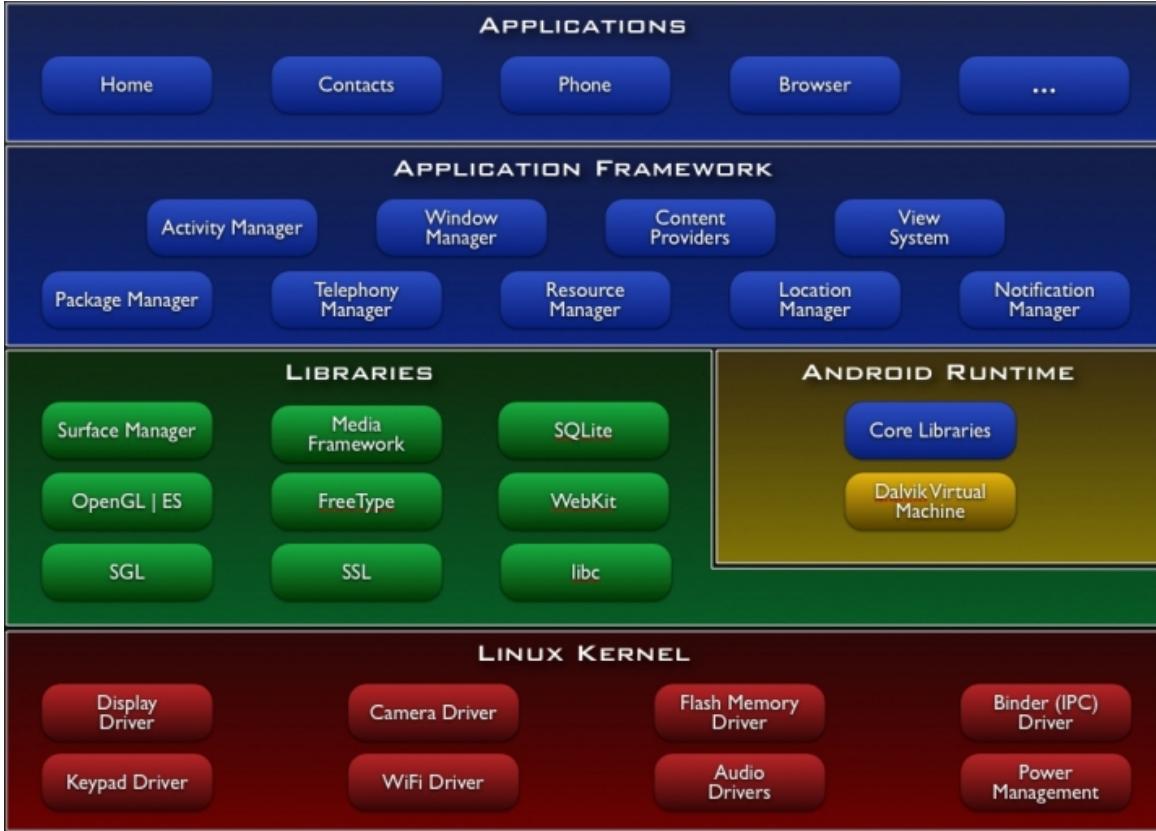


Figure 2.6: Android layered architecture diagram [17]

running Android OS have limited battery capacity and the provided power saving features are different than the ones of personal computers.

In the Application Framework layer a Power Management API is developed as shown in Figure 2.7. Android applications request CPU resources using mechanisms called wake locks through the Application Framework, the Libraries and Linux kernel layers. Wake locks are Power Manager features used to control the power state of the host device. A wake lock that is considered in locked state prevents the system from accessing suspend or low-power states, according to its type. As illustrated in Figure 2.7, when application App1 needs CPU resources, it sends a request to the Power Manager API which transfers the request to the power driver located in the Linux kernel layer. When the wake lock is created the Power Manager reports the incident back to App1. According to the type of the wake lock the appropriate resources are consumed. In Table 2.2 different wake lock types are shown.

Wake lock type	CPU	Display	Keyboard
PARTIAL_WAKE_LOCK	ON	OFF	OFF
SCREEN_DIM_WAKE_LOCK	ON	DIM	OFF
SCREEN_BRIGHT_WAKE_LOCK	ON	BRIGHT	OFF
FULL_WAKE_LOCK	ON	BRIGHT	BRIGHT

Table 2.2: Different types of wake locks in Android OS

### 2.2.3 Android application fundamentals

In this section we introduce some basic details of Android application development. The technical details presented here are essential for understanding specific implementation topics of Open

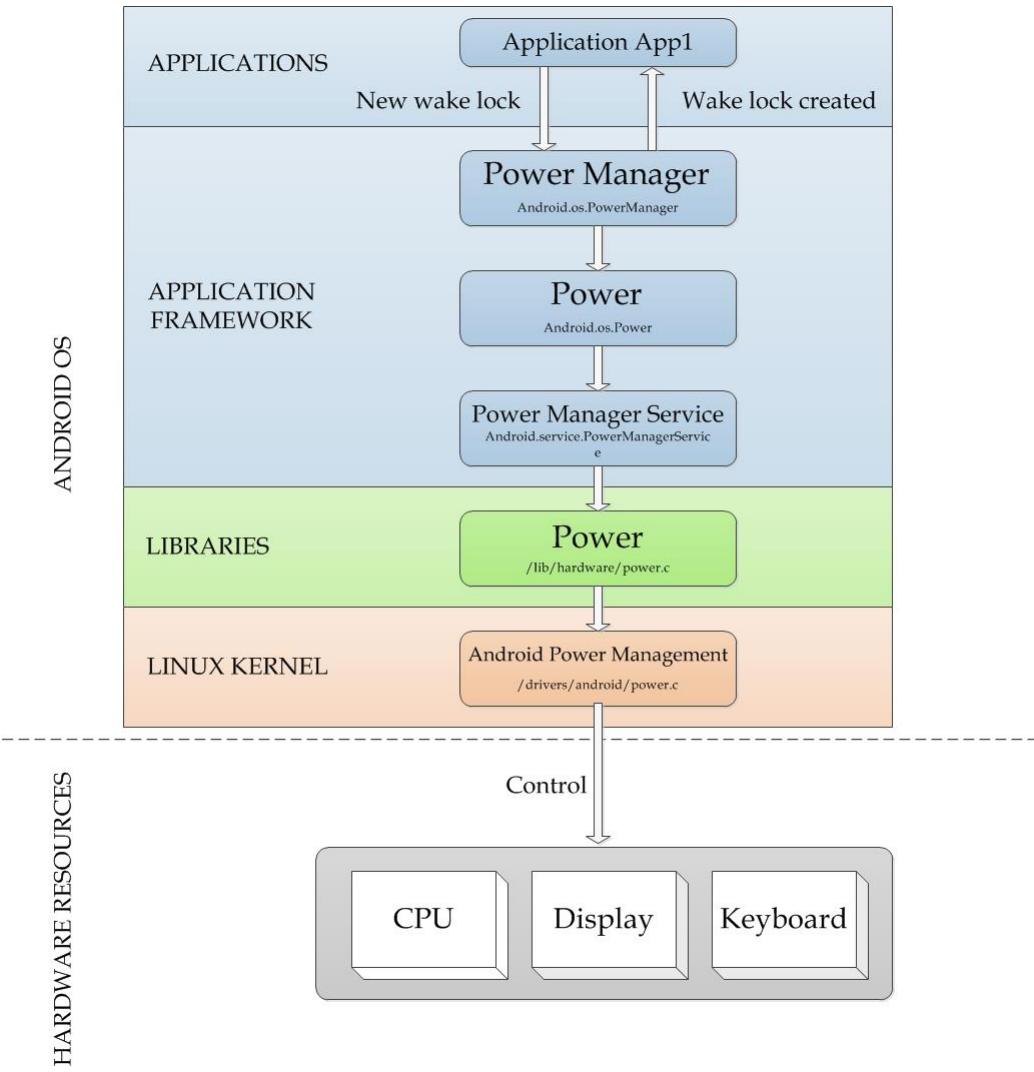


Figure 2.7: Android power management architecture

Battery that are presented in Chapter 5.

### Android application components

From the developer's perspective, the foundation elements of Android OS are the application components. They represent different entry points of the Android application, since applications in Android do not have a single entry point (ie. there is no `main()` method), unlike applications on most other systems. There are five different types of application components, each of whom plays a distinct role and has a specific lifecycle that defines how the component is created and destroyed.

- **Activities** An activity is a component that represents a single screen of the Android application, since it offers the means to render the user interface of each screen. Although the activities are working together in order to present the screen sequence of the application, each activity is considered to be an independent entity in the Android system.
- **Widgets** A widget is a view of the application's most important data and functionality that is accessible from the home screen of the device. Widgets can be moved across the home screen panels and can be resized according to the users' preferences.
- **Services** A service is a non-graphical component that represents an operation running in the background of the application. Services run in the main thread of the application just like the other components and communication between them can be achieved through an interface that the service exposes.

- **Content providers** A content provider is responsible for managing and exposing shared data to applications. The data can be stored in the local file system, an SQLite database, on the web, or any other storage location the application can have write access. Other applications can read or even modify the data, through the mechanisms of the content provider.
- **Broadcast receivers** A broadcast receiver is a component that receives broadcast events of the Android OS and responds appropriately. Broadcast receivers do not display any GUI but they are able to create a status bar notification when a broadcast event occurs. Some examples of Android OS broadcast events include events sent when the battery of the device is low, when the device originally boots or when a picture is captured.

## The manifest file

Every application must have a manifest file named `AndroidManifest.xml` in its root directory that describes the set of all components that it uses, as well as the functionality and requirements of the application to the Android OS. `AndroidManifest.xml` is generated automatically during the build process, and the XML found within `Properties\AndroidManifest.xml` is merged with the XML file that is generated based on custom attributes. The declaration of the components is important since Android OS must be aware of which of them exists in the developed application, before it runs any of its code.

Except of just declaring the components used in the application, the manifest file:

- Creates the unique identifier of the application by giving a name to its Java package.
- States all the user permissions that are required by the application, such as Internet access, external SD card read/write access or SMS messages monitoring.
- Declares the minimum level of the Android Application Programming Interface (API) that the application requires, based on the APIs that it uses.
- Declares any external libraries that the application must be linked against.

## Resource management mechanism

Android applications consist of the source code and external resources, such as images, audio files, layout definitions, user interface strings and generally everything involved into the graphical presentation of the application. Dealing with application external resources makes it very easy to update its GUI without modifying the written code and optimize the application behavior for different device configurations.

Every resource is identified by a resource ID which can be used for referencing the resource from the application source code. For instance, for an image file named `image.jpg` located in the `res/drawable/` directory, the SDK tools generate a resource ID named `R.drawable.image`, that can be used as a reference in the source code and any other resources described in XML format, such as GUI layout definitions. This is a very practical and simple way to insert images or any other graphical elements into the GUI of the application

The most important reason of separating the source code from the static resources of the application is the ability of providing alternative resources for different device configurations. An example of this aspect is multi-language support and application localization. Since the application may run on devices in many regions, the resources should be handled appropriately according to the locale where the application will be used. Declaring the strings displayed in the GUI of the application in XML resource files make it possible to distribute the translations of these strings into separate files located in different directories according to a language qualifier that is appended to the resource directory's name. The resource system uses the ISO3166-1 two letter country codes.

For instance, if the XML resource file `strings.xml` containing the string values of the application in Italian is located in `res/values-it`, then the Android OS will automatically use this file when a device is configured to use Italian language according to the user's preferences.

## 2.3 Machine learning

Machine learning is a specific area of artificial intelligence which involves algorithms and methods that can learn from data and improve by gaining experience. In [34] Tom M. Mitchell provided a formal definition of machine learning as "A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ ". Machine learning allows customizable development of adaptable computer programs which operate according to the automated analysis of datasets rather than the intuition of software engineers who have programmed them. Machine learning overlaps considerably with Statistics, since both fields study data analysis. In this section, we discuss some topics in Machine learning that are relevant to the development of the battery lifetime prediction algorithm described in Chapter 4.

### 2.3.1 Instance-based learning

In Machine learning, instance-based learning or lazy learning is a family of learning algorithms that compare new problem instances with instances observed during the training process. The training process simply stores the data in memory and any computation on these data is postponed until an explicit query is made. The main four steps of instance-based learning when a new instance arrives are:

1. Search the memory for similar instances according to a similarity measure.
2. Retrieve the solutions of these instances.
3. Adapt these solutions to the current instance.
4. Store the new instance in memory.

A remarkable point regarding the instance-based learning methods is that they can create a different approximation to the solution of the problem for each new query instance, since different set of instances are considered as similar each time a new instance arrives. This behaviour makes these methods suitable for complex or incomplete problem domains especially where a complex solution can be represented as a set of less complex local approximations.

The major advantage of instance-based learning has over other methods of Machine learning is that the training process is very fast because it consists only of storing the training data, but the query time complexity can grow along with the size of the dataset. In the worst case scenario of  $n$  training instances stored in an array, the computational overhead of query time can be  $\mathcal{O}(n)$ , resulting in low performance.

### 2.3.2 The $k$ -nearest neighbour algorithm

The  $k$ -nearest neighbour algorithm ( $k$ -NN) is one basic instance-based learning method that is widely used due to its simplicity and its remarkably good performance and accuracy. According to this algorithm, each instance  $x_i$  has  $n$  attributes where the value of attribute  $r$  is denoted as  $a_r(x_i)$ . Thus,  $x_i$  can be represented as a  $n$ -dimensional vector or equivalently a point in the  $\mathbb{R}^n$  space:

$$x_i = (a_1(x_i), a_2(x_i), \dots, a_n(x_i)) \quad (2.2)$$

The training process of  $k$ -NN consists of just storing all training examples  $\langle x_i, f(x_i) \rangle$  in the memory. Considering the algorithm for real-valued function approximation, given a query instance  $x_q$  the estimation for  $x_q$  is computed as the mean of target values of the  $k$  nearest neighbours of  $x_q$ :

$$\hat{f}(x_q) = \frac{\sum_{i=1}^k f(x_i)}{k} \quad (2.3)$$

The  $k$  neighbours selected by the algorithm are the instances that have the minimum distance from query instance  $x_q$ . The distance between two instances  $x_i, x_q$  is calculated according to a similarity measure. Some of them are described next:

### Manhattan distance

The Manhattan distance is the distance between two points that is based on horizontal and vertical paths as seen in Figure 2.8a. The equation of this distance metric between instances  $x_i, x_q$  is given below:

$$d(x_i, x_q) = \sum_{r=1}^n |a_r(x_i) - a_r(x_q)| \quad (2.4)$$

### Euclidean distance

The Euclidean distance is the distance between two points defined as the square root of the sum of the squares of the differences between the coordinates of the points. The equation of this distance metric between instances  $x_i, x_q$  is given below:

$$d(x_i, x_q) = \sqrt{\sum_{r=1}^n (a_r(x_i) - a_r(x_q))^2} \quad (2.5)$$

The Euclidean distance between two points in 2D space is illustrated in Figure 2.8b

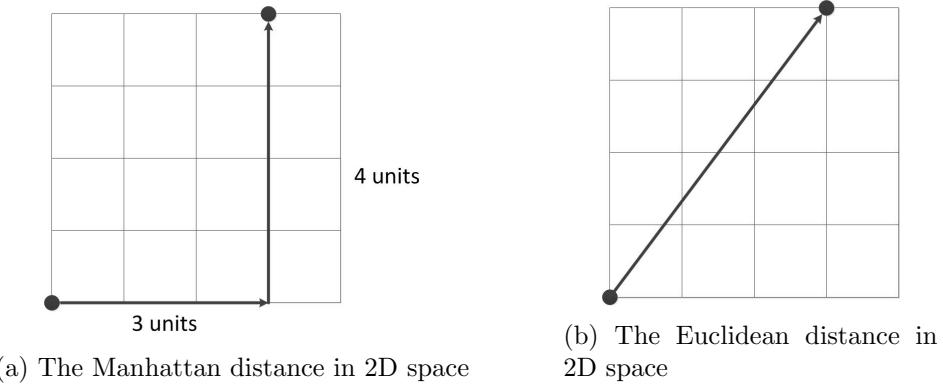


Figure 2.8: Differences between the Manhattan distance and the Euclidean distance

### Minkowski distance

The Minkowski distance is a metric that can be considered as a generalization of the Manhattan distance and the Euclidean distance. The equation of the Minkowski distance of order  $p$  between instances  $x_i, x_q$  is given below:

$$d(x_i, x_q) = \left( \sum_{r=1}^n (|a_r(x_i) - a_r(x_q)|)^p \right)^{\frac{1}{p}} \quad (2.6)$$

From the equation above it is clear that when we substitute  $p = 1$  we get the Manhattan distance while when we substitute  $p = 2$ , we get the Euclidean distance.

### Chebyshev Distance

The Chebyshev Distance or L-infinity norm is defined as the distance the greatest of the differences between two vectors along any coordinate dimension. The equation of this distance metric between instances  $x_i$ ,  $x_q$  is given below:

$$d(x_i, x_q) = \max_r |a_r(x_i) - a_r(x_q)| \quad (2.7)$$

Chebyshev distance is a special case of Minkowski distance when  $p \rightarrow \infty$  and can be used for ordinal as well as quantitative variables.

### Hamming distance

The Hamming distance between two vectors is simply defined as the minimum number of substitutions required to change one vector into the other. The Hamming distance was originally used for error detection and error correction in digital communication. The equation of the Hamming distance is given below:

$$\begin{aligned} d(x_i, x_q) &= \sum_{r=1}^n |a_r(x_i) - a_r(x_q)| \\ a_r(x_i) &= a_r(x_q) \Rightarrow d = 0 \\ a_r(x_i) &\neq a_r(x_q) \Rightarrow d = 1 \end{aligned} \quad (2.8)$$

It should be noted that all three aforementioned distance measures are only valid for continuous variables. For binary variables the Hamming distance is preferable, since it has the same results as the others and its computation overhead is extremely low, due to the fact that it can be computed efficiently using the  $\oplus$  operation between the two binary vectors. An example of the Hamming distance of two binary vectors is shown in Figure 2.9. The highlighted values in positions 1 and 5 are different between these vectors, thus the Hamming distance is 2.

1	1	0	1	0	1	1	1
0	1	2	3	4	5	6	7
1	0	0	1	0	0	1	1
0	1	2	3	4	5	6	7
Hamming distance = 2							

Figure 2.9: The hamming distance between two binary vectors

### 2.3.3 The distance-weighted $k$ -nearest neighbour algorithm

According to this variant of the  $k$ -NN algorithm, the contribution of each of the  $k$  nearest neighbours is weighted to the average, in accordance with their distance from the query instance  $x_q$ , giving higher weight  $w_i$  to closer neighbours. The estimation for  $x_q$  is computed as:

$$\hat{f}(x_i) = \frac{\sum_{i=1}^k w_i f(x_i)}{\sum_{i=1}^k w_i} \quad (2.9)$$

where the weight is defined as:

$$w_i = \frac{1}{d(x_i, x_q)^2} \quad (2.10)$$

In this variant of the algorithm it is reasonable to consider all instances and not only the  $k$  nearest neighbours, since their contribution to the result is weighted making a global estimation. The main disadvantage of this approach is the high time complexity of the calculations needed.

## 2.4 Summary

In this chapter we presented the background material of this thesis. We discussed some basic topics of how batteries work and discussed some ideal and non ideal-properties that several types of batteries have, focusing on lithium-ion batteries that modern portable devices use. We also presented the Android OS, which is the most popular platform of smartphone and tablet devices and discussed how Android OS manages power resources using the mechanism of wake locks. In the last part of this chapter we presented the  $k$ -NN instance-based learning algorithm, a variant of which our prediction model uses.

In the next chapter we present the related work of this project, consisting of various studies regarding battery modelling and lifetime prediction.

# Chapter 3

## Related Work on Battery Power Modelling

In this chapter we present different approaches for battery lifetime prediction that have been proposed as well as their advantages and disadvantages. This will be useful for completeness reasons as well as for locating and describing technical details related to this project.

In the literature there many approaches regarding battery modelling, battery lifetime prediction and generally power management in embedded devices. Many attempts have similar research directions, thus we present the related work in this field in five separate parts each highlighting significant research themes. The order we present these themes is from theoretical battery oriented approaches to usage pattern oriented approaches on which many battery lifetime prediction systems are based.

### 3.1 Battery Analytical Models

Analytical models make a description of the behaviour of batteries in an abstract manner. According to these models, the properties and the behaviour of the battery are modelled using sets of equations, some of them simple and some of them more complex. Generally, these models describe specific types of batteries and most of them use parameters that depend on the battery's specifications. For the implementation of these models it is necessary to determine these parameters which actually form their input data.

#### 3.1.1 Peukert's law Model

The simplest analytical model of prediction of battery life under constant load is based on empirical Peukert's relationship [28]:

$$\alpha = I^b L \quad (3.1)$$

where  $L$  is the lifetime of the battery,  $I$  is the current and  $\alpha > 0$ ,  $b > 0$  are constants which depends on the battery. In ideal circumstances,  $\alpha$  would be equal to the capacity of the battery, while  $b = 1$ . However, this power law does not hold when current is changing over time, thus it contradicts experimental observations. In practice  $\alpha$  has a value close to battery capacity and  $b$  lies between 1.2 and 1.7 for most batteries [33].

In their work [39], Rakhmatov and Vrudhula present an extended version of Peukert's law for variable loads. In Equation (3.1)  $I$  is replaced by the average current until  $t = L$ , yielding to the next equation:

$$\alpha = \left( \frac{\sum_{k=1}^n I_{k-1}(t_k - t_{k-1})}{L} \right)^b L \quad (3.2)$$

where  $t_k$  are the points of time of current change. Although the Equation (3.2) seems to be simple, the unknown variable  $L$  appears inside the  $n$ -term sum as well, since  $t_n = L$ . For  $n = 1$  Equation (3.2) reduces to Equation (3.1).

**Advantages** The model is simple and the results obtained by applying Peukert's law for predicting battery lifetimes give a reasonably good approximation for constant continuous loads.

**Disadvantages** As we mentioned before, the Peukert's law does not hold for variable loads (when the current average does not adequately represent the battery discharge conditions), therefore it cannot be used in practice. Even though the extended version of the model can handle variable discharge profiles, it is still not realistic, since only average discharge is taken into account and the recovery effect is not modeled.

### 3.1.2 Kinetic Battery Model (KiBaM)

The Kinetic Battery Model (KiBaM) is the analytical model proposed by Manwell and McGowan [30, 31, 32].

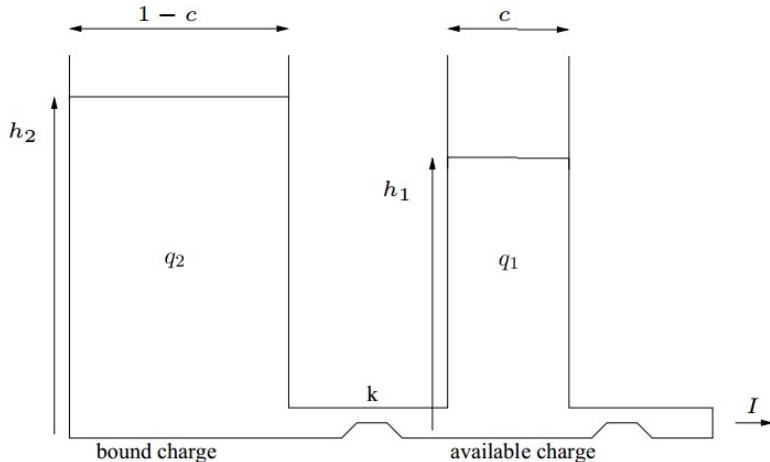


Figure 3.1: The two-tank model of KiBaM [23]

### Charge Model

According to this model, battery charge is distributed over two tanks, as shown in Figure 3.1. The first tank is the available-charge tank and the second is the bound-charge tank. The available-charge tank supplies electrons to the load, while the bound-charge tank supplies electrons to the available-charge tank. The width of each tank is  $c$  and  $1 - c$  respectively. The electrons are supplied with a rate that depends on the heights of the tanks and on a fixed parameter  $k$ . The change of the charge in both tanks is defined as a set of differential equations:

$$\begin{cases} \frac{dq_1}{dt} = -I - k(h_1 - h_2) \\ \frac{dq_2}{dt} = k(h_1 - h_2) \end{cases} \quad (3.3)$$

where  $q_1$  is the available charge and  $q_2$  is the bound charge. The height in each tank is given by the volume divided by the area, that is:

$$\begin{cases} h_1 = \frac{q_1}{c} \\ h_2 = \frac{q_2}{1 - c} \end{cases} \quad (3.4)$$

The differential equations can be solved using the Laplace transforms. Apart from the charge in the battery, KiBaM models the voltage during discharge. This model is described next.

