MODELING SMARTPHONE POWER

by

Sameer Allan Alawnah

A Thesis Presented to the Faculty of the
American University of Sharjah
College of Engineering
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in
Computer Engineering

Sharjah, United Arab Emirates
June 2013

Approval Signatures

We, the undersigned, approve the Master's Thesis of Sameer Allan Alawnah

Thesis Title: Modeling Smartphone Power

Signature **Date of Signature** (dd/mm/yyyy)

_____ _____
Dr. Assim Sagahyroon
Professor
Department of Computer Science and Engineering
Thesis Advisor

_____ _____
Dr. Fadi Aloul
Associate Professor
Department of Computer Science and Engineering
Thesis Committee Member

_____ _____
Dr. Hasan Mir
Assistant Professor
Department of Electrical Engineering
Thesis Committee Member

_____ _____
Dr. Assim Sagahyroon
Head
Department of Computer Science and Engineering

_____ _____
Dr. Hany El-Kadi
Associate Dean
College of Engineering

_____ _____
Dr. Leland Blank
Interim Dean
College of Engineering

_____ _____
Dr. Khaled Assaleh
Director of Graduate Studies

Abstract

Battery technology has not advanced rapidly enough to keep pace with the growing energy demands of today's portable electronics. Leading this critical need for energy are smartphone devices which are being deployed and adopted at an increasing rate. Developing sound energy management techniques for these devices requires a good understanding of where and how battery energy is being utilized. Power consumption modeling is therefore crucial for understanding the inner working of these devices and for developing energy-efficient software to run on them. In this work, we attempt to develop power models for Android-based smartphones. A logger application is developed to monitor users' activity and collect data related to this activity. We then utilize regression techniques and neural networks to develop power models that relate power consumption to usage behavior. We demonstrate the feasibility of applying the two approaches and provide a detailed comparison between them.

Search Terms: Smartphones, Power Modeling, Regression, Neural Networks, Android

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years we have witnessed a phenomenal surge in the demand and utilization of portable electronic devices such as laptops, iPads, cellphones and smartphones. The sale and adoption of these devices is expected to increases in the next few years with these devices gradually becoming an essential part of our daily lives. Advances in the technology of wireless communication and shrinking VLSI technology has allowed for the production of small handheld devices with impressive communication bandwidth and ability to run complex applications. These applications include but are not limited to: audio, web browsing, video gaming, texting, and video streaming. The industry is keen on producing lighter devices with an ever increasing data rates and computing demands, and the race to satisfy consumers' appetite will no doubt continue to escalate. Since these devices depend on batteries for their operation, a major concern in the design of these devices is, therefore their energy consumption. Advances in battery technology have not paralleled the rapid advances in chip design and wireless telecommunication, and it is often the case that the computing power required exceeds battery capacity. This has brought the problems of power consumption, low power design, energy-efficiency, and optimal power management to the forefront of research issues pertaining to portable electronics.

In this work, we intend to tackle the issue of power consumption in portable devices using smartphones as a representative example of this family of devices.

Smartphones are being deployed and adopted at a remarkable pace. In 2012, smartphone penetration in the USA was 44% while in 2011 it was 31% [**1**]. We expect similar aggressive market penetration for most of the rest of the world where services are available.

Even though a great deal of research has been published in recent years related to smartphones, these devices still pose a challenge to researchers due to their propriety designs and other engineering barriers such as the closed and fragmented

nature of the many devices in the market [**2**] [**3**] [**4**]**.** Furthermore, there are many aspects to the power consumption problem that require researchers to tackle the problem from various angles including system level design, design of low-power integrated circuits, design of efficient user interfaces, utilization of efficient energy management techniques that include the powering down of unused components, and dynamic voltage-frequency scaling techniques.

An understanding of the inner workings of these devices should lead to the development of power and energy models that would eventually assist designers in developing sound energy management policies, and assist software developers in writing applications that are energy-efficient. A critical step forward in designing energy-efficient smartphones is an understanding of their energy consumption characteristics, and gaining an insight into where and how energy is used. The nature of users' interaction with these devices is also important to the design of better mobile devices in the future. Users' activity is the driving force and is a critical determinant in deciding battery life.

The need to try and understand the role of users' behavior with the hope of designing better systems and making efficient use of available energy is emphasized in two recent papers [**5**] [**6**]. The authors of these two papers studied the smartphones usage activity of a large number of users. They showed that the usage activity is quite diverse among users. This extent of usage diversity implies that mechanisms that work for the average case may be ineffective for a large proportion of the users. In the case of power modeling, usage diversity means average case power models may be insufficient to accurately predict power consumption for different users, and hence a usage activity-based power model is essential to accurately predict power consumption.