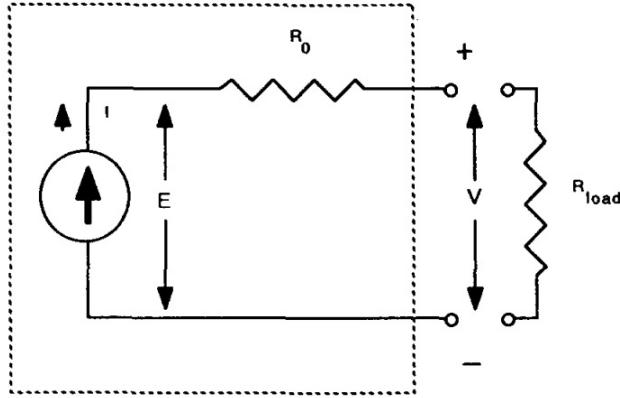


Figure 3.2: Schematic representation of the voltage model [30]

### Voltage Model

The KiBaM also models the voltage during discharge in which battery is described as a voltage source in series with a resistance, as illustrated in Figure 3.2. The voltage is given by:

$$V = E - IR_0 \quad (3.5)$$

where  $I$  is the discharge current,  $R_0$  the internal resistance and  $E$  is the internal voltage which is given by:

$$E = E_0 + AX + \frac{CX}{D - X} \quad (3.6)$$

where  $E_0$  is the internal voltage of the charged battery. The parameter  $A$  represents the initial variation of the internal battery voltage with the state of charge, parameters  $C$  and  $D$  reflect the decrease of voltage when the battery is progressively discharged and  $X$  is the normalized charge removed from the battery. The aforementioned parameters can be obtained from discharge data.

**Advantages** The KiBaM model was developed to model lead-acid storage batteries. It has also been used for modelling Ni-MH batteries. Moreover, the two-tank model is simple, accurate and describes both the rate capacity and the recovery effect. Because of these properties, KiBaM is widely used in battery simulations.

**Disadvantages** However, the equations of the model does not hold for the modern batteries that are used in smartphone and tablet devices such as lithium-ion batteries which have a sloped discharge profile.

### 3.1.3 Rakhmatov and Vrudhula's Diffusion Model

In their work [39], Rakhmatov and Vrudhula present an analytical model that describes the diffusion process of the active material in the battery. In this model, the processes at both anode and cathode are assumed identical, therefore only one of the electrodes is considered.

#### Model Description

According to the model, the diffusion is considered to be one-dimensional in a region of length  $w$ .  $C(x, t)$  is the concentration of the active material at time  $t \in [0, L]$  and distance  $x \in [0, w]$  from the electrode, where  $L$  is the time when the reaction can no longer take place at the electrode surface. The battery lifetime can be determined by computing the specific time at which the concentration  $C(0, t)$  drops below the cutoff threshold  $C_{cutoff}$ . Concentration behaviour due to one-dimensional diffusion is given by the Fick's laws [4]:

$$\begin{cases} -J(x, t) = D \frac{\partial C(x, t)}{\partial x} \\ \frac{\partial C(x, t)}{\partial t} = D \frac{\partial^2 C(x, t)}{\partial x^2} \end{cases} \quad (3.7)$$

where  $J(x, t)$  is the flux (diffusion speed) of the active material at time  $t$  and distance  $x$  and  $D$  denotes the diffusion coefficient. Laplace transform can be used in order to obtain analytical solutions from this set of differential equations. From these solutions the next equation can be derived relating the battery lifetime  $L$ , the load and the battery parameters:

$$\alpha = \int_0^L \frac{i(\tau)}{\sqrt{L-\tau}} d\tau + 2 \sum_{m=1}^{\infty} \int_0^L \frac{i(\tau)}{\sqrt{L-\tau}} e^{-\frac{\beta^2 m^2}{L-t}} d\tau \quad (3.8)$$

where  $\beta = \frac{2}{\sqrt{D}}$ ,  $\alpha = vFA\sqrt{\pi DC^*}\rho(L)$ ,  $C^*$  is the initial concentration and  $\rho(L) = 1 - \frac{C(0, L)}{C^*}$ .

The authors also define Equation (3.8) for the special case of constant load. In this case  $i(\tau) = I$  and the Peukert's law applies. The constant  $I$  can be brought out of the integrals, yielding:

$$\alpha = 2I\{\sqrt{L} + 2 \sum_{m=1}^{\infty} [\sqrt{L}e^{-\frac{\beta^2 m^2}{L}} - \beta m \sqrt{\pi} \bar{\Phi}(\frac{\beta m}{\sqrt{L}})]\} \quad (3.9)$$

where  $\bar{\Phi}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2} dy$ , which is the complementary error function.

A good approximation for  $\alpha$  is the first ten terms of the infinite series. Thus, a prediction of the battery's lifetime can be made given the discharge current. Before the proposed model can be used, quantities  $\alpha$  and  $\beta$  need to be estimated from experimental data.

**Advantages** The model of Rakhmatov and Vrudhula can be used to model lithium-ion batteries and performs really well having very high accuracy, since it takes into consideration both the rate-capacity and the recovery effects.

**Disadvantages** Although the model's performance is high, the computational load makes the on-line battery prediction highly time-consuming in practice. For simplicity, this model considers both electrode reactions as identical, which is not true in reality.

## 3.2 Battery Stochastic Models

While in analytic modelling the battery properties are modelled using sets of equations, in stochastic modelling the battery charging and discharging are described as stochastic processes. We will next

describe some models of battery that fall into this category.

### 3.2.1 Chiasserini and Rao

Between 1999 and 2001 Chiasserini and Rao published a series of papers [7, 6, 9, 8] regarding battery lifetime stochastic modeling using discrete-time Markov chains.

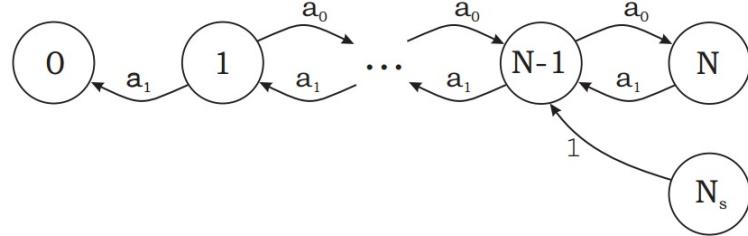


Figure 3.3: The basic Markov chain battery model by Chiasserini and Rao [7]

#### Basic Model

In their first work [7], they present a model of a lithium-ion battery of a mobile communication device that transmits packets. The battery is described as a discrete-time Markov chain with  $N+1$  states as shown in Figure 3.3. The number of states represents the number of charge units available, while one charge unit corresponds to the energy needed to transmit one packet. According to this basic model, each time slot either a charge unit is consumed with probability  $a_1 = q$  or a charge unit recovers with probability  $a_0 = 1 - q$ . The battery is considered to be empty either when the state 0 is reached or  $T$  charge units have been consumed, where  $T$  is equal to the capacity of the battery.

#### Model Extensions

In the same work, they present an extension of this model where at each time slot more than a unit can be consumed, modelling fluctuations in battery consumption. They also strengthened their model by adding the property of non-zero probability of staying in the same state, modelling the situation where neither consumption nor recovery takes place in a single time step.

In their following work on stochastic models [6, 9, 8], Chiasserini and Rao improve their model even more in order to make it more realistic. In their extended model, the probability of recovery becomes *state dependent* and *phase dependent*. More specifically, when less number of charge units are available, the recover probability decreases, making the probability dependent on the state. Moreover, they introduced the phase number  $f$  as a function of the charge units that has been consumed. As the number of the consumed time units increases,  $f$  also increases causing the decrease of the recovery probability making the probability phase dependent.

The extended model is illustrated in Figure 3.4. During idle periods, in state  $j$  the battery either recovers one charge unit with probability  $p_j(f)$  or stays in the same state with probability  $r_j(f)$ . The recovery probability in state  $j$  and phase  $f$  is defined as:

$$p_j(f) = q_0 e^{-(N-j)g_N - g_C(f)} \quad (3.10)$$

where  $g_N$  and  $g_C(f)$  depend on the recovery capability of the battery. The probability to remain in the same state of charge  $j$  while being in phase  $f$  is defined as:

$$r_j(f) = q_0 - p_j(f) \quad (3.11)$$

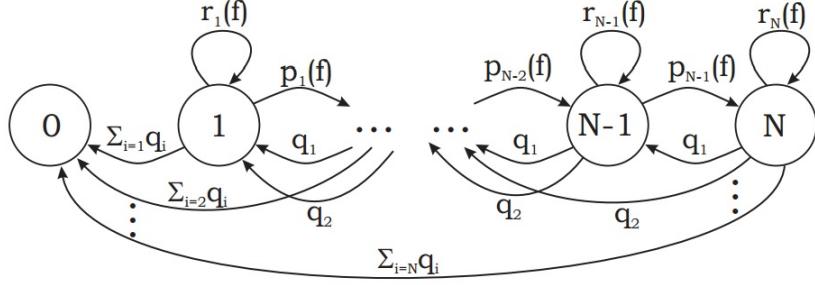


Figure 3.4: The extended Markov chain battery model by Chiasserini and Rao [8]

It is important to mention at this point that different loads can be modelled by setting the transition probabilities in an appropriate way, but it is impossible to model fixed load patterns and compute their impact on the battery life, since the order of taken transitions is uncontrollable [23].

**Advantages** The model of Chiasserini and Rao were the first stochastic models regarding battery lifetime. The results of their work show that the stochastic models give a good qualitative representation of battery behaviour, modelling charging process, discharging process and the recovery effect.

**Disadvantages** Although the model of Chiasserini and Rao has been proven to work theoretically, it is not certain how well this model performs with real batteries, since there are no numerical data proving high accuracy in battery lifetime and only relative numbers of lifetime are compared. Moreover, the rate capacity effect is not modelled increasing the uncertainty of the model for real battery data.

### 3.2.2 Fluid Queue Battery Model

A fluid queue is a mathematical model that is used to describe the level of fluid in a storage facility subject to randomly determined periods of continuous filling and emptying. The model was first introduced by Pat Moran in 1954 [35] and has many applications in computer networks modelling the performance of routers [15], the IEEE 802.11 protocol [3] and P2P file sharing systems [14] to mention some.

Recently it was proposed that fluid queue networks could be used to model the problem energy producers face in routing and allocating energy produced from renewable sources. Understanding the way these networks work will become very important as renewable power becomes a popular source of generating capacity. [21].

#### Formal Definition

Formally, a fluid queue is a bivariate stochastic process  $(J_t, X_t)$  with an associated input rate vector  $\lambda$  and service rate scalar  $\mu$ . The process  $J_t$  is a Continuous Time Markov Chain on the state space  $S = \{1, 2, \dots, n\}$  usually referred to as *background process*, *driving process* or *environment process* and  $\mathbf{r} = \lambda - \mu \cdot \mathbf{1}$  is an  $n$ -dimensional vector.  $X_t$  is a stochastic process such that at time  $t$ , when  $J_t$  is in state  $i$  the evolution of the process satisfies:

$$\frac{dX_t}{dt} = \begin{cases} r_i & \text{if } X_t > 0 \\ \max(r_i, 0) & \text{if } X_t = 0 \end{cases} \quad (3.12)$$

The output process of a fluid queue  $Y_t$  is given by:

$$Y_t = \begin{cases} \mu & \text{if } X_t > 0 \\ \lambda_i & \text{if } X_t = 0 \end{cases} \quad (3.13)$$

The process  $X_t$  is continuous and piecewise linear with the rate determined by the process  $J_t$ . The time period when the fluid queue has a positive fluid level is referred to as *busy period*. This is the time period starting when fluid initially enters the queue ending when the queue is empty again.

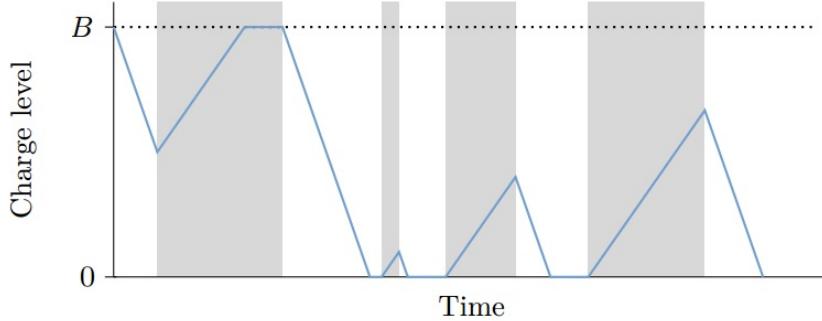


Figure 3.5: Sample trace from a fluid queue [20]

### Fluid queues modelling battery lifetime

Recently fluid queues were used to model battery life of modern electronic devices [22]. More specifically it was shown using the fluid queuing mathematical model that a power-save mode activated when battery level falls a specific threshold level  $V$  (ie 20%) can significantly improve the battery life. Battery was modelled as a reservoir having infinity capacity of  $B$  and the flow of charge as an incompressible fluid, meaning that battery charging can be thought as adding fluid to the reservoir. The charge level in the battery was modelled subject to random charging and discharging periods one following consecutively the other. An example of this approach is illustrated in Figure 3.5. The figure shows a sample trace from a fluid queue model with just two states {charging, discharging}. The grey background represent the time in charging periods while the white background areas represent periods when the battery was discharging.

**Advantages** The fluid queue modelling of battery is a new novel idea that has introduced a new field of study. The theoretical results seem to be very promising showing that this model can be used to optimize some power requirement of battery usage. This theoretical model is suitable for off-line data analysis and it can be used for prediction of battery lifetime.

**Disadvantages** The model has high computation load and may not be suitable for on-line analysis of data, especially when the state space becomes very big and new parameters are introduced to the model. Although the model has been proven to work in theory there is no evaluation yet using real battery data in the literature.

### 3.2.3 Others

There are also other stochastic models that describe the battery properties such as the Stochastic version of KiBaM [40]. However, that system cannot model Lithium-ion batteries like its predecessor and is beyond the scope of this thesis.

## 3.3 Power Model Generation

The analytical and stochastic models that were previously discussed have limitations that make them unusable in practice for on-line adaptive battery life predictions in mobile and embedded

systems. Most of them require a big number of parameters as their input and the computational cost of the calculations of these models is very high, especially for the analytical simulation models.

Another approach for estimating battery dissipation is to generate a power model of the device and predict the battery lifetime based on this. Researchers have successfully generated such models for smartphones correlating the battery consumption with the hardware components of the devices. Some models require external equipment for power measurements, while others make predictions based on benchmarks of each hardware element of the device. Some recent studies in this field are presented next.

### 3.3.1 Application-level Prediction of Battery Dissipation

Krintz, Wen and Wolski in their work [27] present a system that predicts the power consumption of an arbitrary application based on application-level measurements of battery drain for a set of benchmarks running on the entire device, obviating the need for benchmarks running for isolated hardware components that other approaches are focused on.

#### System Overview

This system observes the dissipation characteristics of an application according to a black-box approach where the drain battery behaviour is being profiled for a set of power benchmarks that each executes one type of instruction. The authors define four categories of instruction types: integer register operations, integer stores and loads, floating point register operations and floating point stores and loads.

In order to make a prediction, the system constructs a set of *dissipation curves* from the benchmark profiles and determine the *battery dissipation rate* associated with a specific instruction type. A comparison of these rates is illustrated in Figure 3.6. The dissipation curves change over time, so the specific instruction's impact on the battery lifetime will change following the same pattern. Therefore, a battery dissipation curve can be constructed for the target random application. The rate of drain of the application is computed directly from the rate of dissipation of the benchmarks associated to the specific instructions of the application, observed by execution statistics.

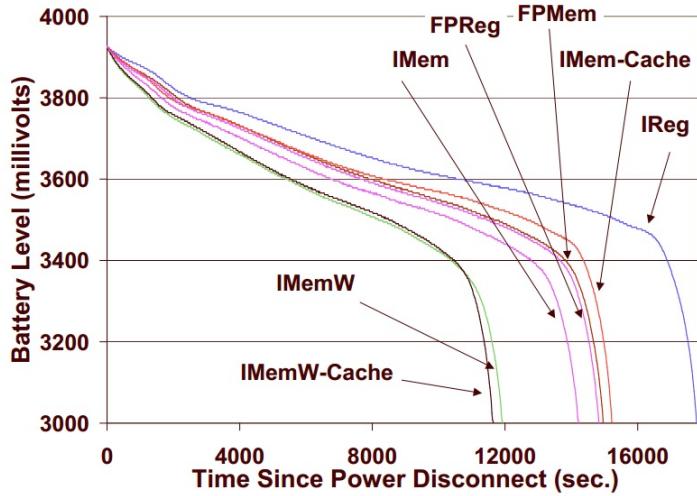


Figure 3.6: Comparison of battery drain rates for all benchmarks [27]

**Advantages** The power model of Krintz, Wen and Wolski uses benchmarks associated with specific instruction types running on the entire device and not to each hardware component and is capable of implementation as part of a dynamic compiler that can predict the battery consumption

of a program. This technique may also have many other applications such as in voltage scaling, dynamic code generation, quality-of-service etc.

**Disadvantages** However, the benchmarks used in this approach consume a lot of energy themselves, especially when the frequency of execution is high. Reducing the frequency of sampling may result into low accuracy of the system. Moreover, this power model considers only CPU related power consumption and does not take into consideration wireless communications, file input/output and display, which are highly used in portable devices. Thus, since their impact to battery lifetime is not taken into account, the model is oversimplified for real devices.

### 3.3.2 PowerBooster and PowerTutor

Zhang et al. in [45] present PowerBooster, an automated power model generation method which uses the battery voltage sensors of Android smartphone devices and knowledge of battery discharge behaviour to monitor power consumption and PowerTutor, a power management tool that uses the model generated by PowerBooster in order to make accurate on-line power estimations.

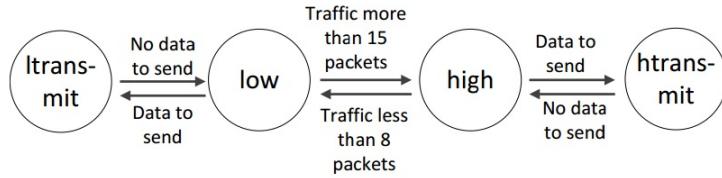


Figure 3.7: Wi-Fi interface power states according to PowerBooster [45]

#### PowerBooster

PowerBooster produces models that take into account power consumption in power-intensive components such as CPU, GPS, LCD, audio as well as Wi-Fi and cellular communication components. The proposed models are based on the influence of the power management and states of hardware modules on the system's overall power consumption. The states of Wi-Fi interface according to PowerBooster are illustrated in Figure 3.7. Hardware units are associated with system variables and coefficients that are used in power estimation of each unit.

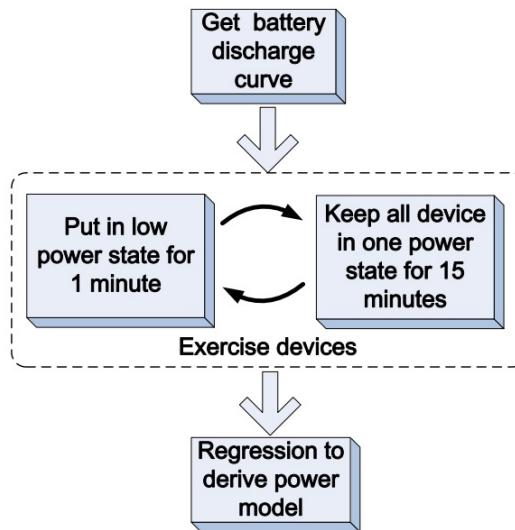


Figure 3.8: Power model construction [45]

The modelling process is illustrated in Figure 3.8 and consists of three steps:

1. *Computation the battery discharge curve for each hardware component individually*, where the battery starts in a fully charged state.
2. *Computation of the battery consumption for each component's state*. In this step the state of an individual hardware component is varied while others are working on low-power states. For the computation of power consumption, the battery voltage is logged at the beginning and at the end of the power discharge time interval and the device is placed in a low-power state immediately before any voltage recording in order to eliminate the impact of the voltage decrease across the battery's internal resistance.
3. *Regression to derive the power model*. After the collecting the data, they are used in order to compute the average power consumption within the fixed time intervals. Then, multi variable regression is used to construct the power model.

$$\begin{pmatrix} P_0 \\ P_1 \\ \vdots \\ P_n \end{pmatrix} = \beta_i \cdot \begin{pmatrix} U_{01} \\ U_{11} \\ \vdots \\ U_{n1} \end{pmatrix} + \dots + \beta_m \cdot \begin{pmatrix} U_{0m} \\ U_{1m} \\ \vdots \\ U_{nm} \end{pmatrix} + c \quad (3.14)$$

where  $U_{ij}$  denotes system variable  $i$  in the  $j$ th state,  $P_j$  is the power consumption when all system variables are in the  $j$ th state and constant  $c$  is the minimum system power consumption. The inputs for regression are the systems variables and the outputs are the power coefficients  $\beta_i$ .

### PowerTutor

PowerTutor[37] is an on-line power estimator based on PowerBooster implemented for Android platform devices. This application provides real-time power estimates for hardware components such as CPU, LCD display, GPS, Wi-Fi and cellular interfaces. PowerTutor can be used by developers in order to identify inefficient behaviour caused by inappropriate use of hardware components as well as smartphone users in order to determine power hungry applications, a process that may help them decide which of them to use or buy.

PowerTutor has been released in the Android market in 2009 and has been downloaded by more than 100,000 users, while in 2011 the source code of the project has been released.

**Advantages** The power model generator takes consideration almost all hardware components of smartphone devices making accurate estimations. The high accuracy of this approach is shown in the evaluation of the work using real Android devices. Moreover, built-in sensors are used for the voltage data logging and there is no need for external measurement equipment.

**Disadvantages** However, an off-line computation of all voltage curves regarding each hardware component is required before any power estimation and users have to choose a trade-off between model construction time and estimation accuracy.

#### 3.3.3 Others

Another work that is relevant to this field is DevScope [24]. DevScope is a automatic power analysis tool very similar to PowerBooster that generates accurate power models and overcomes several limitations of on-line power modeling with a built-in Battery Monitoring Unit. Another power model was presented in [10], which focuses on energy consumption of mobile devices running an older Operating System called Palm OS.

## 3.4 Context-aware power management

A recent definition of context-awareness is the one of Dey and Abowd [1] who defined it as "any information that can be used to characterise the situation of an entity, where an entity can be a person, place, or physical or computational object". They went on to define context-aware computing as "the use of context to provide task-relevant information and/or services to a user, wherever they may be". The concept of context-aware power management has already been proposed and used in relation to assisted living and residential monitoring [44]. Recently, the same approach was adopted to the field of battery management in the system CABMAN that will be described next.

### 3.4.1 Context-aware Battery Management for Mobile Phones (CABMAN)

#### System Overview

CABMAN is a system created to warn the user after detecting that the phone battery could run out before the next charging opportunity is encountered [41]. The substantial algorithms of this system predict the estimated upcoming charging opportunity availability, the total duration of call-time required by the user in the interim, and the lifetime of the battery given that the current set of applications would continue to execute. Several algorithms were proposed to process users location traces and call-logs in order to make the foreseen predictions. CABMAN consists of three key components: Firstly, the use of context information such as location to predict the next charging opportunity, secondly more accurate battery life prediction based on a discharge speedup factor and thirdly the notion of crucial applications, such as telephony.

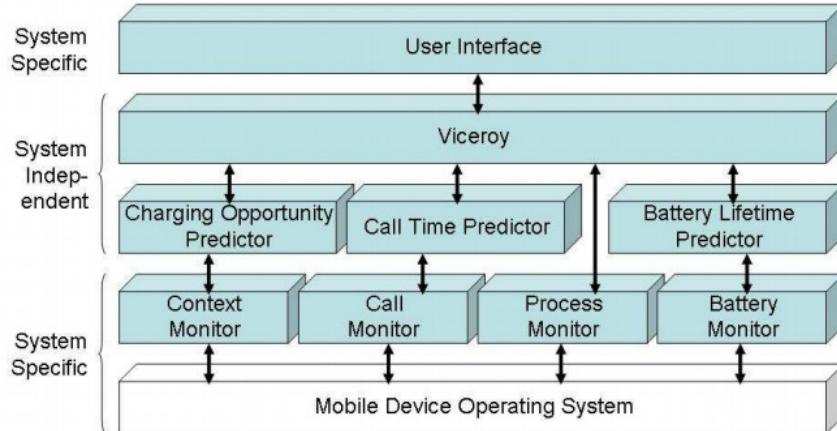


Figure 3.9: CABMAN system architecture [41]

#### System Design

The main architecture of the system is illustrated in Figure 3.9, where we can see a multi-layered design consisting of several modules in each layer. The most vital components of the system are the *Charging Opportunity Predictor*, the *Call Time Predictor* and the *Battery Lifetime Predictor* which will be discussed next.