In this work, and in attempt to contribute to better designs of smartphones, we will approach the power modeling problem from a user's behavior point of view. We developed a logger application (running in the background) that tracks users' interaction with the smartphone over a period of time. It creates and logs power-related records using the smart battery interface built in the device. The data sets logged overtime are then used to develop power models using, first, regression modeling techniques, followed by applying neural network modeling approaches.

These power models will ultimately assist in understanding these devices and this will lead to the design and implementation of energy management policies that will elongate battery life. Power management in portable devices is the main technique by which we can reduce power consumption while ensuring performance.

The reminder of this thesis is outlined as follows: in Chapter 2 we provide an informative literature review of published and related work. Chapter 3 discusses the "data logger." Chapter 4 presents our attempt to use regression techniques in deriving power models, followed by Chapter 5 where we discuss our results in using neural networks to model the problem. In Chapter 6 we discuss and compare results obtained using these approaches, and in Chapter 7 we conclude this thesis.

Chapter 2
Background and Related Work

Smartphones are a recent addition to the portable electronics family and related published literature is still in its infancy. However, since smartphones are simply a continuation of a trend, they therefore have strong commonalities with previously introduced mobile phones. We will first give an overview of the sources of power dissipation in these devices, followed by a discussion of some of the research attempts made to address the power modeling problem in smartphones.

2.1 Sources of Power Consumption in Mobile Phones

A representative architecture of modern mobile phones is shown in Figure 1. The figure shows the major hardware components where each comes with its own energy demands for correct operation. Most of the power consumption is typically broken down between the following components: CPU, memory banks and controller, GSM, GPS, Bluetooth, LCD panel and touch screen, LCD backlight, Wi-Fi, audio (codec and amplifier), internal NAND flash, SD card, and camera. The system load or application heavily influences the power needs of these components. For example, if the load is a video game, more power will be drawn by these components than, when for example, the load is simply an editing session of a text file. In the following subsections we will describe the main power-draining sources in smartphones.

2.1.1 CPU Power

In general, CPU power is directly proportional to the supply voltage, clock frequency, and the capacitive load. Whether it is single core or multicore, the CPU is responsible for a sizable portion of the activity on the mobile in its active state. Different strategies can be used to optimize its power consumption. For example, by using appropriate scheduling policies the operating system can adjust one or more of the parameters that contribute to CPU power consumption. The most prevalent technique, however, is DVFS (dynamic voltage and frequency scaling). In a multicore architecture the problem is aggravated by the power consumed by inter-process communication. Communication channels are typically designed using multiplexers,

buses, drivers, etc., and hence additional power is drained due to the capacitive load of this added hardware.