**Charging Opportunity Predictor** This predictor is based on location sensing as a way of finding the next charging opportunity. It uses the ID of the cell that a phone is connected to, by marking a subset of them as opportunities of charging (ie those at home or work) and estimate the time needed for the user to reach them. The time prediction is based on pattern-matching the current trace of cell movement against a dataset of past movement traces.

**Call Time Predictor** Assuming that telephony is the most crucial application of mobile phones, this module predicts the call time needs of the phone's user. The prediction is made in a dynamic manner by using past telephony behavior of the user in order to find the average amount of minutes that the user needs regarding giving or receiving calls during each hour of the day. This is used as an upper bound on the call time that is required in a specific time interval.

**Battery Lifetime Predictor** In order to make an estimation on the battery lifetime, this module uses the battery's discharge curve when the device is in idle mode called the *base curve*. The predictor compares the discharge when the phone is used regularly with many applications running, against the base curve. It is important to mention here that this procedure requires an off-line computation of the base curve before making any prediction. After computing the base curve, the predictor measures the battery capacity  $c_1$  and  $c_2$  at times  $t_1$  and  $t_2$  when the device is used regularly. Then, it finds the time instances  $t_3$  and  $t_4$  which correspond to  $c_1$  and  $c_2$  in the base curve and compute the discharge speedup factor defined as:

$$\frac{t_4 - t_3}{t_2 - t_1} \quad (3.15)$$

Then the lifetime of the battery in the base curve is divided by the discharge speedup factor to obtain an estimation on the remaining lifetime of the battery.

**Advantages** CABMAN is a sophisticated system that uses context information in order to predict the next charging opportunity as well as the remaining lifetime of the battery. According to the evaluation of the system using traces of real users and against experiments on real devices, the battery lifetime prediction algorithm that is used is very accurate for PDAs and laptops running several applications, having average errors of between 4 and 12 minutes depending on the device used .

**Disadvantages** It should be pointed out that charging-opportunity predictor and call-time predictor perform reasonably well for an average user whose life entropy is not very high, implying that additional context information would be needed in order to improve the accuracy of these prediction algorithms for users with different lifestyles. In addition, the majority of users nowadays, charge their phones in their cars while driving. For such users, it is not always possible to associate charging with location. The Charging Opportunity Predictor would have to associate charging with the users presence in the car, and should be able to predict when the user will be in the car next. This would also require additional context information to be logged.

### 3.5 Analysis of battery usage patterns in smartphones

The initiative for energy efficient design in smartphone devices was enhanced as a result of human-battery interaction (HBI) observations, a reciprocal process which includes the users' reaction to limited battery lifetime through information received by indicators for charge levels and battery interfaces [38]. Other studies, such as [13] focus on user interaction with the device and the applications installed and users' behavior regarding the usage of battery hungry smartphone features such as Wi-Fi, 3G and GPS. According to these observations, users should be provided with options on how to better manage the remaining battery, and, to some extent, automated power features can also help them use the device as intended. A new research field based on this concept focuses on battery lifetime prediction models which depend on usage patterns through definition of several mobile states, according to the device's functions. Some recent and very interesting approaches are discussed next.

### 3.5.1 Kang, Seo and Hong's Personalized Battery Lifetime Prediction

Kang, Seo and Hong have recently presented a personalized battery prediction model for mobile devices based on usage patterns in [25] and their data analysis using this model in [26]. In their work, they define a model of a mobile device having several states that affect battery consumption and make an on-line estimation of the remaining battery lifetime based on the usage time and battery's dissipation over these states.

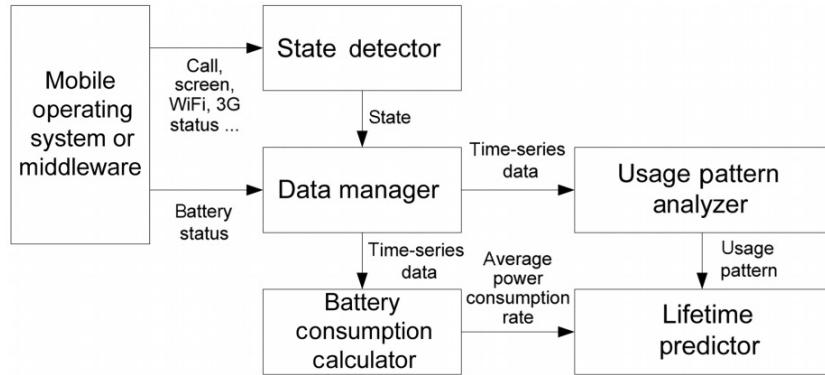


Figure 3.10: Architecture of the personalized prediction system of Kang, Seo and Hong [25]

### System Design

Figure 3.10 illustrates the architecture of the system. The State Detector discovers the operational state of the mobile device by collecting data regarding the call status, the Wi-Fi and 3G status and the screen status from the operation system of the device. The information of the detector then passes to the Data Manager unit which stores battery related data and usage duration time in each state. The Battery Consumption Calculator computes the average battery consumption and the use time rate for each state based on the real-time data of the Data Manager. Then, the Usage Pattern Analyzer determines a usage pattern distribution representing the time spent in each state of the device. Finally, the Lifetime Predictor predicts the remaining battery life, based on the average battery consumption rate and the usage pattern.

Subsystem	S0	S1	S2	S3	S4	S5	S6	S7
LCD	ON	ON	ON	ON	OFF	OFF	OFF	OFF
VOICE	ON	ON	OFF	OFF	ON	ON	OFF	OFF
DATA	ON	OFF	ON	OFF	ON	OFF	ON	OFF

Table 3.1: Possible states of a mobile device according to Kang, Seo and Hong

### Prediction Model

The model defines a mobile device having  $n$  possible states. Conceptually, each state of the device is constructed as the set of the subsystems of the device described by binary variables according to whether they are activated or not. All possible states of a mobile device using the subsystems LCD, VOICE and DATA is shown in Table 3.1. According to the model, the average battery consumption in each state is defined as a vector:

$$\mathbf{B} = (B_1, \dots, B_n) \quad (3.16)$$

where  $B_i$  is the average battery consumption rate in state  $i$ . The usage pattern is also defined as

a vector:

$$\mathbf{p} = (p_1, \dots, p_n) \quad (3.17)$$

where  $p_i$  is the use time rate in state  $i$ , which satisfies the equation:

$$\sum_{i=1}^n p_i = 1 \quad (3.18)$$

The remaining battery lifetime is predicted using the following equation:

$$T = \frac{V}{\sum_{i=1}^n p_i B_i} \quad (3.19)$$

where  $T$  is the estimated value of the battery lifetime and  $V$  is the total battery capacity. Since each user has a different vector  $\mathbf{p}$  according to his usage characteristics, each user's battery has a different remaining lifetime. The model can be also used to determine variable life patterns, such as usage during weekdays and weekend etc. This can be achieved by dividing each week into several modes based on these variable patterns and predict the lifetime using the pattern vector  $\mathbf{p}$  of the specific mode.

**Advantages** This model predicts battery lifetime on an individual basis, since it takes into account personal usage characteristics that previous prediction models which use a static battery consumption rate do not consider. Moreover, this model is capable of real-time prediction since it does not require high computational time for making the battery's lifetime estimations.

**Disadvantages** However, the authors present only statistics regarding the usage patterns of real smartphone users and a case study demonstrating the usage of the model. Although this analysis is interesting in a conceptual manner, there is no evaluation regarding the accuracy of the prediction model or any comparison to other modeling approaches making its real performance questionable.

### 3.5.2 Others

Falaki et al. in their work [13] have done a wide survey regarding smartphone usage characterizing user interactions and their impact on the device's battery. They show by analysing traces from users of Android and Windows Mobile smartphone platforms that although the variance in energy usage is high even for the same user, there are patterns in the user's behavior that can be considered for accurate estimations. The large scale work of Oliver and Keshav in [36] complements the study of Falaki et al. evaluating a huge dataset of Blackberry related data. In the same work they present the *Energy Evaluation Toolkit* (EET) which is a tool that gives developers the opportunity to evaluate their applications in terms of battery consumption and the *Energy Management Oracle* (EMO) which is a framework designed to provide hints regarding battery consumption to applications in runtime.

## 3.6 Summary

In this chapter we presented various studies relevant to our project, pinpointing their advantages and disadvantages. Firstly we discussed about analytical battery models which model the behaviour of the battery using sets of complex equations. Another category discussed here is the stochastic models which describe the battery's charging and discharging behaviour as stochastic processes. We also described how power modelling tools generate battery models according to hardware benchmarks and how context-aware power management systems use context information

in order to make predictions. In the last part of this chapter we focused on studies covering battery models that are based on usage patterns.

In the next chapter we present our prediction model which is capable of making battery lifetime estimations based on usage patterns. We also present Open Battery tool, an Android-based application that logs battery and system related data and provides statistics of battery usage.



# Chapter 4

## Open Battery: a battery data analysis software tool

In this chapter we present our prediction model, which makes battery lifetime estimations based on usage characteristics using an instance-based learning engine. We also present the design and implementation of Open Battery, a battery data analysis application for Android-based devices which provides battery and system diagnostics, as well as battery lifetime estimations using the aforementioned prediction model.

### 4.1 Battery lifetime prediction model based on usage patterns

Most of the battery dissipation prediction models and systems presented in Chapter 3 are either not capable of on-line analysis in lightweight devices such as smartphones and tablets due to their heavy computation overhead or they need specific battery and hardware related data that are not always applicable in order to work properly and give accurate results. In this section we present our prediction model which is based on usage pattern recognition and aims to tackle the problems that previously mentioned models have.

#### 4.1.1 Exploiting usage characteristics for battery lifetime prediction

In Chapter 3 we presented various studies regarding battery lifetime prediction from different research perspectives. Some of the models presented are highly used in practice, while others are more theoretical. However, the adaptation of most of these models in portable devices for on-line analysis of data and battery lifetime prediction is not always applicable.

Analytical and stochastic models are designed for off-line analysis and need a lot of time-consuming complex calculations in order to predict the battery dissipation. The heavy workload of these models make their adaptation to lightweight portable devices impractical. Moreover, they use several parameters taken from the battery's specifications as input which can only be provided hard-coded. On the contrary, the context-aware power management systems do not involve complex equation solving but the data they use include sensitive information, such as GPS location. However, there are some major privacy concerns regarding the use of such sensitive data, making the practical use of these systems questionable. The power model generation tools use the hardware specifications of specific devices for modelling and some of them use external equipment for measurements that cannot be taken directly from the OS of the device. Taking into consideration the diversity of Android devices nowadays, we can clearly see that these models cannot work equivalently in all devices.

There are many studies presented in Chapter 3 that tried to identify high-level characteristics of smartphone usage. The main idea behind these approaches is that users tend to have recurrent behaviour on how they use their devices. Users usually use Wi-Fi connectivity while they are at home and 3G while they are away. Moreover, many users do not use their phones at early hours

in the weekdays since they are at school or university if they are young or are at work if they are older. Others have their phone charging at night when the phone is in idle mode and check their emails and social messages every morning and every night before sleeping.

These characteristics indicate different device usage and different battery consumption in each case. In our approach we try to model and recognise these patterns dynamically as new data are becoming available and predict the future lifetime based on them.

#### 4.1.2 Model overview

According to a very interesting and recent survey presented in [11], the future direction of power management in Android-based devices is applications that use a learning engine for prediction instead of statically defined profiles. Our model follows this trend and uses an instance-based learning engine in order to predict the future battery lifetime based on usage patterns. A general picture of our model is illustrated in Figure 4.1.

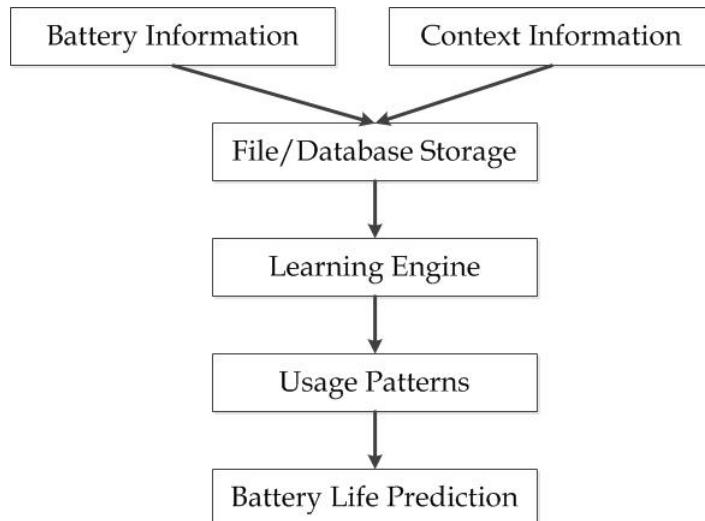


Figure 4.1: Prediction model using a learning engine based on usage patterns

The model uses periodically logged battery and context related information that is considered as the input of the instance-based learning engine. The term context information refers to information regarding power hungry subsystems of the device, such as CPU usage, display, Wi-Fi and cellular connectivity. The learning engine is capable of recognizing patterns relevant to the device usage which are used to make an estimation of the future battery lifetime.

#### 4.1.3 Preliminaries

In this section we present the basic blocks of our prediction engine. Before discussing the details of the algorithm, it would be very useful to make a short introduction to these fundamental blocks in order to have a better understanding on how the proposed method works.

##### Subsystem state

In our work, we consider several power consuming subsystems that a portable device may have, including CPU, OLED display screen, Wi-Fi and cellular 3G interfaces and GPS. According to our model, each subsystem is modelled as a binary variable according to its current usage state (ON/OFF). We list the subsystems that we take into consideration in Table 4.1. While we use these specific subsystems, it is very simple to add to the model other types such as telephony, audio, video, Bluetooth and many others. We chose this set of subsystems because they are the most power consuming features of a modern portable device.

Subsystem	ON state	OFF state	Description
CPU	1	0	CPU is considered to be in ON state if its usage level (%) is above the global average of CPU measurements. Otherwise is considered to be in OFF state.
Display	1	0	The display is considered to be in ON state if the screen is active. Otherwise it is considered to be in OFF state.
Wifi	1	0	Wifi is considered to be in ON state if the user uses a Wifi connection. Otherwise it is considered to be in OFF state.
3G	1	0	3G is considered to be in ON state if the user uses a 3G cellular connection. Otherwise it is considered to be in OFF state. If the device do not support 3G connectivity, the default state is the OFF state.
GPS	1	0	GPS is considered to be in ON state if the user uses the GPS. Otherwise it is considered to be in OFF state.

Table 4.1: Different subsystems and their states according to Open Battery prediction model

### Device state (Context Information)

The device state is the context information of our model. The device is modelled as an entity that consists of its subsystems. More formally, we define the device state as a vector  $d$  of size  $n$  that consists of the current states  $s_i$  of its  $n$  subsystems:

$$d = (s_1, s_2, \dots, s_n) \quad (4.1)$$

Figure 4.2 illustrates an example of the device state notation. Vector  $(1,0,1,0,1)$  denotes a device which has CPU usage above the global average, its display screen is not active, has connected to Wi-Fi, is not connected to 3G network and has its GPS system activated.

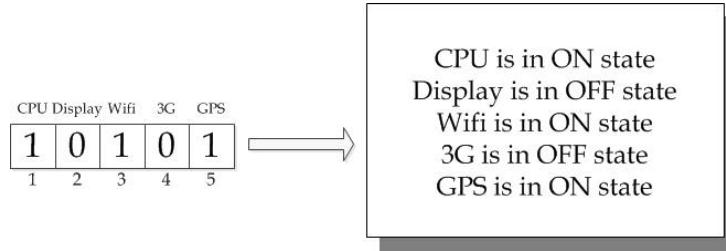


Figure 4.2: An example of a device state in Open Battery prediction model

### Battery state (Battery Information)

The battery state consists of information related to the battery at a specific moment. This includes the battery capacity and the current status of the battery. Formally, the battery state is modelled as a tuple:

$$b = (c, \sigma) \quad (4.2)$$

where  $c \in \mathbb{Z}$  is the battery capacity measured in % and  $\sigma \in \{\text{charging}, \text{discharging}\}$  is the status of the battery. For example, vector (34, charging) represents a battery that has capacity 34% and it is charging.

## Measurement

A measurement is an entity that consists of the appropriate battery and context information needed for our model concatenated with the timestamp when it was taken. A measurement is a vector defined as:

$$m = (t) \| b \| d \quad (4.3)$$

where  $t \in \mathbb{Z}$  is the timestamp of the measurement. The measurements are taken and logged periodically in a constant rate (ie. every 1 minute) and saved in files and in database storage. The measurement vector (251668, 34, charging, 1, 0, 1, 0, 1) denotes a measurement with timestamp 251668, that has been taken from a device with a battery that has capacity 34% currently charging and has CPU usage above the global average, its display screen is not active, has connected to Wi-Fi, is not connected to 3G network and has its GPS system activated.

## Usage Pattern

The model was designed to recognise device usage patterns and predict the battery lifetime according to them. The usage patters in our case are consecutive series of device states taken from logged measurements. A usage pattern of size  $n$  containing device states measured in a fixed rate is defined as a  $n$ -tuple:

$$p = (d_1, d_2, \dots, d_n) \quad (4.4)$$

For instance, the pattern  $p_1 = ((10001), (10001), (10001), (11001))$  measured in 1 minute intervals denotes the behaviour of a person who uses his device in high CPU power and has the GPS system activated for three consecutive minutes and then suddenly activates the display of his device.

### 4.1.4 Training process

Our model use an instance-based learning engine. The training algorithm is a typical instanced-based algorithm where new instances are simply stored in the memory. In our case the instances are fixed size usage patterns and our target function maps them to the consumption rate of the usage patterns that follow them.

## Data preprocessing

The data stored as a huge list of measurements are not in a form that can be easily handled by our model. The construction of two data structures is essential for improving the performance of both training and prediction processes.

**Construction of the State consumption/recovery rate tables** Before making our prediction, we need to easily access the average consumption and recovery rate at each device state. The consumption rate is the percentage of battery capacity that is depleted during discharging periods while the recovery rate is defined as the percentage of battery capacity replenished when plugged in a power source. From our observations on our data we can clearly see that the rates are different between charging and discharging periods. One easily can tell that the charging process is faster than the discharging. Treating them equally will make the model not realistic. Therefore, we need two tables, one for discharging and one for charging called *State consumption rate table* and *State*

*recovery rate table* respectively. Next we discuss the construction of the former but the process is almost the same in both cases.

Firstly, we collect all measurements that have  $\sigma = \text{discharging}$ . Then, we iterate the collected measurements from the oldest to the newest looking for identical consecutive device states. Assuming that we started iterating at timestamp  $t = t_1$  having battery capacity  $c = c_1$  and observed the same device state  $d$  until  $t = t_2$  where  $c = c_2$ , we can compute the consumption rate  $r_1$  as:

$$r = \frac{|c_2 - c_1|}{t_2 - t_1} \quad (4.5)$$

Then, the device state  $d$  is mapped with the consumption rate  $r$ . We continue this process until we run out of data. If a specific device state is mapped with more than one rate, then the final rate of the device state is computed as the average of its mapped rates.

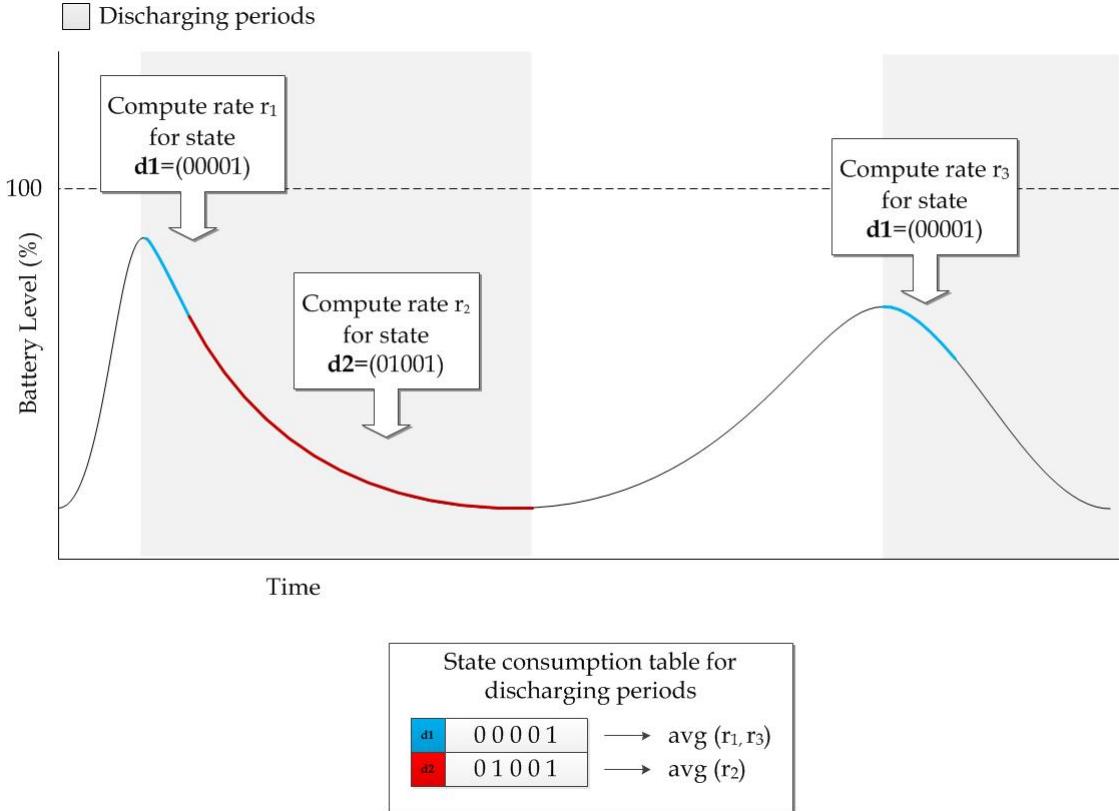


Figure 4.3: The construction of the state consumption table

An example is illustrated in Figure 4.3. We can see some fictional time series of battery level over time. The periods highlighted are the discharging periods. Suppose the blue colour is chosen for device state  $d_1 = (0,0,0,0,1)$  and red colour for device state  $d_2 = (0,1,0,0,1)$ . In the first charging period the rate  $r_1$  is mapped to  $d_1$ , while  $r_2$  is mapped to  $d_2$ . In the next charging period we compute  $r_3$  and map it to  $d_1$ . In the end we take the averages and the final table is shown in the bottom of the figure.

**Construction of the Pattern table** The next data structure that is needed for battery lifetime estimation is the pool that will hold the usage patterns of our model. The *Pattern table* is designed to be the mapping of all fixed sized usage patterns of size  $pSize$  seen so far to the average energy consumption of the usage pattern of size  $fSize$  following them. There is no obligation for the following pattern to have the same size as their predecessors. In order to distinguish the sizes we use the parameters  $pSize$  and  $fSize$  respectively. Formally:

$$(d_i, d_{i+1}, \dots, d_{i+pSize}) \rightarrow \frac{\sum_{j=1}^{fSize} cons(d_{i+pSize+j})}{fSize} \quad (4.6)$$

where,  $cons(d)$  is a function that returns the consumption rate of device state  $d$  from the state consumption table.

Figure 4.4 depicts how the pattern table can be popularised from a sequence of device states using parameters  $pSize=3$  and  $fSize=2$ . The sequential series of device states  $d_1, d_2, d_3$  of size 3 are translated into a usage pattern  $p_1$  which is mapped with the average consumption rate of the device states  $d_4, d_5$  (which actually form the following usage pattern of size 2). Next, we continue the same process for the sequential series of device states  $d_2, d_3, d_4$  acting like a sliding window until we reach the end of our device state series.

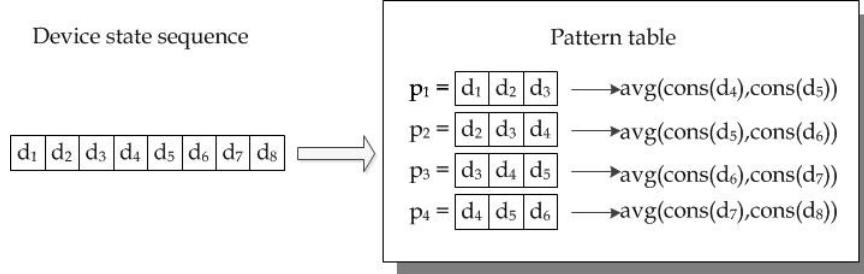


Figure 4.4: The construction of the pattern table with parameters  $pSize=3$  and  $fSize=2$

#### 4.1.5 Prediction algorithm

Since we have covered the basic blocks and the training process of our model, we can now present our prediction algorithm. The estimation is based on the distance-weighted variant of the  $k$ -NN algorithm that was discussed in detail in Chapter 2. Firstly, we compute the most recent usage pattern of size  $pSize$  from our device state sequence taken from our measurements. Having a device state sequence  $(d_1, d_2, \dots, d_n)$  the most recent usage pattern is defined as:

$$precent = (d_{n-pSize}, \dots, d_{n-2}, d_{n-1}, d_n) \quad (4.7)$$

This is the pattern that we feed into the  $k$ -NN algorithm as an input. The algorithm will return the  $k$  most similar patterns from the pattern table according to a similarity measure. Since our patterns are represented as binary data, the most suitable distance metric for our case is the Hamming distance which overcomes the others in terms of performance in binary data. As mentioned earlier the time efficient  $\oplus$  bitwise operation between two binary vectors gives the Hamming distance. After obtaining the  $k$  most similar patterns we compute the consumption rate estimation as:

$$\hat{r} = \frac{\sum_{i=1}^k w_i f(p_i)}{\sum_{i=1}^k w_i} \quad (4.8)$$

where  $f(p)$  is our target function that returns the rate of the usage pattern following  $p$  from the pattern table and the weight is defined as:

$$w_i = \frac{1}{d(x_i, x_q)^2} \quad (4.9)$$

Since we computed the consumption rate estimation, the remaining lifetime of the battery can be

computed as:

$$\hat{\lambda} = \begin{cases} (C_{max} - c)/\hat{r} & \text{if } \sigma = \text{charging} \\ c/\hat{r} & \text{if } \sigma = \text{discharging} \end{cases} \quad (4.10)$$

where  $\hat{\lambda}$  is the estimation of the remaining battery lifetime,  $\hat{r}$  is our consumption rate estimation,  $C_{max}$  is the maximum capacity of the battery,  $c$  is the current capacity of the battery and  $\sigma$  its current status.

## 4.2 Software Architecture of Open Battery

In this section we present the overview of Open Battery software tool. We discuss some details about the previous version of the software as well as its limitations and present the features that the new version was designed to have. Next, we present the architecture of the system focusing on the Android application client that was the essential part of this project.

### 4.2.1 The original version of Open Battery

This project is based on the software tool Open Battery which is a prior work of Gareth L. Jones, who is a PhD student in the AESOP research group at Imperial College London. Open Battery was released in the Google Play appstore in July 2012 and its source code is available at <https://code.google.com/p/openbattery/>.

#### Overview

The original version of the tool was developed for Android-based portable devices, while its primary aim was to collect battery related data for academic purposes and provide some statistics that can be viewed online at <http://www.openbattery.com/data-analysis.php>. The mobile client was designed to listen for changes in the battery state and log information about the state each time there is an alteration in the battery. The logged data are sent to a web server while the device is charging for further statistic analysis. Technical details of the original version of Open Battery are presented in Appendix A.

#### Limitations

Although the original version of Open Battery is functional, it has some essential limitations. The GUI of the application is minimal consisting of only two screens. Since the analysis of the data are only available on-line, redirecting to a web page in order to view the device's statistics was a rather bad design option, since certain charts cannot be viewed in the web browser, possibly because of the lack of support of Adobe Flash tool in recent Android distributions (version 4.1+).

Another limitation of the system is that the computation and the visualization of the statistics were designed to be responsibilities of the web server. As a result, statistic analysis is not applicable without an active internet connection, leading to poor user experience. Data caching is not available for off-line processing, making the on-line web page redirection the only solution.

The logged data are stored in large Comma-Separated Values (CSV) files in the local storage of the device. Opening a raw CSV file requires a Text Viewer built-in application of Android OS. Taking into account the frequency of the periodic data measurements, we can easily assume that the CSV files can be huge, thus very demanding in memory. Loading the entire CVS file into the Text Viewer may be a very slow procedure regarding the number of records that this file contains and may cause fatal memory errors.

Last but not least, the application does not support any languages other than English, which is a big disadvantage for an application distributed worldwide by the Google Play store.

#### **4.2.2 Goals and requirements**

The new Open Battery is designed to achieve specific goals and tackle the problems that the previous version had, providing new features, rich user experience and valuable feedback to the user regarding power management. Briefly, the new version of Open Battery is designed to:

1. have a creative and effective GUI that take advantage of the graphic features that Android OS provides.
2. log periodically battery and context related information and store the data in local CSV files.
3. send the logs to the Open Battery web server for academic purposes.
4. provide a cache model using a local database storage for off-line analysis on the data.
5. use a prediction model in order to make battery lifetime estimations.
6. grant battery and hardware related feedback to the user, including interesting graphs and analytics regarding device usage and battery consumption.
7. have a built-in viewer for visualization of the recent historical data.
8. provide options for customizing the application according to the users' needs.

Except from the goals that the application have to achieve, it also have to fulfill some technical requirements. According to these requirements, Open Battery should:

1. make use of the Android OS resource management mechanism in order to provide multi-language support.
2. be extensible and capable of hosting different types of prediction algorithms in the future.
3. not consume inconsiderately hardware resources such as memory, CPU and battery.
4. be able to work properly to as much Android-based devices as possible.

#### **4.2.3 Open Battery architecture and design**

In this section we give a detailed description of the architecture of the Open Battery system as well as the high level design of the new version of the Android application, discussing the reasons that made us follow these specific software design paradigms.

#### **System architecture**

The entire software system follows a Three-Tier architecture where lightweight Android collect periodically battery data and make requests to the business logic server. The business logic server sends requests to the database server in order to save the data. The architecture is shown in Figure 4.5. In this architecture, application behaviour is implemented as stored procedures running in the database. It is very important that the system can scale up the each tier as required without affecting the other. For example, adding more smartphone and tablet clients, buying a bigger database server or even clustering the database can be accomplished independently without affecting the robustness of the system.

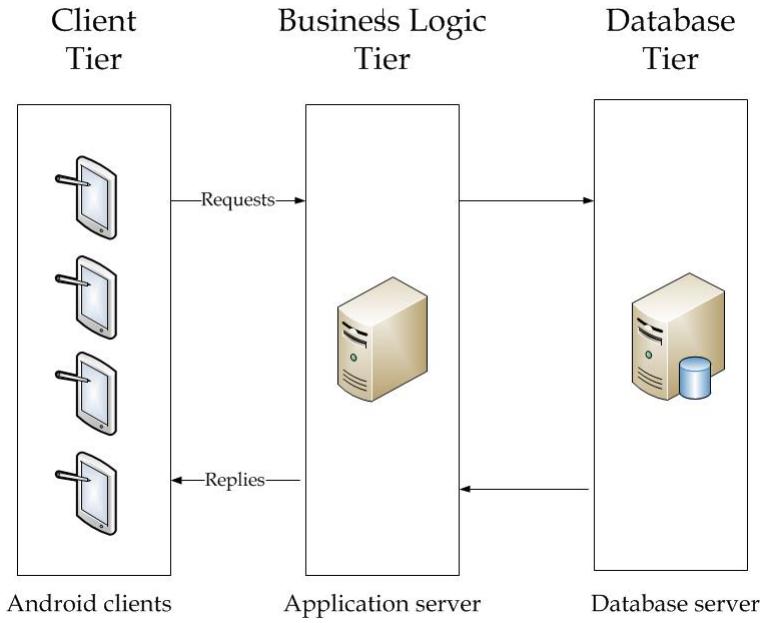


Figure 4.5: The Three-Tier software architecture of Open Battery

### Android client design

The Android client application follows the Model-View-Controller (MVC) design paradigm, firstly introduced by Trygve Reenskaug [42]. This architectural pattern organises a graphical application into three types of objects:

1. **Model** The model implements the application's logical behaviour.
2. **View** The view renders the current state of one or more model objects to the user.
3. **Controller** The controller translates input events into model commands in order to change its state.

The MVC architecture is clearly shown in Figure 4.6. This separation of responsibilities allows flexibility in large projects, as well as better and easier code maintenance. Each component is unaffected by the others, thus any future changes in specific components of the application can be applied without ruining the entire codebase. Since we wanted the code to be extensible and maintainable, MVC was definitely a suitable design pattern for our application.

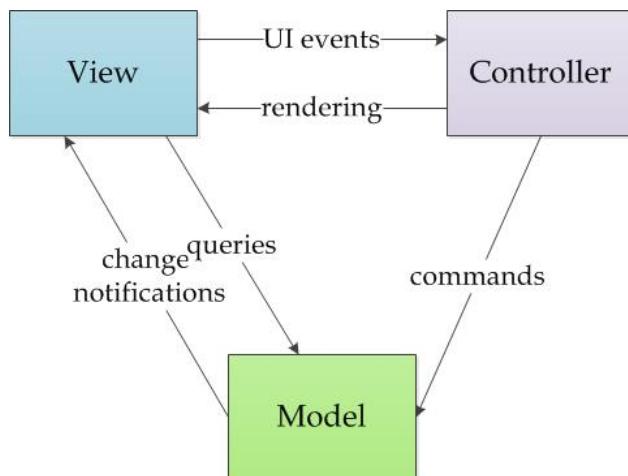


Figure 4.6: The Model-View-Controller software paradigm

In the Open Battery, the view component consists of the GUI of the application. Android OS provides a structured hierarchy of views, making the construction of these modules a simple process.

The model consists of the application data, which are the logged battery measurements, whereas the controller consists of all necessary modules responsible for mediating input and converting it to commands for the model or view. The design of the new Open Battery Android application is illustrated in Figure 4.7

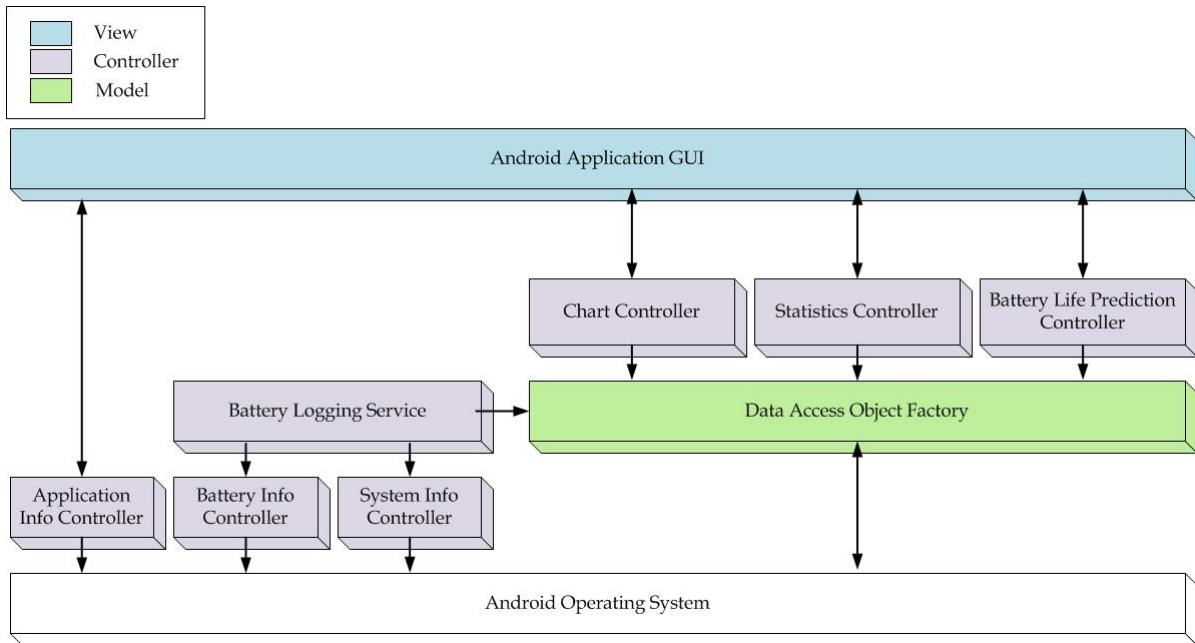


Figure 4.7: The design of Open Battery Android application

Each component shown in the figure has a specific role in the design, but interaction with the others is essential for providing the required functionality. Next we present each of these components along with a short description of its role:

**Android Application GUI** The GUI is the visual part of the application and is considered as the view entity of the MVC design. It is the only part of the application that the user interacts with and consists of all the screens of the application as well as its widgets, icons, charts and animations.

**Data Access Object Factory** This component is the model entity of our implementation of MVC paradigm. For the efficient management of our persistent data we used the Data Access Object (DAO) software pattern. According to DAO, the type of data access that application needs is separated from how these requirements are fulfilled in terms of implementation. In other words, DAOs provide some specific data operations without exposing details of the underlying storage system. Using the same interface defining read/write operations, we were able to manage different types of storage systems, such as:

1. a local SQLite database for data analysis without internet connection.
2. a set of flat CSV files for compatibility with the original Open Battery and data analysis in non-application context.

**Application Info Controller** This component is part of the controller entity of MVC design, like all the other controller components that will be presented from now on. Application Info controller is responsible for listing all processes running in of the device and sorting them according to their CPU usage. Since the gathered information is not needed for further analysis it is not saved in storage but provided directly to the GUI of the application for display purposes.

**Battery Info Controller** This controller is responsible for retrieving battery related details from the Android OS, such as the state of the battery, its level, its voltage, its temperature, its type, and many others. These technical details are essential for the prediction model of Open Battery, since they are part of the measurements used as training data of its learning engine.

**System Info Controller** This controller is responsible for retrieving system related details from the Android OS, including the device name, its model, its unique device ID, its CPU usage level, its screen state, its Wi-Fi interface state, its 3G interface state, its GPS state and many others. Part of these data are also used in the prediction model of Open Battery as the context part of the measurements forming the training data of its learning engine.

**Battery Logging Service** This service is also part of the controller entity of the application and has the responsibility of gathering the measurements of the prediction model. This service starts automatically when the device boots and takes measurements in a constant rate defined by the user. It uses the Battery Info Controller and the System Info Controller units in order to retrieve the appropriate data. Each time a measurement is taken, the service feeds the data to the DAO Factory which is responsible for data management and storage.

**Chart Controller** The Chart Controller is responsible for retrieving the appropriate data from the DAO Factory needed for displaying time series charts in the application. These include the time series of the battery level, battery temperature, battery voltage, CPU usage and others.

**Statistics Controller** This controller is responsible for making a statistic analysis of the data. It retrieves the logged measurements from the DAO Factory component and make an analysis on them. The statistics include the proportion of time that the device spend charging/discharging, the proportion of time that the device is charged by AC or USB and the frequencies of battery level at start of charging periods during the day. The results of the analysis are provided to the GUI of the application which is responsible for visualising the findings in the form of charts.

**Battery Life Prediction Controller** This controller encapsulates the functionality of the battery lifetime predictor of the system. It uses the monitored data provided by the DAO Factory and makes an estimation of the battery lifetime. The mathematical model behind the predictor was presented in detail in the previous chapter.

## 4.3 Implementation Details

In this section we discuss implementation details of the Open battery application. In the previous section we focused on the top-level architecture of the application hiding any technical details regarding each software component presented there. In this section we focus on technical details of the basic Android application components of Open Battery and our prediction model. We also discuss about our persistent data storage implementation and the mechanisms we used in order to make our tool extensible and maintainable. Everything presented here can be considered as a low-level layer of the design presented in the previous chapter. Software diagnostics of the application are presented in Appendix C.

### 4.3.1 Basic Android application components of Open Battery

We start our discussion by giving a detailed presentation of the most important basic application components of Open Battery. These components include the application's activities, widgets, services and broadcast receivers. General topics regarding these components and their role in Android application development were discussed in Chapter 2.

## Activities

Here we present the set of activities that our application has. The activities are the GUI components of the Open Battery providing the human-computer interaction. Designing the graphical user interface in portable devices and especially smartphones, should not be considered as a trivial part of application development, since a well-developed application is written to work equivalently in different display screen types and sizes. In the development stage of this project we focused on designing the GUI to be user friendly and easy to navigate by benefiting of all the GUI capabilities of the Android OS. A transition graph showing the sequence of the activities in our implementation is illustrated in Figure 4.8, while the visuals of the tool are shown in Appendix B. We continue by analysing each activity individually.

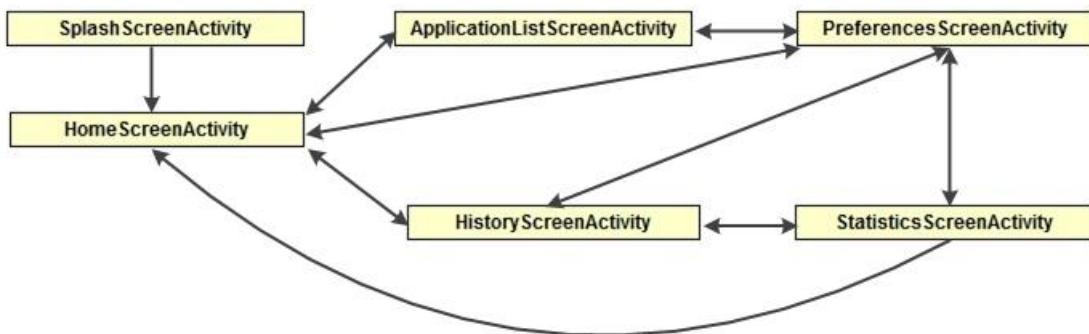


Figure 4.8: The activity sequence of Open Battery

**Splash Screen Activity** The first activity represents a simple splash screen. This is the first screen launched when the application starts. The screen shows the logo of Open Battery and is displayed for 4 seconds before the Home Screen Activity launches. However, it does not provide any further functionality and is used as an introduction screen.

**Home Screen Activity** The Home Screen Activity represents the main screen of the application. It displays information regarding the battery of the device and system details as well as interesting time-series charts. However, the size of the data needed for display is large and retrieving them from the storage can be very demanding in terms of execution time. In order to prevent the application to freeze when it retrieves the needed data, we encapsulated the process in a separate asynchronous task called `GraphDataLoadingTask` which extends the `AsyncTask` class. This way the execution of the data retrieval is performed in a separate thread without affecting the GUI thread of the application. The structure of an asynchronous task is depicted in Listing 4.1. The `AsyncTask` provides also methods for updating the GUI with progress information about the task, while the time consuming process is implemented in the `doInBackground()` method.

As mentioned earlier, this activity displays both battery and system related information. In order to separate the visuals of each category we split the GUI into two fragments hosted in a `TabHost` class. Users can navigate to each fragment by tapping each tab in the same screen. The charts displayed in both fragments are based on an open source library called `AChartEngine` [2]. This library supports fully customizable line charts, scatter charts, bar charts, pie charts, time, charts, bubble charts and many others accessible using a very well-written API.

Listing 4.1: Structure of an `AsyncTask`

```
private class MyTask extends AsyncTask<String, Void, Boolean> {
```

```

@Override
protected void onPreExecute() {
    //initialization before the execution of the task.
}
@Override
protected Boolean doInBackground(final String... args) {
    //the background work.
}
@Override
protected void onPostExecute(final Boolean success) {
    //things to be done after the execution of the task.
}
}

```

**Application List Screen Activity** This Application List Screen can only be accessed from the Home Screen and lists all the processes that are currently running on the device and shows their usage in CPU resources. Again, a separate task called `ApplicationsLoadingTask` is executed in order to avoid long waiting times before the list is ready to display. Working in the background, this task uses the `ApplicationInfoController`, responsible for obtaining the requested list from the Android OS.

Although Android platform provides an API through the `ActivityManager` for retrieving the running processes of the device along with the name of the host application, there is no support from the API regarding the retrieval of the CPU usage of each process. However, by executing the command `top -n 1` in the runtime environment through `Runtime.getRuntime().exec("top -n 1");` method invocation, we collected CPU related data including the CPU usage per process. Combining the results, we were able to construct the process list and sort it according to their CPU usage.

**History Screen Activity** The History Screen Activity is responsible for displaying the most recent measurements to the user. The measurements are visualized as a list of entries displaying the measurement's timestamp, the battery level, its voltage, its temperature and the values of the current. Before the launch of the activity, an asynchronous task called `MeasurementsLoadingTask` starts in order to retrieve the requested measurements from storage. When the measurements are loaded in memory, the list view is populated through an `ItemAdapter` which is responsible for the initialization of each entry of the list.

**Statistics Screen Activity** The role of the Statistics Screen Activity is to display the visuals of our statistic analysis. There is a total of four graphs that are displayed in separate tabs. For better user experience we used `android-support-v4` library which implements a mechanism that enables swiping between the tabs' contents. This behaviour allows horizontal swiping across the selected tab's contents for navigating to adjacent tabs, without having to directly interact with the tabs themselves.

**Preferences Screen Activity** The Preferences Screen Activity has the role of a screen of options, where the user can choose to enable some customizable features of the application. Preferences are stored as key-value mappings, where hard-coded keys representing the available preferences of the application are mapped to values representing the active preference options. For instance, the key `history_log` mapped to value `true` represents the preference of enabling history logging.

## Widgets

**The Open Battery widget** The widget of the application is a view that can be placed anywhere in the home screen and displays the battery's level, the battery's temperature as well as an

estimation of when the battery will be full or empty, according to its status. The technical specifications of the widget, such as size and update rate are defined in an XML file called `widget_info` in `res/xml` direcory. After the widget is created, it executes a service called Update Service in order to update the data displayed in the view. More details regarding the application's services are discussed next.

## Services

The Open Battery application uses two services that run in the background. Their execution is designed to be transparent to the user. Here we present some technical implementation details for each one of them.

**Update Service** This service is executed from the Open Battery widget's content provider implemented in the `ApplicationWidgetProvider` class and is responsible for updating the data displayed in the view. The service make use of a broadcast receiver listening to the `ACTION_BATTERY_CHANGED` system event. This event is sent by the Android OS when the status of the battery changes. When such changes occur, the service uses the Battery Info Controller to retrieve new battery data and the Battery Life Prediction Controller to get an estimation of the remaining battery duration. The Battery Life Prediction Controller uses the concrete implementation of our prediction model described in the next subsection. When the necessary data become available from the controllers, it displays the results. The entire process is illustrated in Figure 4.9.

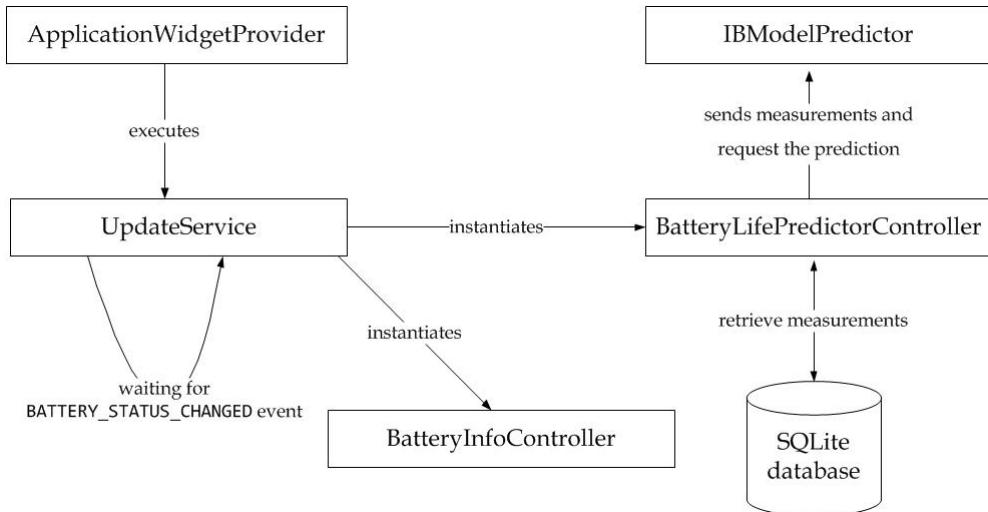


Figure 4.9: The widget updating process.

**Battery Logging Service** The Battery Logging Service is responsible for taking a single measurement and save it locally. Moreover, when a certain amount of traces is collected, this service prepares the data in order to send them to the Open Battery server. Each time a measurement is taken, the service spawns a new thread for doing the appropriate background work. We chose this implementation in order to avoid blocking the main GUI thread of the application when a new measurement is taken. The service is designed to start when the device boots or a new version of the application is installed. The Android application component that executes this service when the device boots is the Boot Broadcast Receiver which is described next.

## Broadcast receivers

**Boot Broadcast Receiver** This component listens for `BOOT_COMPLETED` and `PACKAGE_REPLACED` system actions. The former is sent when the device boots, while the latter when an application is installed or updated. When such an action is received by the Boot Broadcast Receiver, an new

instance of `AlarmManager` is created. This class provides access to the system alarm services and is responsible for scheduling future events. In our implementation the `AlarmManager` instantiates and executes a Battery Logging Service periodically in its `setRepeating()` method.

### 4.3.2 Prediction Model

In this section we describe the implementation of our prediction model. The abstract functionality of the prediction system is defined in an interface called `Predictor` shown in Listing 4.2. The method `train()` feeds the engine with the necessary training data while the method `computeRemainingBatteryLife()` returns the battery's remaining lifetime estimation measured in seconds. The concrete implementation of the instance-based engine is defined in a class called `IBModelPredictor` which implements the interface. Our design leaves room for other prediction models, since a new model has to implement only these two abstract methods in order to be used in Open Battery application.