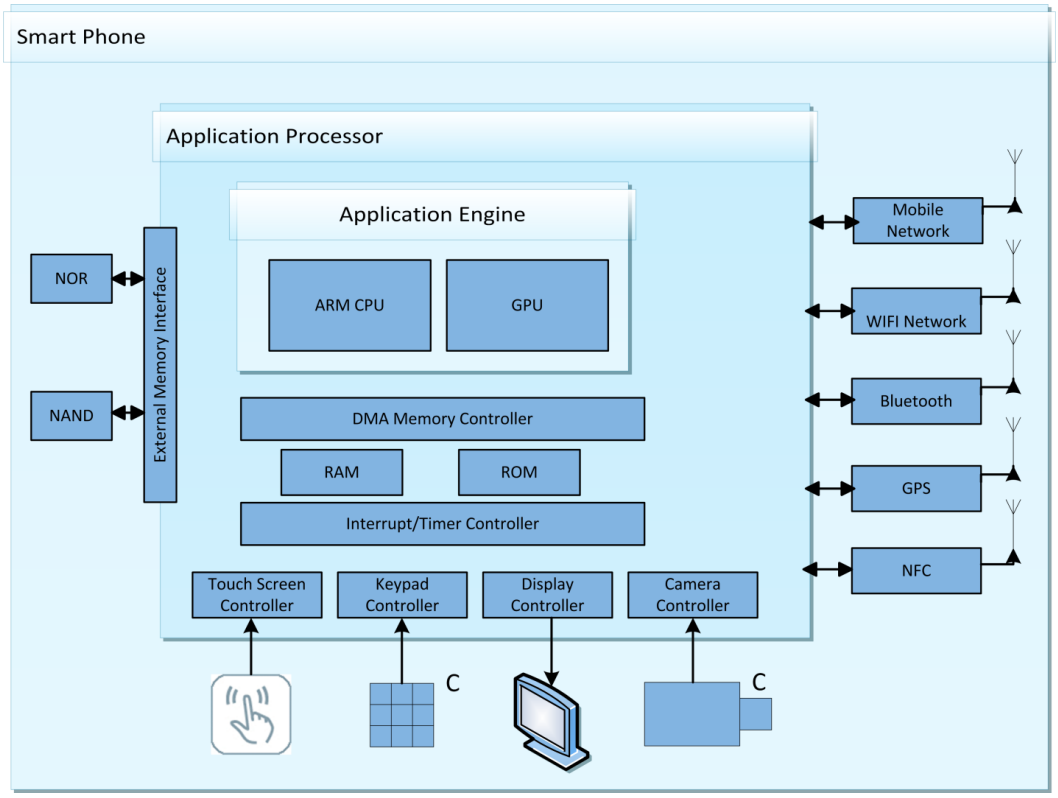


Figure 1: System level depiction of a smartphone

2.1.2 Memory and Storage Power

Generally, for embedded mobile systems, especially data-intensive applications, energy is consumed when memory is accessed for read and write operations. Additionally, energy is also required for any kind of processor-to-memory communication. Techniques such as data compression and optimization of the code are used to effectively reduce the frequency of accesses to memory, resulting in power savings. The bulk of storage is taken care of by the internal NAND flash and the external micro secure digital card (SD card). Here, as well, power is consumed during streaming reads and writes.

2.1.3 Network Power

Numerous smartphone applications generate some sort of traffic network. This activity, whether data-related or voice-related, triggers the activation of a number of hardware components such as the digital baseband chip that supports the 3G protocol standards (Universal Mobile Telecommunications System, General Packet Radio

Service, and Global System for Mobile Communications). Further, the implementation of signal processing algorithms such as Viterbi Decoders [**7**] and Turbo Coders [**8**] use computations that require intensive digital signal processing and thus energy. The RF (Radio Frequency) subsystem, as well, consumes energy in transporting data.

2.1.4 Displays

Today's smartphone users demand larger and brighter displays with high resolution. For displays the power consumed is mainly due to backlight and the degree of brightness. The energy overhead depends on the size of the display, its architecture, the technology used in designing it, and whether it uses colors or not.

2.1.5 Sensors

In addition to the camera module, today's smartphones are equipped with additional sensors such as GPS (Global Positioning Service), accelerometers, and gyroscopes. Upon invocation of any of these devices, energy is consumed and hence has an impact on battery life.

2.1.6 Applications Demand

Finally, the nature of the application running on these devices is a decisive factor in determining battery life. Typical applications include text messaging, audio and video playback, video gaming, emailing, phone conversations, and web browsing. The amount of energy needed to successfully launch and run an application varies depending on the application's computing needs.

2.2 Power Consumption Breakdown

A number of papers have reported on the power consumption of these devices, often using models from different manufacturers. Rice and Hay [**9**] examined the power consumption of Android 1.5 mobile phones. They ran a number of automated tests focusing on network-related activity, particularly the cost of sending messages over a wireless network. Their tests were meant to replicate user scenarios and did not require user interaction. They concluded that energy optimization for data transmission purposes varies depending on factors such as the device itself, the operating system, and the context or operating scenario.

Caroll and Heiser [**10**] reported on the power consumption characteristics of the Openmoko Neo Freerunner. The main difference between this device and other recent smartphones is the fact that it is not equipped with a camera or 3G modem. A data acquisition unit and sense sensors were used to measure voltage and current of each hardware component. The researchers simulated near real-time usage scenarios using benchmarks and computed the breakdown of power distribution between the various components including CPU, memory, touchscreen, storage, and the different networking interfaces. One of the limitations of this work was the lack of a 3G cellular interface in the device, coupled with the fact that the Freerunner model was an older generation of devices and cannot be considered a viable representation of the recent state-of-art smartphone devices on the market. In this work the authors concluded that the majority of power consumption could be attributed to the GSM module and the display. Furthermore, Perrucci and others [**11**] used a Nokia N95 running Symbian OS 9.2 operating system for their measurements. This model had a built-in energy profiler developed by Nokia. This energy profiler allowed for measuring power, current, temperature, signal strength, and CPU usage without making use of any external hardware. Scripts were used to create usage scenarios and control the phone, eliminating the need to physically interact with the phone. The energy measurements reported included data communication aspects (Bluetooth, Wi-Fi and cellular). The power consumption of the CPU was also considered. The authors concluded that for the model used, wireless technologies were the major energy consumers, and not the CPU or display. The power consumption of smartphones over LTE networks is considered in [**12**]. The authors performed an analysis of different traces stemming from various applications in an effort to understand traffic characteristics. The platform used was the HTC thunderbolt which is the first LTE phone introduced to the market. Their work provided cumulative distribution functions of the packets' size and packet inter-arrival times of background traffic of common applications such as Skype and Twitter. Their analysis showed that background traffic could be noticeably different, thus resulting in different energy demands. The authors provided statistical characteristics that could be used to develop smart power management policies. Power demand at the application level and a methodology for energy performance testing in smartphones is introduced in [**13**]. The authors identified and tested a process, which they claim, could be followed by application developers to assess the energy performance of a given application.