**Listing 4.2: The Predictor interface**

```
public interface Predictor {
    //the training process.
    public void train(List<Measurement> measurements);
        //estimates the remaining battery lifetime in seconds.
    public Long computeRemainingBatteryLife(BatteryStatus status, BatteryLevel level);
}
```

In the concrete implementation of our prediction model, the `train()` method stores the list of measurements taken as a parameter into the main memory and process it in order to populate the State consumption/recover rate table and the Pattern table. The former is computed in the method `computeDeviceStateRateTable()`, while the latter in the `computePatternTable()` method. For the computation of the pattern table, two parameters are needed:

- `PATTERN_WINDOW_SIZE`: The size of each usage pattern. These patterns are candidates for similarity matching with the most recent one and they act as keys in the pattern table.
- `FUTURE_PATTERN_WINDOW_SIZE`: The size of each chronologically adjacent pattern. Their consumption rates are used as values in the Pattern table.

The implementation of the prediction algorithm is defined in `computeRemainingBatteryLife()` method. Firstly, the most recent pattern of size `PATTERN_WINDOW_SIZE` is computed in the method called `computeLastPattern()`. Then, this pattern is compared with all the others in the pattern table and the closest  $k$  are returned from the `KNNalgorithm()` method which implements the  $k$ -NN search algorithm. All patterns of the pattern table are stored in a priority queue and first  $k$  are selected, according to the similarity measure of the algorithm. We used the Hamming distance as the similarity measure, since it has the same results as the others for binary vectors and its computation time is low comparing it to the computation time of other metrics. The last stage of the algorithm defined in method `computeDistanceWeightedKNN()` is the calculation of the consumption rate's estimation based on the distance-weighted  $k$ -NN property.

### 4.3.3 Persistent data management

The persistent data of Open Battery are the logged measurements. Any other data, such as the active process list and the CPU usage of each process are not saved since they are not used in our prediction model and they are not considered important. Open Battery supports both local and remote storage systems.

#### Local storage

We provide two mechanisms for local data storing; an SQLite database and CSV flat files. We describe each one of them next.

measurements	
 <b>id</b>	text
<b>timestamp</b>	text
<b>level</b>	text
<b>scale</b>	text
<b>voltage</b>	text
<b>temperature</b>	text
<b>plugged</b>	text
<b>status</b>	text
<b>health</b>	text
<b>current</b>	text
<b>cpu</b>	text
<b>screen</b>	text
<b>wifi</b>	text
<b>cellular</b>	text
<b>gps</b>	text

Figure 4.10: The SQLite local database

**SQLite Database** The measurements of Open Battery are saved locally in an SQLite Database. The schema of the database consists of a single table as illustrated in Figure 4.10, while each record in the table consists of 15 attributes in text format. A brief description of each attribute is provided next:

- **id**: The key of the table used to uniquely identify measurements.
- **timestamp**: The exact time when the measurement was taken.
- **level**: The current level of battery capacity measured in %.
- **scale**: The maximum battery level capacity measured in %.
- **voltage**: The current battery voltage measured in mV.
- **temperature**: The current battery temperature measured in C.
- **plugged**: Attribute indicating whether the device is plugged in to a power source.
- **status**: Attribute indicating the status of the battery. For instance, it may be charging, discharging or full.
- **health**: Attribute indicating the condition of the battery.
- **current**: The current that is received from the power source measured in mA.
- **cpu**: The level of CPU usage measured in %.
- **screen**: The state of the display screen.
- **wifi**: The state of the Wi-Fi interface.
- **3G**: The state of the 3G cellular interface.
- **gps**: The state of the GPS system.

The database plays the role of a local cache which holds the last 10,000 logged measurements. After reaching this limit, each time a new measurement is taken the oldest one is deleted from storage, acting like a sliding window.

**CSV flat files** The history of the traces are also saved in a CSV file called `battery_log.csv` in the internal storage of the device. The exact location of the file is `\OpenBattery\`. Each entry of these files contains a user trace with the exact attributes that a record in the database has. The entries are separated by a line break while the attributes are separated by a ';' delimiter character. The data saved as CSV files can be processed by other software suites such as Matlab and Microsoft Office.

### Remote storage

The data collected from Open Battery are sent to the web server which connects to a PostgreSQL database. The data are sent using an Hypertext Transfer Protocol (HTTP) client, defined in `HttpDataSender` class. The system provides automatic device registration the first time a connection between the Android client and the web server is established and periodical data submission of blocks consisting of 100 measurements. The PostgreSQL database schema is shown in Figure 4.11.

batterydata_devicedetails	
deviceid	text
devicemodel	text
technology	text
designcapacity	text
manufacturer	text

battery_data	
deviceid	text
timestamp	text
level	text
pluggedid	text
stateid	text
currentcapacity	text
voltage	text
temperature	text
healthid	text
timezone	text
cpu	text
screenid	text
wifiid	text
cellularid	text
gpsid	text

Figure 4.11: The PostgreSQL remote database

#### 4.3.4 Software extensibility

Open Battery follows specific software design patterns in order to be extensible. In this section we discuss how new features can be added without affecting and breaking the existing codebase.

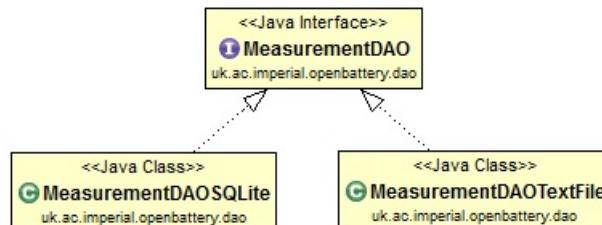


Figure 4.12: The DAO implementation of two different types of storage

## Adding new types of storage

In our implementation we make use the DAO design pattern as discussed in the previous chapter. Since we consider only measurements as the persistent data of Open Battery, we created an interface called `MeasurementDAO`, which defines all the operations that our abstract storage should perform regarding this type of object. The interface is shown in Listing 4.3.

Any type of storage should implement this interface and provide a concrete implementation of any of these methods needed in the client program, although it is suggested to implement all of them for completeness reasons. The classes already implementing the interface are shown in the UML diagram of Figure 4.12. Each type of `MeasurementDAO` is instantiated by the corresponding `DAOFactory`, which is defined as an abstract class which produces DAO objects. When a new type of storage is added, a new DAO class should implement the `MeasurementDAO` interface and define a factory subclass of `DAOFactory` responsible for instantiating the specific DAO object.

Listing 4.3: The `MeasurementDAO` interface

```
public interface MeasurementDAO {  
    //adds a new measurement to storage.  
    public void addMeasurement(Measurement measurement);  
    //returns a measurement from storage.  
    public Measurement getMeasurement(int id);  
    //returns the last num measurements.  
    public List<Measurement> getMeasurements(int num);  
    //returns all the measurements.  
    public List<Measurement> getAllMeasurements();  
    //returns the number of the logged measurements.  
    public int getMeasurementsCount();  
    //updates a measurement.  
    public int updateMeasurement(Measurement measurement);  
    //deletes a measurement.  
    public void deleteMeasurement(Measurement measurement);  
    //deletes all measurements.  
    public void deleteAllMeasurements();  
}
```

## Adding new subsystem states

Open Battery provides also a mechanism for adding new subsystem states. The functionality of the state is defined in an interface called `State`. All existing subsystems states implement this interface. When a new state is introduced, it should also implement the `State` interface and should be added as a field in the `DeviceState` class.

## Adding new prediction models

A new prediction model can be added easily in Open Battery. The class representing the new model just has to implement the interface `Predictor` and provide a concrete implementation of the methods defined there.

## Multi-language support

The new Open Battery tool provides multi-language support. The application is translated in English, Greek, French and German already by specifying the entire set of strings used in the application in four different XML files, one for each language. A new translation can be added by simply creating a new XML file called `strings.xml` containing all strings translated in the new language and storing it to the `res/values-{language-code}` folder.

## 4.4 Summary

In this chapter we gave a detailed presentation of the prediction model that we developed. We started by discussing the limitations of other similar studies and giving an overview of our approach. We modeled a portable device state as set of the binary states of its subsystems and a usage pattern as a sequence of consecutive device states. The model uses an instanced-based learning engine which is capable of recognizing them. The engine chooses the most similar patterns to the most recent one using the distance-weighted  $k$ -NN algorithm and estimates the battery's remaining lifetime based on the selected patterns.

In this chapter we also presented the Open Battery tool. We started by describing the limitations of the previous version of the application and setting the goals that the updated version is planned to achieve. Then, we presented the top-level architecture of the entire system which follows the Three-Tier paradigm and focused on the software design of the Android client which follows the Model-View-Controller pattern. The view consists of all the GUI components of the application, the model includes all the data that are collected periodically, while the controllers translate GUI event into commands to the model. In the last part of the chapter we gave a brief description of each component of the design.

Moreover, we presented some important implementation details of the new version of Open Battery. We focused on the Android application components of the tool and the technical details of the prediction model we developed. We showed which operations a prediction model should implement and how our model provided the given functionality. We also presented our storage system, consisting of two types of storage, while in the last part of the chapter, we discussed how new features can be added to the application.

Next we present our evaluation part of this thesis, presenting a statistical analysis of real data from two Android-based devices.



# Chapter 5

## Evaluation and Discussion

In this chapter we evaluate our work both quantitatively and qualitatively using real data. We start by presenting our dataset and how we worked on this in order to get valuable results for our system. Next, we perform a statistical analysis on our data and we show which usage patterns are discovered. In the next two sections we discuss the performance of our prediction model in terms of accuracy and computation time as well as the resource consumption of Open Battery application itself. In the last part of this chapter we discuss the limitations of our approach.

### 5.1 Mobile data analysis

#### 5.1.1 Dataset details

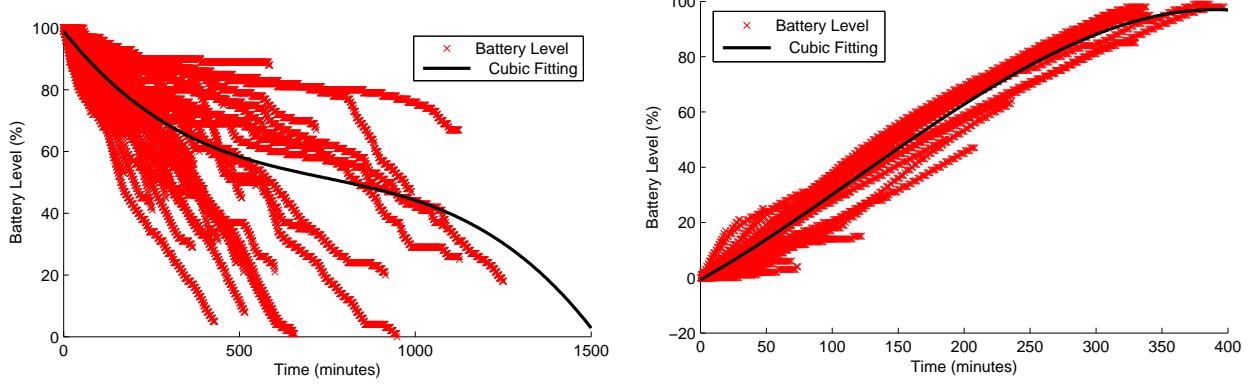
Our dataset consists of 70,000 measurements taken from real Android-based devices, a Samsung I9300 Galaxy S3 smartphone and an Acer Iconia A1-810 tablet device, both running Android 4.1 platform. The Galaxy S3 device has a Quad-core 1.4 GHz Cortex-A9 CPU and uses a 2100mAh battery, while the Iconia tablet has a Quad-core 1.2 GHz CPU and uses a 4960mAh battery. From now on, the users of these devices will be referred to as User 1 and User 2 respectively. The duration of data logging was approximately one month, from July 19th 2013 to August 26th 2013. Data were collected periodically in 1 minute intervals and include battery related information, such as battery level, voltage and temperature as well as system related information such as CPU level and the states of the device's subsystems as described in Chapter 4. We have to point out here that due to the limited amount of Android devices that we had in our possession and the limited time that this project lasted, it was not possible to get a larger dataset for our analysis.

#### 5.1.2 Analysis results

Our analysis on the data consists of three parts. In the first of we process our time-series data and convert them into an appropriate form for evaluating a regression model. Next, we provide some useful statistics on how users interact with their devices, recognizing specific usage patterns. Our last part of our analysis also focuses on identifying patterns in usage behaviour after merging the device states into larger clusters.

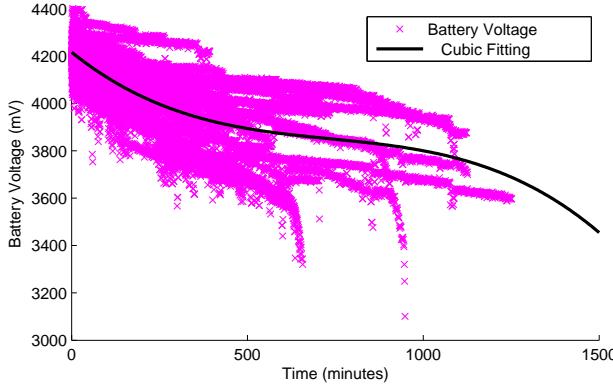
#### Regression curve fitting

Regression analysis is a very useful process for estimating correlations among a set of variables. We use this model in our evaluation to find relationships between time and battery properties such as level, voltage and temperature. Before starting our regression analysis we decomposed the dataset into two segments, including measurements of discharging and charging periods respectively. Firstly, we took into consideration our battery related data and tried to investigate the relationship between them and time, using a cubic fit regression model. Our original data were measurements logged as time-series, thus we had to convert them to in a way to be processed using relative time.

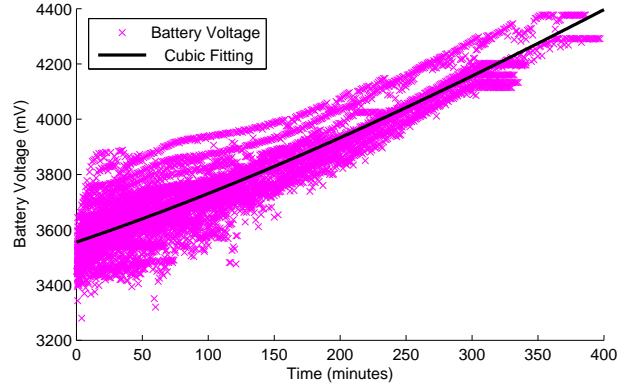


(a) Battery Level vs Time during discharging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = -7.157e - 008$ ,  $p_2 = 0.00016033$ ,  $p_3 = 0.14358$  and  $p_4 = 99.015$ . The norm of residuals is 2490.9

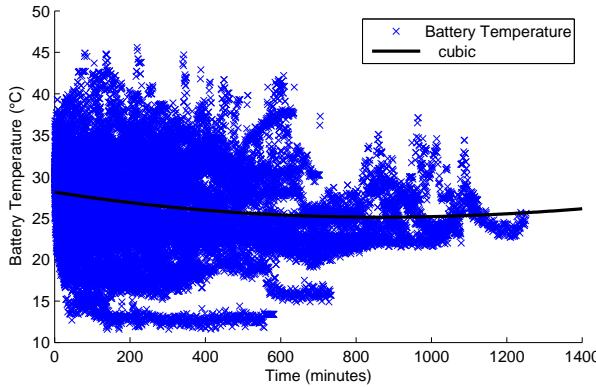
(b) Battery Level vs Time during charging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = -1.514e - 006$ ,  $p_2 = 0.00054164$ ,  $p_3 = 0.26982$  and  $p_4 = -0.76093$ . The norm of residuals is 437.74



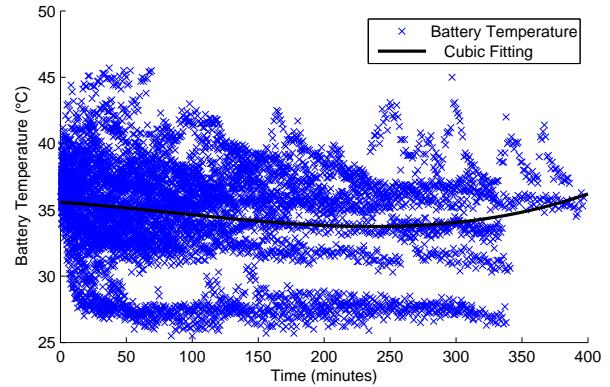
(c) Battery Voltage vs Time during discharging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = -6.3929e - 007$ ,  $p_2 = 0.00014154$ ,  $p_3 = -1.1923$  and  $p_4 = 4216.2$ . The norm of residuals is 19702



(d) Battery Voltage vs Time during charging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = -7.6705e - 007$ ,  $p_2 = 0.0015346$ ,  $p_3 = 1.6123$  and  $p_4 = 3555.2$ . The norm of residuals is 6722.8



(e) Battery Temperature vs Time during discharging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = -3.4265e - 010$ ,  $p_2 = 4.636e - 006$ ,  $p_3 = -0.0072507$  and  $p_4 = 28.144$ . The norm of residuals is 936.71



(f) Battery Temperature vs Time during charging periods. Here we show the cubic fitting of our data with coefficients  $p_1 = 1.5036e - 007$ ,  $p_2 = -3.8577e - 005$ ,  $p_3 = -0.0070454$  and  $p_4 = 35.575$ . The norm of residuals is 316.44

Figure 5.1: Our dataset regarding battery consumption

For the battery level data of the discharging segment we started from relative time  $t_0 = 0$  and maximum battery level  $l_0 = 100$ . Then, we iterated the data until a discharging period was over at time  $t_1$ . At this point we reseted the time to  $t_0$  and repeated the same procedure. A tricky point in this process is that we are not guaranteed that the next discharging period will start at  $l_0 = 100$ . For instance, a user could have stopped charging his device when it reached level 45%. We wanted to have each discharging period starting from the same point. In order to tackle this problem we considered also relative values in the  $x$  axis for computing the regression curve. By relative values we mean that when a new discharging period started we also reseted the battery level to  $l_0 = 100$  and then computed the relative differences from the real values. For instance, the battery level trace (45, 45, 44, 43, 43, 42) was converted to (100, 100, 99, 99, 99, 98). When a new period started we also reseted the voltage level as well as the temperature level and we did the same processing for the charging segment of data.

Our results using cubic curve fitting are shown in Figure 5.1. As we can clearly see, battery level and voltage values tend to decrease while the battery is discharging, while they increase during charging periods. This behaviour reflects the non-ideal battery properties discussed in Chapter 2. The fitted curves shown in black colour can be used for the prediction of battery's lifetime. From the same figure we can see that the time needed for a full cycle last approximately 1900 minutes, 1500 for the charge process and 400 for the discharge process. The difference between the duration of the two processes indicates that the consumption and recovery rates of the battery are considerably different. That is the main reason why we used one rate table for each battery status in our prediction model. However, the battery temperature values do not follow the same pattern and they cannot be used for predicting the battery performance over time.

### Statistics identifying usage patterns

In this part of our analysis on the data collected from Open Battery tool, we try to observe how users interact with their devices and identify some usage patterns. This analysis focuses on validating our original hypothesis that each user has different usage characteristics which can be used for personalized services on power management. Figure 5.2 shows the proportion of time that each device spent in charging, discharging and full states. As we can see from the figure, User 1 leaves his device charging almost six times more than User 2, while both users use their devices powered with full battery almost the same amount of time. These results show that the usage behaviour and the charging habits of these two users are slightly different. .

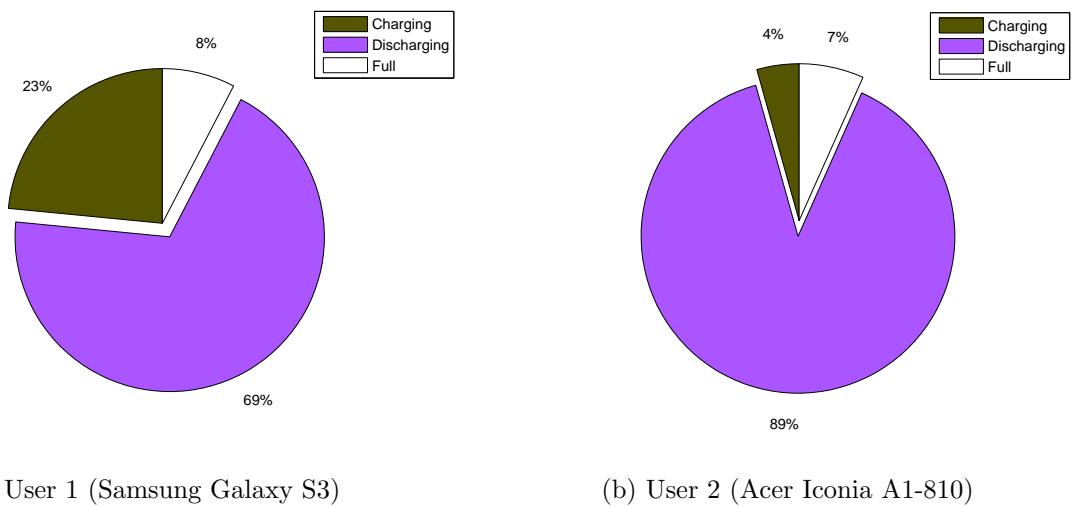


Figure 5.2: Proportion of time spent charging/discharging

The next pair of charts in Figure 5.3 shows a comparison of the power sources that both users use. We consider USB and AC power sources, where USB provides slower charge than normal AC chargers. Generally, USB is preferred for short charges, while AC for regular long charges. Here

the difference between the charging preferences of the two users is clear. User 1 mostly charges his device using USB while User 2 uses regular chargers almost always. The proportion of time charging his device using USB is less than 1%. If we combine these results with our previous ones we can assume that User 1 spends so much time charging his smartphone because he uses USB very frequently. We can also assume that User 1 is frequently near a computer where a USB power source is available, while User 2 works in a place where an AC power source is reachable or connects his device to an AC power source while he sleeps at night.

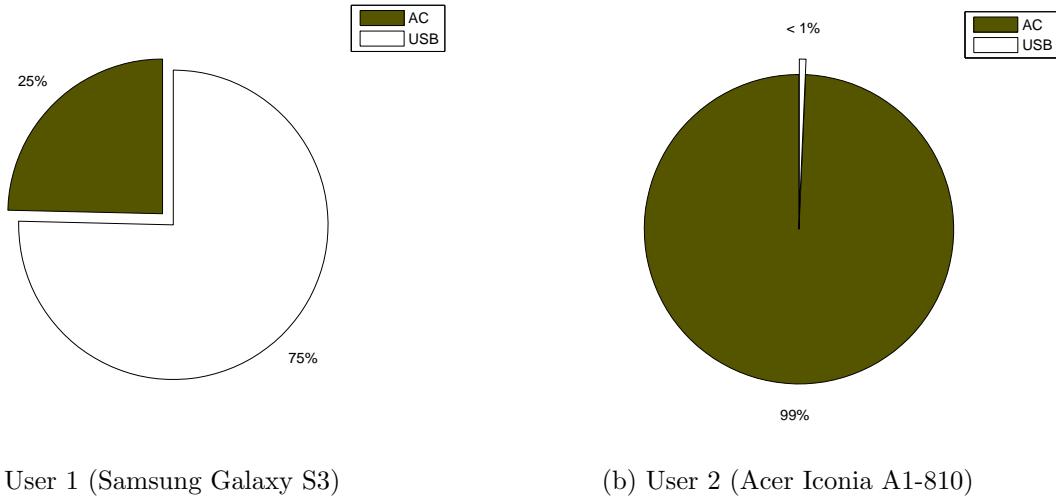


Figure 5.3: How the device is being charged

The next pair of charts in Figure 5.4 shows the battery level at the moment when charging begins. User 1 usually charges his device when its battery level is lower than 40%, while User 2 prefers to charge his tablet when its battery level ranges between 10 % and 20%. More specifically, User 1 chooses to charge his device when its battery level reaches the lowest limit of 0%, while User 2 seems to be more conservative. Before charging his device he ensures that it has at least 10% of battery level.

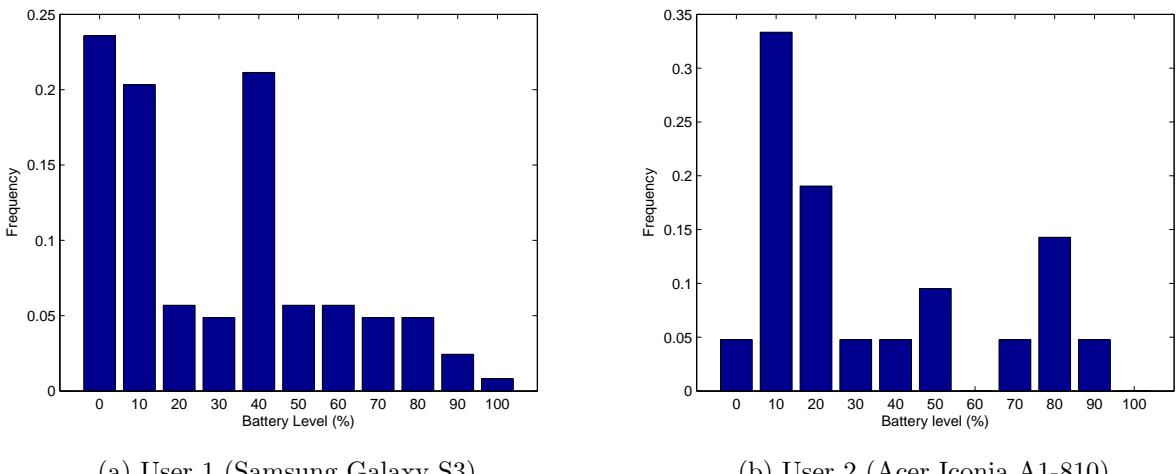


Figure 5.4: Battery level at start of charging periods

Our next charts illustrated in Figure 5.5 show when the two users charge their devices during the day as well as the battery level at start of charging periods. Each bubble represents the start of a charging process. The visualization consists of three dimensions; battery level, time and frequency. Frequency is highlighted by both size and colour, where small yellow bubbles indicate low frequencies while big red ones indicate high frequencies. As we can see in Figure 5.5a, User 1 charges his device more frequently than User 2 and he prefers charging it between 15:00 and 21:00, with

a majority initiating charging when battery level is 40%. On the other hand User 2 charges his device less frequently and especially between 00:00 and 05:00.

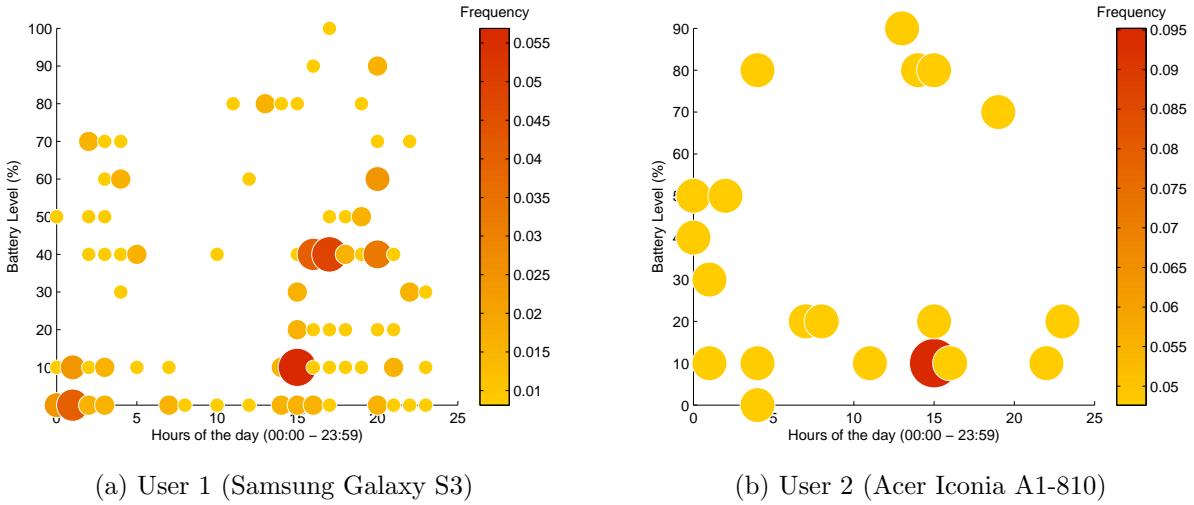


Figure 5.5: Battery level during the day at start of charging periods

From our analysis we observed that both users have different charging preferences and their usage behaviour varies a lot. User 1 charges his smartphone more frequently by USB short charging processes in the evening, while User 2 prefers charging his tablet rarely using an AC charger at night. Since we identified some charging characteristics from our data, it would be very interesting to know the most frequent device states when the devices are charging and discharging. Figure 5.6 illustrates the distribution of the device states. States with frequency lower than 1% are considered very rare and are not shown. The device state (00100) is the most frequent during both periods. This is expected, since users leave their Wi-Fi open very often, even though the screen of their device is off. Moreover, we can see that the second more frequent state during discharging is (01100) while the second most frequent during charging is (00000). This shows that users interact with their devices when the battery is discharging but they do not use it when they plug it in a power source. We have to point out here that some device states had zero frequency, meaning that the devices used in our experiments have never been in those states. An example of a state that has never been logged is (11111), where all subsystems of the device are activated and the CPU is overloaded.

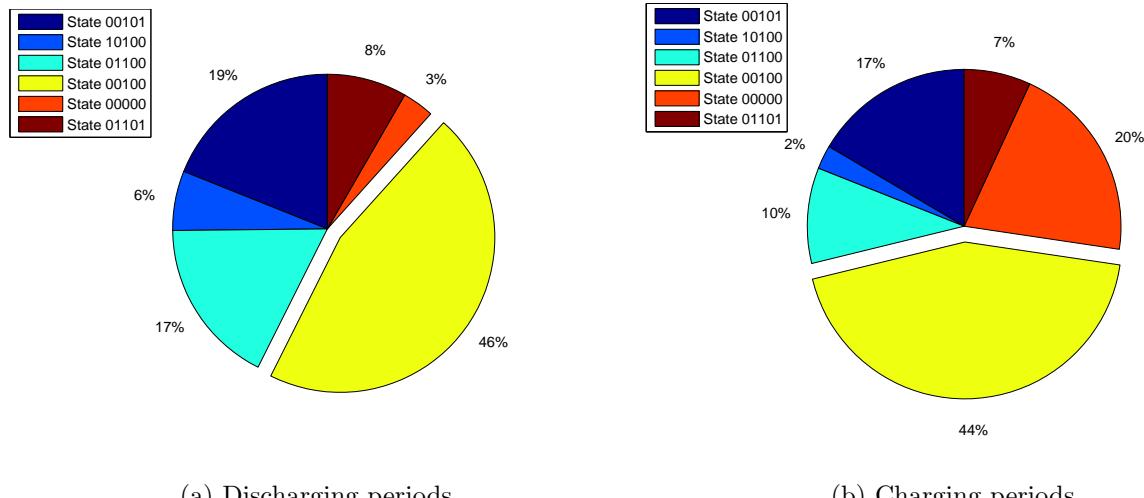


Figure 5.6: Frequencies of device states during charging and discharging periods

Our analysis on the data collected from two different users shows that each user has different usage

behaviour and follow specific usage patterns, as we expected. We mainly focused on charging characteristics. There are significant differences on which types of power sources are used for charging and which parts of the day users prefer to charge their devices. There is also difference on which subsystems of the device are enabled more frequently during charging and recharging periods. Next, we focus on how users interact with their devices in terms of the usage of the features these devices offer. In order to simplify our model, we merged the device states of our model into larger sets of operational states.

### Merging device states

Our model takes into consideration 5 subsystem states modelled as binary variables. The state space of our approach consists of  $2^5 = 32$  individual device states. As mentioned before, some of them were never reached according to our data. We can tackle this problem and simplify our model by reducing the number of possible states. We can minimize the complexity by merging some states in order to model the real operational states of a portable smartphone or tablet device. As shown in Table 5.1, we created four operational states, IDLE, WIFI, 3G and GPS. We also consider an ETC state consisting of all device states that do not fall to any merged state category.

<b>IDLE</b>	<b>WIFI</b>	<b>3G</b>	<b>GPS</b>
00000	00100	00010	00001
01000	01100	01010	01001
10000	10100	10010	10001
10000	11100	11010	11001

Table 5.1: The merged operational states of a portable device

Figure 5.7 illustrates a state transition diagram of a portable device after merging the possible device states. When the device boots, it stays in the IDLE state. When a user enables the Wi-Fi, the current state changes to WIFI and when he disables it, the current state returns back to state IDLE. When a user enables data communication using 3G, the current state changes to 3G and after disabling it, the current state returns back to the state IDLE. Finally, when a user enables the GPS system, the current state changes to GPS state and returns back to IDLE when GPS is disabled.

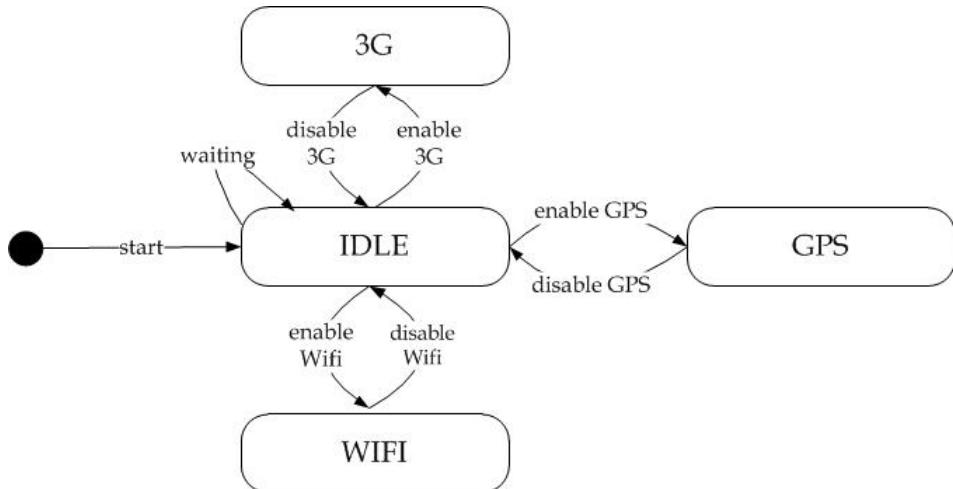


Figure 5.7: A simple transition diagram with merged device states

Now that the state space is limited, it is easier to identify patterns regarding the usage of the device. In Figure 5.8 we show the time spent in each merged state. According to our results both users leave their devices into the IDLE state almost the same amount of time. However, User 1 seems to be a heavy Wi-Fi user who rarely uses 3G for data transmission, while User 2 almost never uses data communication. Moreover, there is a big difference between these two users on the way they

use the GPS system of their devices. User 1 spends almost 10% of time using GPS in contrast to User 2 who spends more than 60% of time interacting with applications that use the GPS system.

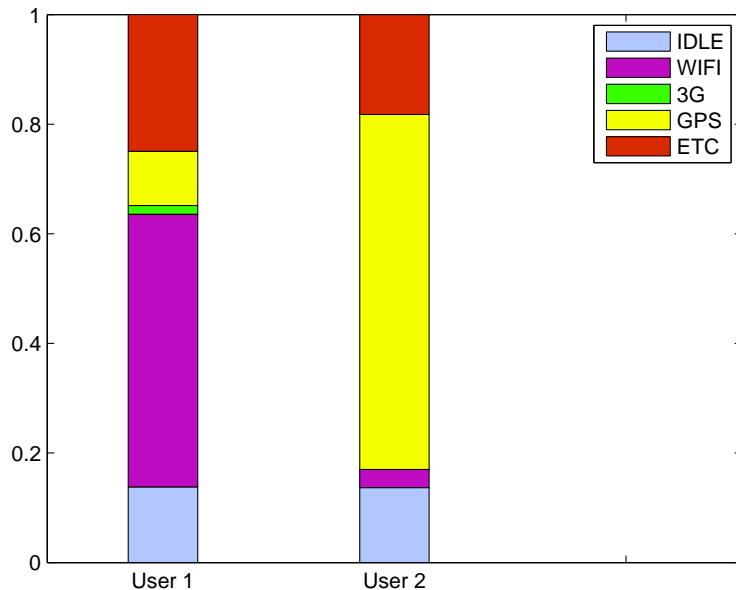


Figure 5.8: Spent time in each merged operational state

Our results show clearly the magnitude of the different usage characteristics that two individual users have. User 1 probably spends his time browsing the web or using applications that require internet connection while he is somewhere where Wi-Fi is available, like his residence or maybe at work. However, the behaviour of User 2 is completely different. He almost never interacts with applications which require internet access and uses a lot location-based services that the device provides. We may assume that he is probably a professional driver who uses his tablet for map services while he is working, where internet connection is not available.

## 5.2 Prediction model Evaluation

In this section we evaluate our instance-based learning engine. Our basic concerns are the prediction model's accuracy and its performance.

### 5.2.1 Accuracy

For our experiments we took into consideration the traces logged from User 1 consisting of 45,000 measurements and used the 10-cross validation method for observing the error. We used the nine tenths of the dataset as the training set and the one tenth as the test set. Since our traces are logged from real users who charge their devices randomly, they do not include complete cycles of discharging and charging periods. Thus, we predict when the next 1% of battery depletion or recovery will occur and we measure the error as the average of the absolute difference between the actual duration and our prediction. In our experiments we consider four parameters:

- **Parameter  $k$**  : The number of the nearest neighbours that  $k$ -NN algorithm returns.
- **Dataset size**: The number of measurements in the dataset.
- **Parameter  $pSize$**  : The size of each usage pattern used in the  $k$ -NN algorithm.
- **Parameter  $fSize$**  : The size of each chronologically adjacent pattern.

We did three experiments in total to measure the accuracy of our model. In each experiment we tested different parameters while some of them were set as constants, as described in Table 5.2.

	Parameter $k$	Dataset size	Parameter $pSize$	Parameter $fSize$
<b>Experiment 1</b>	1-40	45,000	10	5
<b>Experiment 2</b>	5,30	500-45,000	10	5
<b>Experiment 3</b>	30	45,000	1-15	1-15

Table 5.2: Parameters used for our experiments

### Experiment 1

Firstly, we compare the accuracy of the  $k$ -NN and the distance-weighted  $k$ -NN algorithms for various values of parameter  $k$ . In this experiment we set  $pSize$  and  $fSize$  as constants with values 10 and 5 respectively. Our results are illustrated in Figure 5.9. As we can see, our model has an error of approximately 9 seconds per 1% battery dissipation (or recovery), which is satisfying for a battery estimation according to related studies. Both algorithms perform almost the same for  $k$  values lower than 7. However, the distance-weighted  $k$ -NN variant performs better when  $k > 25$  and has a difference of 0.1 seconds with the basic algorithm. We also see that the minimum error values of the basic  $k$ -NN are observed using  $k$  values between 1 and 10, while the distance-weighted variant performs better with  $k$  values between 20 and 40. This happens because the basic algorithms do not calculate weights for closest neighbours and the target function values are averaged. Thus, when the number of neighbours becomes bigger it take into consideration equally close neighbours and neighbours that may have large distance from the query pattern as well. On the contrary, the distance-weighted  $k$ -NN need a lot of neighbours to perform well, since it weighs the contribution of each neighbour according to their distance from the query pattern.

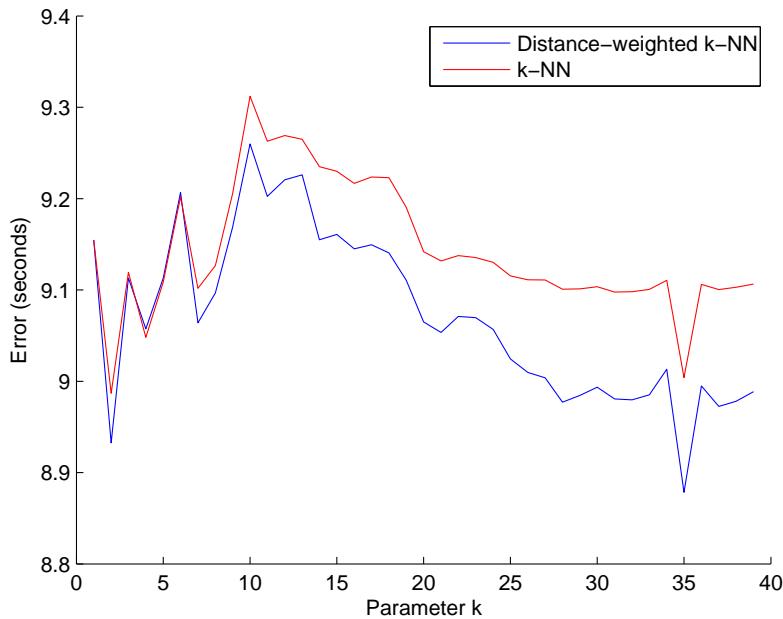


Figure 5.9: Error vs parameter  $k$  for  $k$ -NN and distance-weighted  $k$ -NN algorithms

### Experiment 2

In this experiment, we show how both algorithms perform when different datasets are used. In this experiment we took again  $pSize$  and  $fSize$  as a constant with values 10 and 5 respectively and we ran the algorithms for different random subsets of the original dataset. Our results for  $k = 5$  and  $k = 30$  are shown in Figures 5.10a and 5.10b respectively.

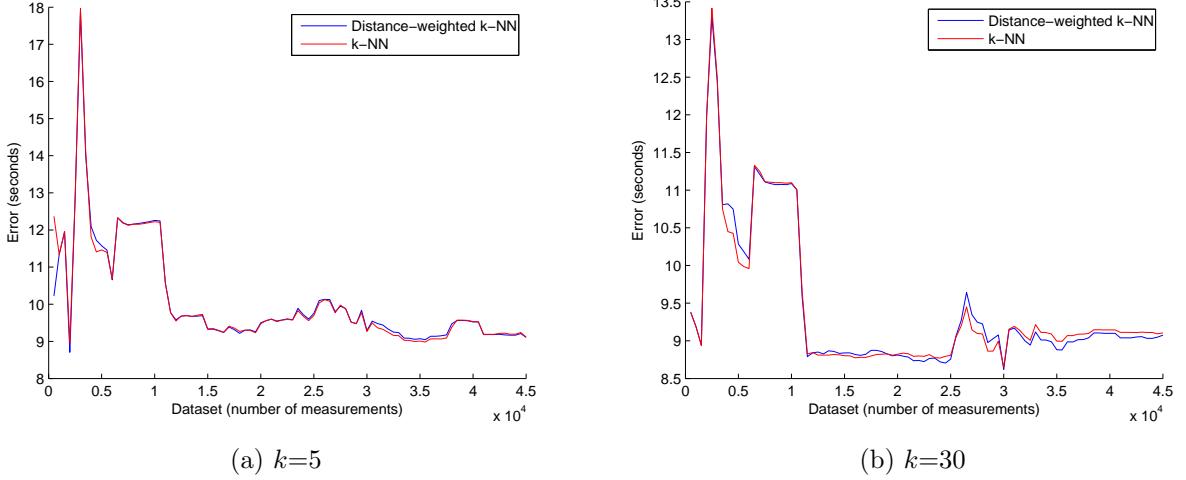


Figure 5.10: Error vs Size of dataset

As we can clearly see, the behaviour of the two algorithms is similar for both values of  $k$ . When few training examples are used, the error is high, reaching 18 seconds when  $k = 5$  and 13.3 seconds when  $k = 30$ . It is also remarkable that the curves have volatile behaviour when the datasets used consist of less than 10,000 measurements. However, the behaviour of the curves tend to stabilize when more than 10,000 measurements are used as training data, reaching an average error ranged from 9 to 10 seconds. Therefore, 10,000 measurements are considered enough for an accurate estimation of our model. A larger set of data will be a waste of space without significant differences in results.

### Experiment 3

In our last experiment we show which are the optimal values of  $pSize$  and  $fSize$  in terms of minimizing the error. We used the entire dataset of 45,000 measurements taken in 1 minute intervals and we tested the distance-weighted  $k$ -NN algorithms for several values of  $pSize$  and  $fSize$ . Our results are illustrated in Figure 5.11. Each square of the diagram represents a combination of  $pSize$  and  $fSize$ . The darker an area is, the lower is the error for this combination of parameters. As we can see from the figure, the error is high for  $pSize < 5$  regardless the values of  $fSize$ , reaching approximately 11 seconds per 1% of battery dissipation (or recovery). When higher values for parameter  $pSize$  are used, the error drops to almost 9 seconds. The combination of high values of  $pSize$  and low values of  $fSize$  seems to be the optimal as shown in the dark upper left corner of the diagram. This is reasonable, since we predict the next 1% drop or recovery of the battery and our prediction is short-term. The parameter  $fSize$  represents the size of the patterns which follow the ones that are returned from the  $k$ -NN search algorithm. The size is considered as a metric of the time duration of a pattern. More specifically, it denotes the time spent in the device states that form it. Patterns having  $fSize = 3$  last 2 minutes and represent the future horizon of the estimation. Thus, they are most accurate for short-term predictions.

#### 5.2.2 Performance

Our model is designed to work in lightweight portable devices, thus its time efficiency is a topic of major concern. Therefore, the next thing we tested is the time complexity of the two processes of our learning engine, the training process and the prediction process. In this experiment we used many different subsets of our original dataset. Our time measurements were taken from the Samsung Galaxy S3 smartphone device using the system instruction `System.nanoTime()`. Our results are illustrated in Figure 5.12.

In Figure 5.12a we can see how the duration of the training process oscillates when different

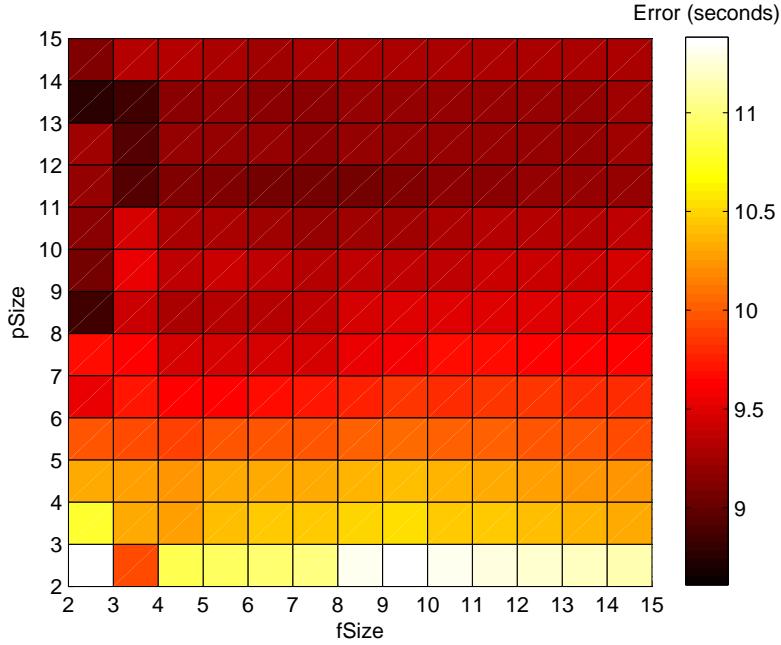


Figure 5.11: The error for different values of  $pSize$  and  $fSize$

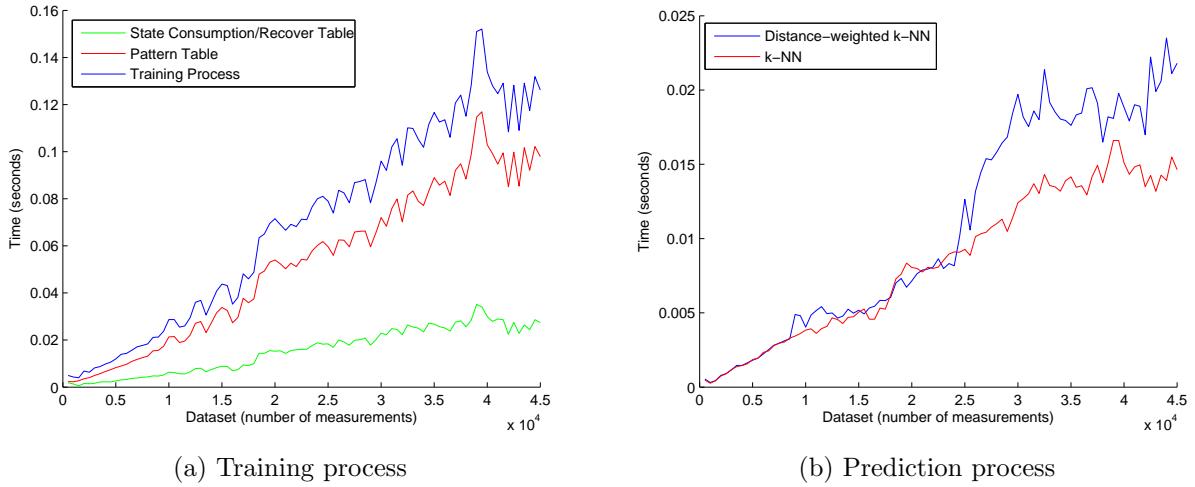


Figure 5.12: Time complexity of the instance-based learning engine

sized datasets are used for training. Moreover, we also see how long it takes to construct the State consumption/recovery tables and the Pattern table, which are the basic operations of the training process. As expected, the duration of constructing these tables as well as the duration of the entire training process increases when larger datasets are used. The training process using a small dataset of 10,000 measurements requires almost 0.03 seconds, while using a larger set of 45,000 traces requires approximately 0.12 seconds, showing a linear increase in duration. We can also see from the same figure that the construction of the Pattern table is the most time-consuming operation of the training process and grows rapidly when larger datasets are used. On the contrary, the construction of the State consumption/recovery tables has lower requirements in time, having a peak of only 0.03 seconds when 40,000 measurements are used.

Figure 5.12b shows the duration of the prediction process. For our experiments we used both  $k$ -NN and distance-weighted  $k$ -NN algorithms and measured their time complexity using datasets of various sizes. We used the same value for parameter  $k = 30$  when we tested the performance of the two algorithms. According to our results, the time needed for the termination of both algo-

rithms increases as the datasets used become larger. Both algorithms perform almost equally for small datasets, containing less than 20,000 traces. However, the time requirements of the distance-weighted version of the  $k$ -NN algorithm increase steeply when datasets containing more than 25,000 measurements are used having a peak in 0.022. This happens because of the calculations of weights that the algorithm performs. On the other hand, the basic  $k$ -NN algorithm performs better in terms of time, having a peak in 0.017 seconds when 40,000 measurements are processed. Although the basic  $k$ -NN terminates faster, the difference is not noticeable practically and does not overcome the superiority of the accuracy of the distance-weighted  $k$ -NN algorithm.

### 5.3 Software Evaluation

In this section we evaluate our software tool regarding its consumption in hardware resources, such as battery dissipation, CPU usage and memory consumption. It is very important for a power management tool not to be a high consumer itself. In order to calculate the dissipation in resources, we took measurements using the Dalvik Debug Monitor Server (DDMS) tool of Android OS in the Samsung Galaxy S3 device.

Firstly, we consider battery dissipation. The most power hungry feature of Open Battery is the battery logger which takes measurements in a constant rate. We expect that the battery consumption will decrease if the logging time intervals become larger. Our chart in Figure 5.13 shows exactly that. The battery consumption of the tool is approximately 21% when the logger takes measurements every 1 minute. The consumption almost halves when the rate quintuples, while the lowest battery dissipation is approximately 8% where the time interval between two consecutive measurements is 10 minutes.

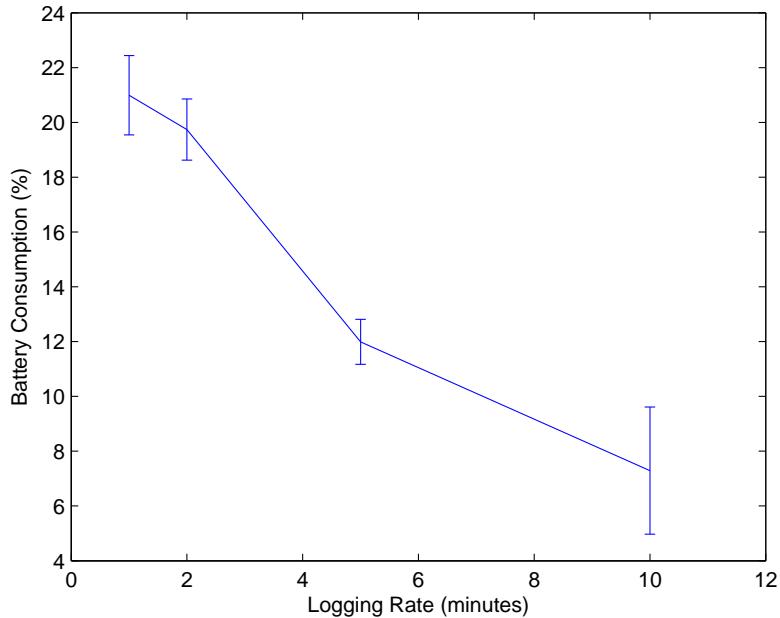


Figure 5.13: Battery consumption vs Logging rate

The CPU usage follows the same pattern as the battery consumption. As shown in Figure 5.14, the CPU level is almost 4.5% when the logger takes measurements every minute. The CPU drops slightly to 4% when the logging rate doubles and decreases even more to 1.6% when the logging rate is 5 minutes. However, when the logging rate changes from 5 to 10 minutes there is not any significant difference in CPU usage. Therefore, a logging rate of 5 minutes is a very good option for the implementation of Open Battery logger, since it does not overload the CPU and is frequent enough.

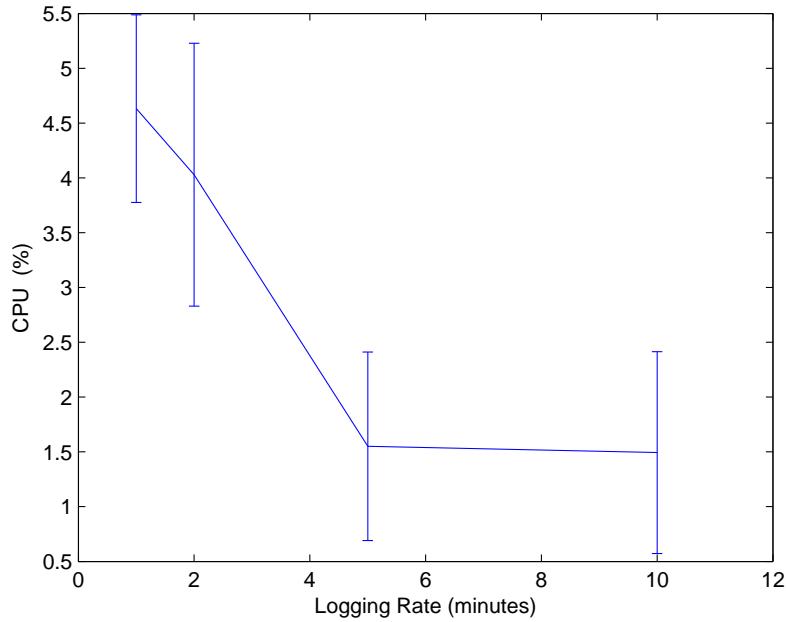


Figure 5.14: CPU usage vs Logging rate

Our last experiment focuses on calculating the heap size allocated for Open Battery application for different sizes of datasets. When the application does not retrieve any data from the local database, it uses a heap of 13.1 MB. For a dataset containing 500 measurements the Open Battery application uses 13.6 MB of RAM in average and for 1000 measurements the memory requirements increase to 15 MB of RAM. From this point the memory requirements increase almost linearly, reaching nearly 18 MB when Open Battery uses a dataset of 3000 measurements. In general, the memory requirements are not very high, but using a larger dataset may cause memory allocation exceptions in older devices running previous versions of Android OS.

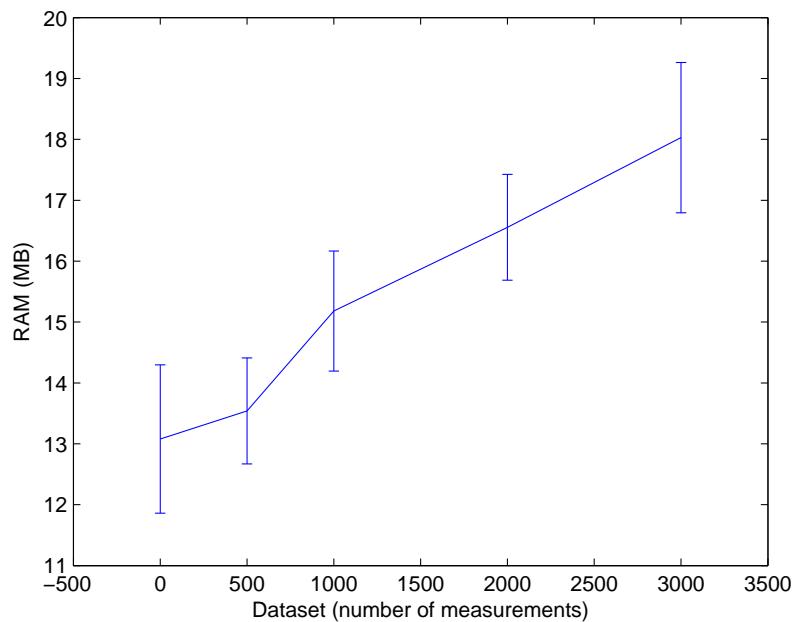


Figure 5.15: Memory heap vs Dataset size

## 5.4 Limitations and areas of improvement

In our analysis we used 70,000 measurements taken from two Android-based devices during approximately one month. Although through our analysis we draw some reasonable conclusions and we evaluated our initial hypotheses regarding usage patterns, our dataset is considered small compared to the datasets presented in other studies. As we discussed earlier, many device states were not reached at all for both devices and we did not have enough information to compute their consumption and recovery rates. The lack of data is due to the fact that there was not sufficient time and resources in devices for a large-scale analysis. However, the new version of Open Battery will be released soon in Google Play appstore and we will be able to collect a lot of data in the Open Battery servers from a huge variety of smartphone and tablet devices.

The general performance of Open Battery is very satisfying according to our results when the analysis was performed off-line but it has some limitations under specific circumstances. Our prediction model requires a lot of data in order to perform accurately. As we observed the more data it has, the more accurately it predicts the remaining battery lifetime. However, there is always the trade-off between accuracy and complexity. From our results, the computation time of creating the pattern table is the most time-consuming procedure of the training process, even using a small dataset of 10,000 measurements. The construction of this table is performed, just before the predictor estimates the battery lifetime, resulting in delays in the Open Battery widget where the estimation is displayed. A feasible solution could be the infrequent computation of the table in a constant rate. For example, the table could be populated daily and saved in the local database as persistent data.

## 5.5 Summary

In this chapter we presented the evaluation of our work, using real data collected from two Android-based devices. We used a regression model on our dataset and measured the difference between charging and discharging durations. According to our results, discharging lasts almost four times more than charging. We also presented some useful statistics and showed how two users may have completely different charging and usage preferences, validating our original hypothesis on usage patterns. In the next part of this chapter we did experiments using our prediction model and calculated the error of our predictions which is approximately 9 seconds for each 1% of battery consumption (or recovery). We also compared the  $k$ -NN and the distance-weighted  $k$ -NN algorithms in terms of accuracy and duration. Our results show that the distance-weighted  $k$ -NN algorithm is more accurate but terminates later than the basic  $k$ -NN. Moreover, we tested the Open Battery application using different parameters and measured its requirements in hardware resources. In the last part of the chapter we discussed the limitations of our work and suggested some possible solutions.

Next we conclude this thesis and present some ideas for further work on Open Battery.



# Chapter 6

## Conclusions and Future Work

### 6.1 Conclusions

Smartphones and tablet computers are two of the most popular and fastest growing types of portable devices in the modern mobile networks. They are powered from batteries that have limited size and capacity. Therefore, the optimal power management of these devices is a major research topic and there are many tools in the market that present diagnostics and provide estimations of the remaining battery lifetime. However, most of them predict the battery dissipation using statically defined profiles and do not take into consideration usage characteristics.

In this thesis we presented a lightweight battery lifetime prediction model that uses an intelligent learning engine to make estimations recognizing usage patterns. Our original hypothesis was that we can recognize these patterns by analysing users' activity and charging behaviour on smartphones and tablets. Therefore, both battery and context information is needed to train the engine. We modelled the subsystems of the device as binary variables having two states, ON and OFF. The device state was defined as a set of the binary states of its subsystems and the usage pattern as a sequence of chronologically adjacent device states. The core of our model is a learning engine which is trained with fixed-size usage patterns and uses the distance-weighted  $k$ -NN algorithm in order to obtain the most similar patterns to the most recent one. Using the consumption rate of these patterns, our model estimates the remaining battery lifetime. The training process is fast and space efficient, thus is suitable for lightweight devices, such as smartphones and tablets. Moreover, the model adapts to the user's behaviour and is not based on statically defined consumption rates.

We also redesigned from scratch an application for Android devices called Open Battery. The application monitors battery and system related data and provides statistics regarding the usage of the device. It stores the collected data in a database which acts as a local cache and also exports them as CSV files in the file system of the device for further off-line analysis. Moreover, the data are sent periodically to a remote server for academic purposes. Open Battery uses our prediction model in order to estimate the future lifetime of the battery which is displayed in the widget of the application. Although our prediction model fits our goals, the tool was designed to be extensible and host new algorithms for prediction in the future. We also showed that the application has limited consumption in hardware resources and it can be lowered even more by adjusting the parameters of the logging system.

The last part of our project was the statistical analysis of data from real Android-based devices. By analysing the usage patterns of 70,000 traces logged in a two-month period from one smartphone and one tablet computer, we showed that users have different behaviour on how they use their devices and that the operational states of the handhelds have significant variation in consumption rates, validating our initial hypothesis.

## 6.2 Further Work on Open Battery project

The work presented in this thesis has potential for many future improvements. One option is to improve the existing prediction model. The Open Battery application can be easily adapted to collect any kind of context information from the Android OS, such as audio, video and telephony states. However, the choice of the number of the subsystems used in the prediction model should be considered carefully to avoid constructing a large state space and make the system fine-grained. Moreover, more elegant data structures can be used for storing the pattern instances of the distance-weighted  $k$ -NN algorithm. A suitable choice is a  $k$ -dimensional tree which is a geometric data structure used for organizing points in a  $k$ -dimensional space [5]. The nearest neighbour search of our prediction model can be performed efficiently by using the  $k$ -dimensional tree's properties in order to eliminate large segments of the search space.

It would be also very interesting to investigate how other prediction models perform using the data collected from Open Battery tool. Since the device states are already defined in our work, another model can use them in order to make estimations of battery's lifetime. A good option might be a Hidden Markov Model, where similar states could be merged in order to decrease the state space. The adaptation of other prediction models in Open Battery is a simple process, since it is designed to be able to make use of any new prediction algorithm without breaking its functionality.

Last but not least, a large-scale analysis could be made using the data reported in the remote server of Open Battery, hoping that using a larger dataset of different types of portable devices could provide a more concrete validation of this work. More pattern distributions could be shown as well as prediction results and comparisons between more users.

# Bibliography

- [1] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.
- [2] AChartEngine. Open source chart engine for android os. <http://www.achartengine.org/>.
- [3] Viswanathan Arunachalam, Vandana Gupta, and S Dharmaraja. A fluid queue modulated by two independent birth-death processes. *Computers & Mathematics with Applications*, 60(8):2433–2444, 2010.
- [4] Allen J Bard and Larry R Faulkner. *Electrochemical methods: fundamentals and applications*, volume 2. Wiley New York, 1980.
- [5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [6] Carla F Chiasserini and Ramesh R Rao. A model for battery pulsed discharge with recovery effect. In *Wireless Communications and Networking Conference, 1999. WCNC. 1999 IEEE*, pages 636–639. IEEE, 1999.
- [7] Carla F Chiasserini and Ramesh R Rao. Pulsed battery discharge in communication devices. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 88–95. ACM, 1999.
- [8] Carla F Chiasserini and Ramesh R Rao. Energy efficient battery management. *Selected Areas in Communications, IEEE Journal on*, 19(7):1235–1245, 2001.
- [9] Carla F Chiasserini and Ramesh R Rao. Improving battery performance by using traffic shaping techniques. *IEEE Journal on Selected Areas in Communications*, 19(7):1385–1394, 2001.
- [10] Todd L Cignetti, Kirill Komarov, and Carla Schlatter Ellis. Energy estimation tools for the palm. In *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 96–103. ACM, 2000.
- [11] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.
- [12] Open Data Commons Public Domain Dedication and License. <http://opendatacommons.org/licenses/pddl/>.
- [13] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.
- [14] R Gaeta, M Gribaudo, D Manini, and M Sereno. Analysis of resource transfers in peer-to-peer file sharing applications using fluid models. *Performance Evaluation*, 63(3):149–174, 2006.

- [15] Nicolas Hohn, Darryl Veitch, Konstantina Papagiannaki, and Christophe Diot. Bridging router performance and queuing theory. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 355–366. ACM, 2004.
- [16] Google Inc. Android developers dashboards. <http://developer.android.com/about/dashboards/index.html>.
- [17] Google Inc. Android system architecture. <http://developer.android.com/images/system-architecture.jpg>.
- [18] Google Inc. Google i/o 2013. <https://developers.google.com/events/io/>.
- [19] Bradley A Johnson and Ralph E White. Characterization of commercially available lithium-ion batteries. *Journal of power sources*, 70(1):48–54, 1998.
- [20] Gareth L. Jones and Peter G. Harrison. Collecting battery data with Open Battery. In *2012 Imperial College Computing Student Workshop*.
- [21] Gareth L Jones, Peter G Harrison, Uli Harder, and Anthony J Field. Fluid queue models of renewable energy storage. In *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on*, pages 224–225. IEEE, 2012.
- [22] Gareth L Jones, Peter G Harrison, Uli Harder, and Tony Field. Fluid queue models of battery life. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 278–285. IEEE, 2011.
- [23] MR Jongerden and BR Haverkort. Battery modeling. 2008.
- [24] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 353–362. ACM, 2012.
- [25] Joon-Myung Kang, Sin-seok Seo, and JW Hong. Personalized battery lifetime prediction for mobile devices based on usage patterns. *Journal of Computing Science and Engineering*, 5(4):338–345, 2011.
- [26] Joon-Myung Kang, Sin-seok Seo, and JW-K Hong. Usage pattern analysis of smartphones. In *Network Operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific*, pages 1–8. IEEE, 2011.
- [27] Chandra Krintz, Ye Wen, and Rich Wolski. Application-level prediction of battery dissipation. In *Low Power Electronics and Design, 2004. ISLPED’04. Proceedings of the 2004 International Symposium on*, pages 224–229. IEEE, 2004.
- [28] David Linden and Thomas B Reddy. Handbook of batteries. *New York*, 2002.
- [29] Ingrid Lunden. Android, led by samsung, continues to storm the smartphone market, pushing a global 70<http://techcrunch.com/2013/07/01/android-led-by-samsung-continues-to-storm-the-smartphone-market-pushng-a-global-70-market-share/?ncid=tcdaily>.
- [30] James F Manwell and Jon G McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399–405, 1993.
- [31] James F Manwell and Jon G McGowan. Extension of the kinetic battery model for wind/hybrid power systems. In *Proceedings of EWEC*, pages 284–289, 1994.
- [32] JF Manwell, JG McGowan, EI Baring-Gould, W Stein, and A Leotta. Evaluation of battery models for wind/hybrid power system simulation. In *Proceedings of EWEC*, 1994.

- [33] Thomas L Martin and Daniel P Siewiorek. *Balancing batteries, power, and performance: system issues in cpu speed setting for mobile computing*. PhD thesis, PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.
- [34] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.
- [35] PAP Moran. A probability theory of dams and storage systems. *Aust. Jour. App. Sci.*, 5:116–124, 1954.
- [36] Earl A Oliver and Srinivasan Keshav. An empirical approach to smartphone energy level prediction. In *Proceedings of the 13th international conference on Ubiquitous computing*, pages 345–354. ACM, 2011.
- [37] PowerTutor. <http://powertutor.org>.
- [38] Ahmad Rahmati, Angela Qian, and Lin Zhong. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, pages 265–272. ACM, 2007.
- [39] Daler N Rakhmatov and Sarma BK Vrudhula. An analytical high-level battery model for use in energy management of portable electronic systems. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 488–493. IEEE Press, 2001.
- [40] Venkat Rao, Gaurav Singhal, Anshul Kumar, and Nicolas Navet. Battery model for embedded systems. In *VLSI Design, 2005. 18th International Conference on*, pages 105–110. IEEE, 2005.
- [41] Nishkam Ravi, James Scott, Lu Han, and Liviu Iftode. Context-aware battery management for mobile phones. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 224–233. IEEE, 2008.
- [42] Trygve Reenskaug. Thing-model-view-editor an example from a planningsystem. *Xerox PARC technical note*, 1979.
- [43] Peng Rong and Massoud Pedram. An analytical model for predicting the remaining battery capacity of lithium-ion batteries. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5):441–451, 2006.
- [44] Anthony Wood, Jack Stankovic, Gilles Virone, Leo Selavo, Zhimin He, Qiuhua Cao, Thao Doan, Yafeng Wu, Lei Fang, and Radu Stoleru. Context-aware wireless sensor networks for assisted living and residential monitoring. *Network, IEEE*, 22(4):26–33, 2008.
- [45] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.



# Appendices



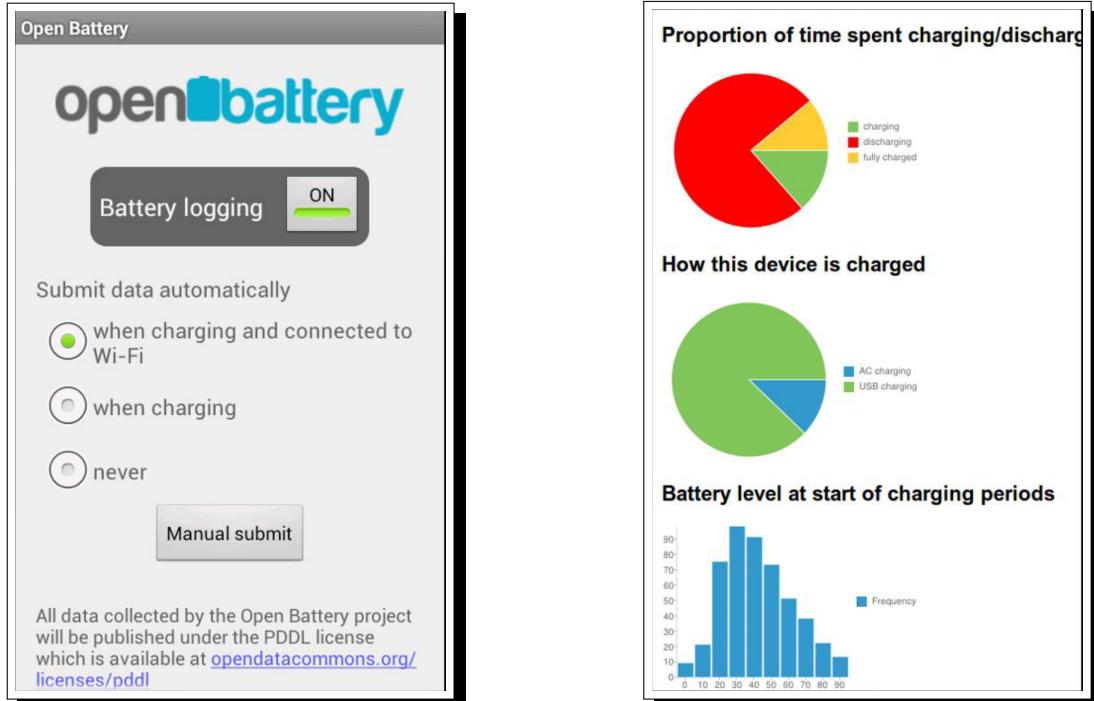
## Appendix A

# Technical details of the original Open Battery Android application

For completeness reasons, in this chapter we present some technical details of the original Open Battery Android client, developed by Gareth L. Jones.

### A.1 Graphical User Interface

The GUI of the application is illustrated in Figure A.1. It is minimal and consists of two screens, the home screen and the statistics screen as a redirected web view of the <http://www.openbattery.com> website. In the home screen there is a toggle button enabling or disabling the battery logging. Options for submitting data automatically are displayed also in the home screen. Access to the log file and the statistics screen is provided by an application menu.



(a) The Home Screen of the original Open Battery (b) The Statistics Screen of the original Open Battery

Figure A.1: The GUI of the original Open Battery application

## A.2 UML class diagram

In this section we present a UML class diagram of the application. The original Open Battery application does not follow a specific architectural pattern and consists of only five classes hosted in the same package `uk.ac.imperial.doc.aesop.openbattery` as shown in Figure A.2. The `BootUpReceiver` is responsible for starting a new service when the device boots. The service is implemented in `OpenBatteryService` class and performs periodic measure loggings. The implementation of a log is written in the `Log` class. Finally, the `OpenBattery` class is the main activity of the application and `submitDataTask` is an inner class responsible for the data submission to the Open Battery web server.

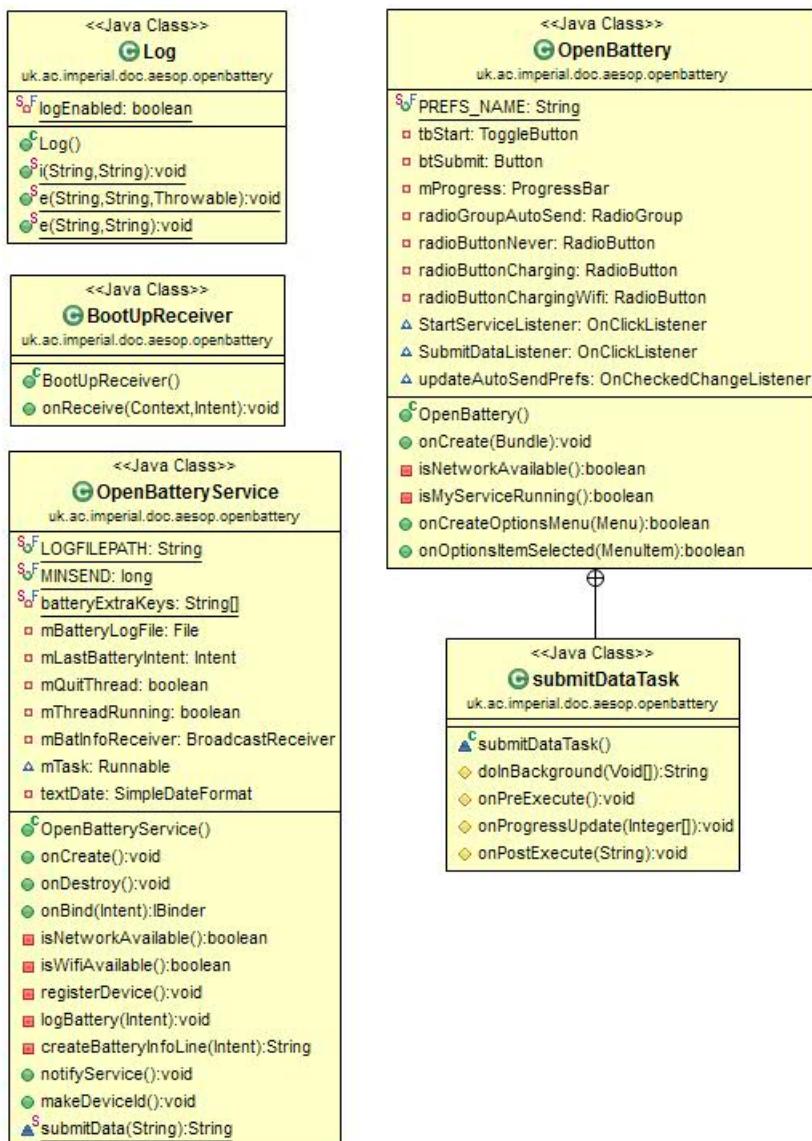


Figure A.2: UML diagram of the original Open Battery Android application

# Appendix B

# Open Battery User Guide

## B.1 Installation

Open Battery application has been tested in Android OS versions 2.1+. The installation file is called `OpenBattery.apk`. In order to install it from the Android Debug Bridge (ADB) using a laptop or a PC, the following command has to be issued:

```
adb install OpenBattery.apk
```

Alternatively, the installation can be performed by simply copying the `.apk` file into an Android-based device and locate it using the local File Manager. The automatic installation process begins after tapping the `OpenBattery.apk` file icon and selecting the Install option.

## B.2 Open Battery Screens

In this section we present the main screens of Open Battery.

### Splash Screen

When the Open Battery application launches, the splash screen is displayed, as shown in Figure B.1. This screen does not provide any functionality and is used as an introduction screen.

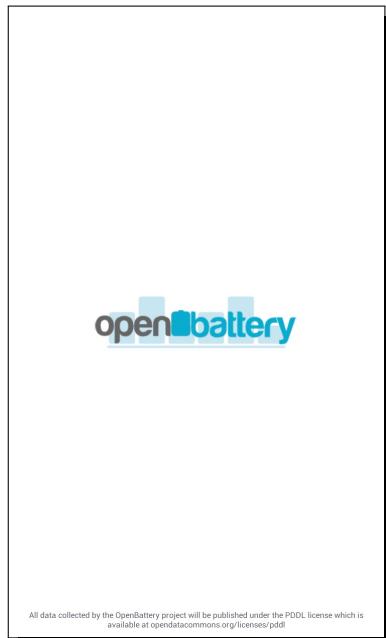
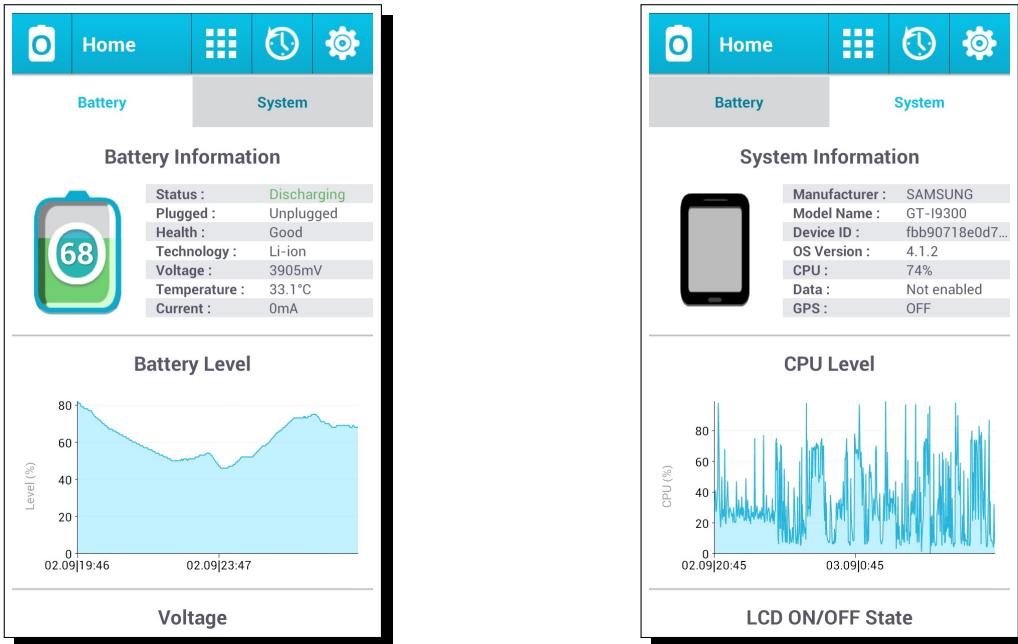


Figure B.1: The Splash Screen of Open Battery

## Home Screen

This is the home screen of the application. It consists of two tabs implemented as separate GUI fragments as shown in Figure B.2. These tabs show battery and system related information as well as interesting time-series graphs.

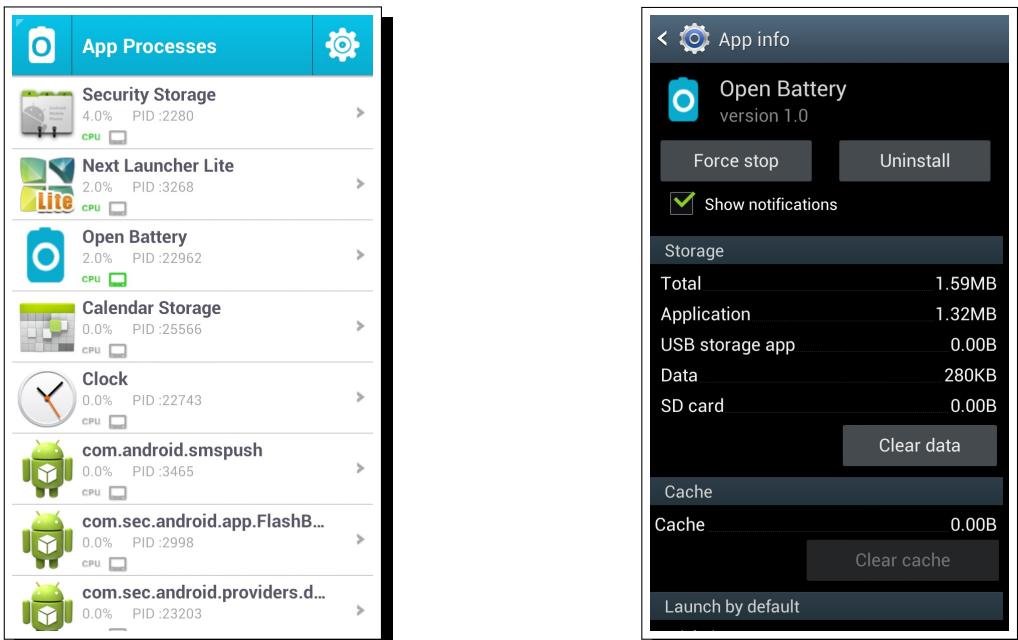


(a) The battery information fragment

(b) The system information fragment

Figure B.2: The Home Screen of Open Battery

Navigation across the screens is provided through the blue action bar on top of the screen. The button in the left top corner returns the application to the home screen, while the other three buttons redirect the user to other screens of the application. More specifically, the user may navigate to the Application Process Screen, the History Screen and the Preferences Screen.



(a) The list of running processes

(b) The Application Manager

Figure B.3: The Application Process Screen of Open Battery

## Application Process Screen

In this screen the user can see which processes are running in his device, as well as some relevant information, such as the PID of each process and its usage in CPU resources, as illustrated in Figure B.3a. By tapping an entry of the process list, the application redirects to the Application Manager of Android OS, where the user can see more detailed information regarding the process and terminate it, as shown in Figure B.3b.

## History Screen

In this screen the most recent logged battery related measurements are listed, in reversed chronological order. A measurement consists of its timestamp, the battery level, its voltage, its temperature and the current of the power source used for charging, as shown in Figure B.4. The user can see older measurements by scrolling down the list.

Timestamp	(%)	mV	°C	mA
2013-09-03 03:45:50	67	3855	36.3	0
2013-09-03 03:44:54	68	3905	33.1	0
2013-09-03 03:43:50	68	3905	33.1	0
2013-09-03 03:42:50	68	3926	33.0	0
2013-09-03 03:41:50	68	3911	33.4	0
2013-09-03 03:40:50	68	3911	33.4	0
2013-09-03 03:39:51	68	3911	33.4	0
2013-09-03 03:38:50	68	3901	33.8	0
2013-09-03 03:37:50	68	3922	33.7	0
2013-09-03 03:36:50	69	3916	34.1	0
2013-09-03 03:35:50	69	3930	35.5	0
2013-09-03 03:35:44	69	3930	35.5	0
2013-09-03 03:34:50	69	4000	37.7	466
2013-09-03 03:33:50	68	4007	37.3	466
2013-09-03 03:32:50	68	4006	37.4	466
2013-09-03 03:31:50	68	3995	38.8	466
2013-09-03 03:30:50	68	3933	38.6	466
2013-09-03 03:30:03	68	3890	36.5	466
2013-09-03 03:29:03	69	3861	38.2	0
2013-09-03 03:28:03	69	3861	38.2	0
2013-09-03 03:27:57	69	3861	38.2	0
2013-09-03 03:26:57	69	4011	38.4	466
2013-09-03 03:25:57	69	3947	40.2	466
2013-09-03 03:25:40	60	3947	40.2	466

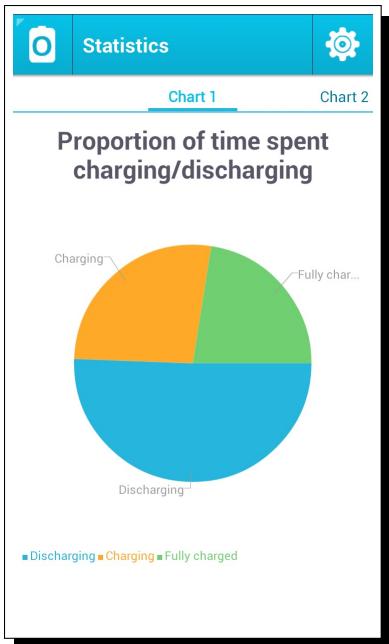
Figure B.4: The History Screen of Open Battery

## Statistics Screen

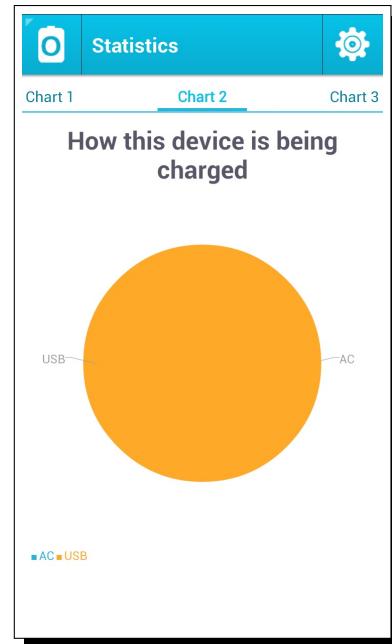
From the History Screen the user is able to navigate to the Statistics Screen where four charts are displayed, providing statistics regarding the charging characteristics of the user. Each chart is displayed individually in tabs and navigation across the charts is provided by swiping horizontally across the selected tab's contents. The four charts are shown in Figure B.5

## Preferences Screen

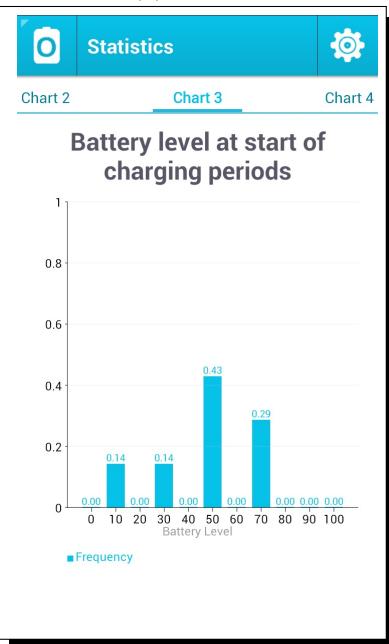
This screen is reachable from all other screens of Open Battery and displays the application's customizable options as a list, as illustrated in Figure B.6. The user can change his preferences by tapping an entry of the list and choose on of the available option values. More specifically, the user may enable/disable the battery logger and decide whether he prefers to save the log in a CSV file or not. Moreover, the user is able to choose the update rate of the logging procedure and the maximum size of the history data. The user is also offered several display options regarding the charts and the option to enable/disable the automatic submission of data to the Open Battery server.



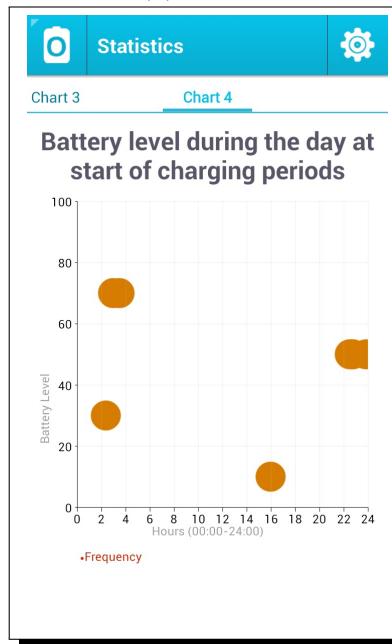
(a) Chart 1



(b) Chart 2



(c) Chart 3



(d) Chart 4

Figure B.5: The Statistics Screen of Open Battery

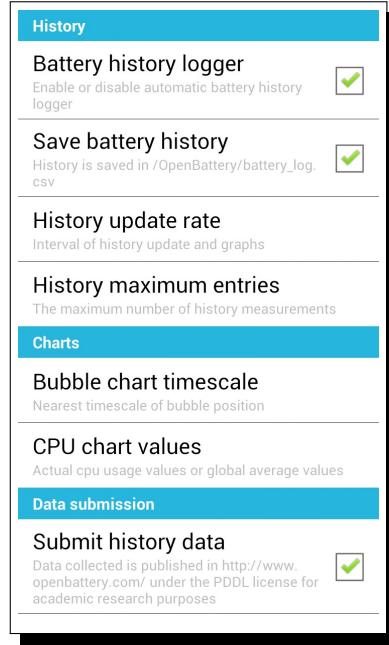
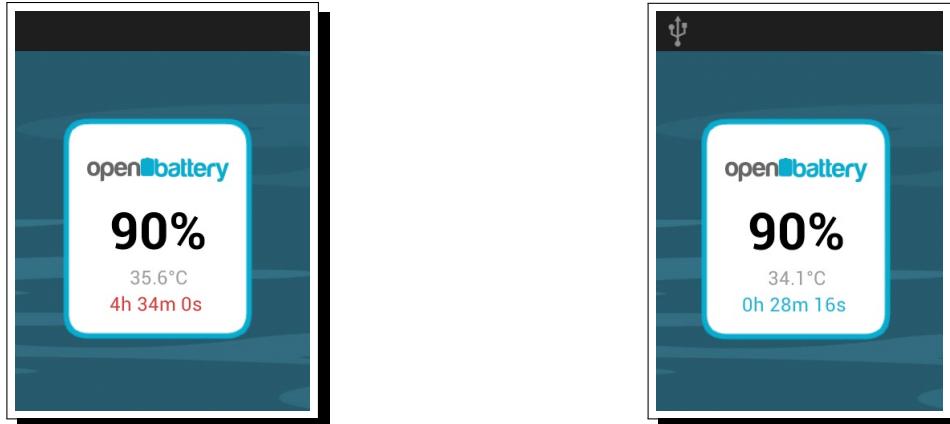


Figure B.6: The Preferences Screen of Open Battery

### B.3 Widget

In this section we present the Open Battery widget, which displays the battery's level, its temperature as well as an estimation of its remaining lifetime. When the device is discharging, the battery lifetime estimation is displayed in red colour. On the contrary, when it is plugged in to a power source, the prediction is displayed in blue colour.



(a) The Open Batter widget when the device is dis- (b) The Open Batter widget when the device is charg-  
charging ing

Figure B.7: The Open Battery widget

## Appendix C

# Software diagnostics of the new Open Battery application

In this chapter we present some technical details regarding the development of Open Battery. We also show the list of the software packages of the tool, as well as the package dependency graph.

### C.1 Software engineering tools

In this section we list the software tools that have been used in this project.

- **Programming languages** Java, php, sql
- **Code documentation** Javadoc
- **Android development** Android SDK tools, Android ADT, DDMS
- **IDE** Eclipse ver. 4.2.1
- **Unit testing** JUnit
- **Database** SQLite, PostgreSQL
- **Package dependencies analysis** Stan4J
- **Graph visualization** MatLab ver. R2011a
- **Code analysis** CodePro Analytix

### C.2 Software metrics

Table C.1 shows some technical details and software metrics of Open Battery Android application.

Metric	Value
Number of packages	14
Number of classes	59
Number of interfaces	9
Average block depth	0.85
Lines of code	6,402
Comments ratio	12.5%

Table C.1: Software metrics of Open Battery tool

### C.3 List of packages and classes

In this section we present the packages and the classes of the Open Battery Android application.

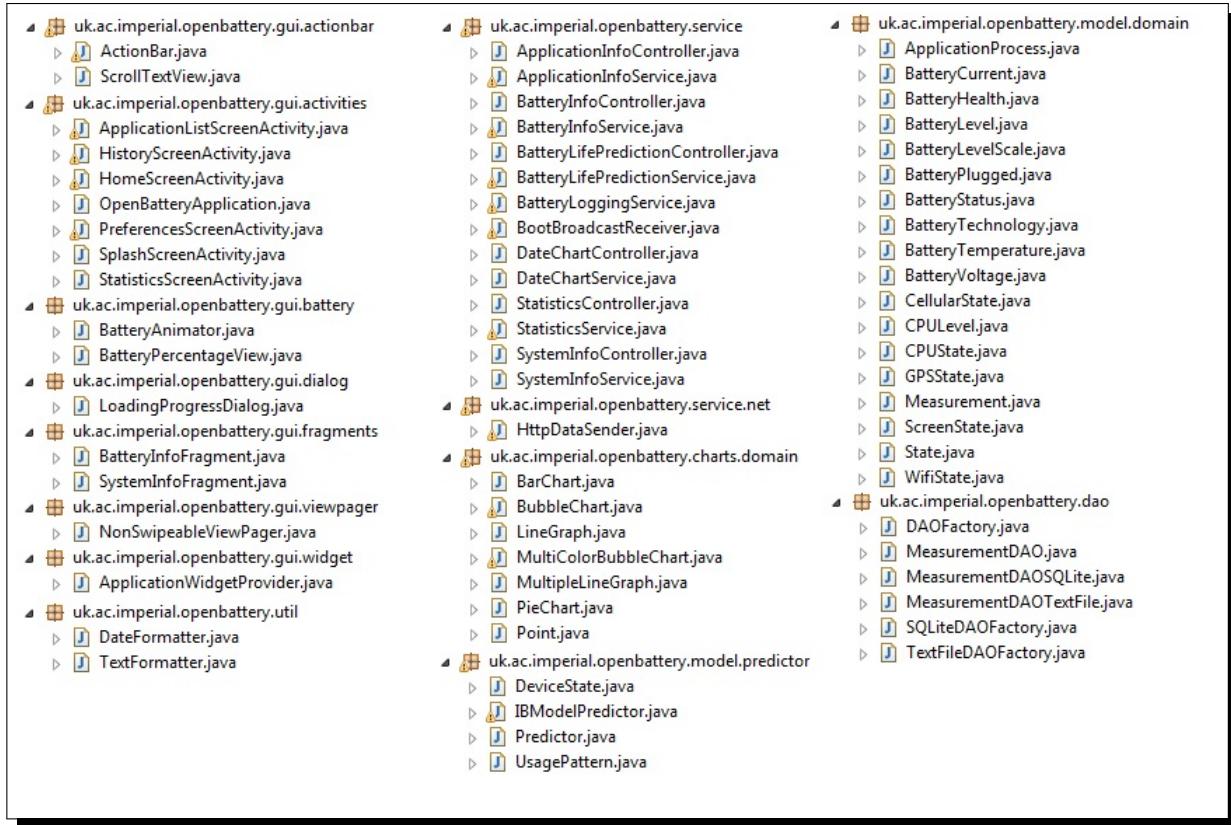


Figure C.1: The packages of Open Battery tool

## C.4 Dependency graph

In Figure C.2 we present the package dependency graph of the Open Battery application. As we can clearly see, the graph is acyclic, reflecting a good software architecture.

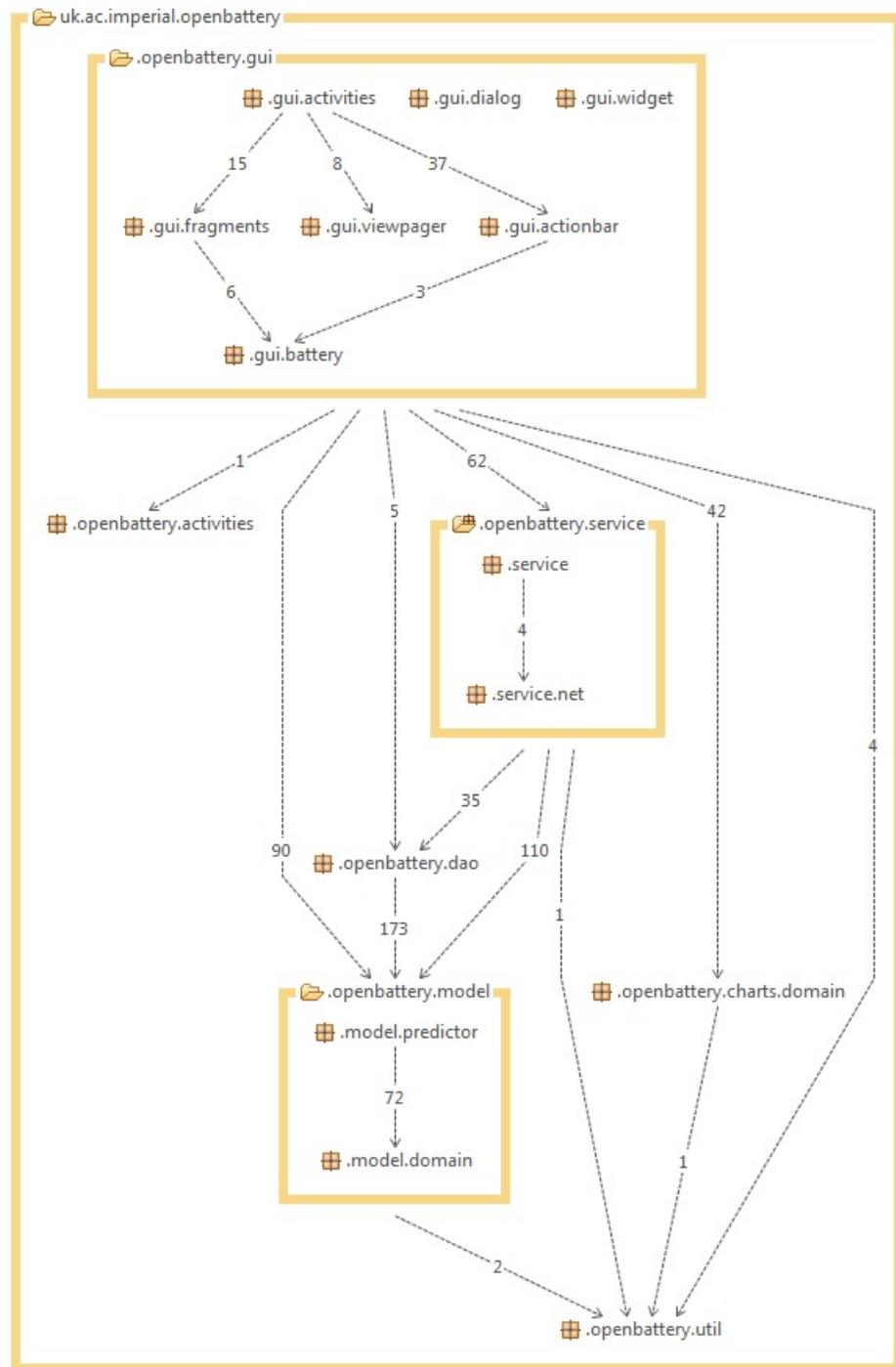


Figure C.2: The package dependency graph