However, their proposed approach provides an overall power consumption scenario without relating it to individual component demands.

2.3 Power Modeling

The ability to characterize and model power consumption of mobile devices is critical to the development and guidance of the energy optimization aspects of these devices. Designing energy efficient smartphones requires insight into their power behavior at all levels and this in turn calls for the creation of models that capture their behavior. Even though the power consumption problem has been extensively investigated in portable devices such as laptops, few attempts have been made to model power in smartphones. In what follows we will discuss some of these contributions.

An attempt to model the energy cost of an application running on a portable wireless device is discussed in [**14**]. The authors divided the functionality of the device into two categories: computation and communication. They then modeled each component using a state transition graph where each state has an average power cost (hardware dependent); this cost was measured using scripts to force hardware components to be in a predefined state and then assess its power in that state, as well as a residence time (determined by the attributes of the application) associated with it. Thus, the application behavior together with the power cost of the different states determined the energy cost of the application. They used an external setup to measure the power. Similarly, Shye and colleagues [**15**] developed a power model derived using data collected from real end users over a period of time. They argue that architects have always overlooked the end user who in essence is the workload, and hence any practical power model should take this into consideration. The researchers developed a logger application to capture users' activity. Their logger application, which required a rooted device to operate, polled the smartphone component statistics to measure their utilization. Furthermore, usage scenarios were simulated by scripts, and power was measured using an external setup. The shortcomings of this approach were two-fold: The usage scenarios were scripted, and a lab setup was required. This limits the generalization of the outcomes since it does not necessarily cover all possible users' usage patterns. The model makes use of high-level real time measurements collected for a selected set of hardware components. A linear regression model was then used to build the model. The parameters selected to build

the model include CPU, screen, and Wi-Fi. The system was validated using additionally collected data. The authors concluded that the power breakdown of a device was highly dependent on users' behavior and usage patterns and that the screen and CPU tended to consume most of the energy during an activity.

A model for energy level prediction is presented in [**16**]. The researchers collected data from a set of Blackberry smartphone users and then exploited energy traces within the collected data set to build an energy emulation toolkit. Users here were divided into three groups depending on their energy consumption characteristics, that is, their daily battery charge and discharge pattern or behavior. A prediction algorithm was then used to predict energy level. Using results from the energy emulation kit, developers could then modify their designs or fine tune certain parameters to optimize energy consumption. Similarly, Zhang and others discuss a battery-based power model in [**17**]. Based on measurement oriented experiments, they excluded the hardware components with negligible power consumptions, such as the SD card, from being used in the model. The modeled components were CPU, display, GPS, Wi-Fi, cellular, and audio interfaces. A set of training programs was next used to determine the relationship between a state variable and the power consumption for each hardware component selected for the model. The authors next proposed the use of battery discharge behavior and the built-in battery voltage sensors in some smartphones to determine the average power consumption that resulted from varying the power states of the different components. The smartphone components were held in a particular state for long periods of time while the state of discharge of the battery was monitored using the built-in voltage sensors, therefore providing an estimate of the power consumption for the particular activity state. The total consumed energy within that test period or interval was then computed. This was repeated for different states and regression techniques were used to derive models based on battery behavior.

Furthermore, in [**18**], the authors put forward a power modeling technique based on tracing system calls of an application. They argue that their approach captures both utilization-based, as well as non-utilization based power behavior of components, and hence leads to an accurate fine-grained energy estimation approach. The premise in their work was that applications gain access to hardware (I/O) components via system calls and therefore their names, together with the parameters,

provide a clear indication as to which component is being requested and the level of utilization of this component. System calls were thought of here as triggers to power state transitions allowing the use of a "finite state machine" to model these state transitions. The proposed approach uses these power states to model and estimate the power of the smartphone. This approach requires a rooted device to capture system calls; it also uses an external setup to measure power. These two facts limit the applicability of the technique and force trace generation in lab settings only. A self-constructive high-rate system for energy modeling in battery-operated and mobile systems, which is thus applicable to smartphones, was presented in [**19**]. Here, the smart battery interface was used as a means of self-power measurement in the device, and the authors described a self-modeling paradigm where a mobile system has the ability to automatically generate its energy model. Their argument was that the dependencies of energy models on the hardware configuration of a device, the pattern of usage of the devices, and even the device's battery, is a strong reason for exploring a self-modeling approach. A data collector was developed to interact with the native operating system of the device to acquire energy predictors for the model using regression techniques. This approach was tested using an IBM T61 laptop and Nokia N900 mobile devices, both with a Linux OS; it is very difficult to test this approach with un-rooted Android devices since native Linux programs cannot be used with these devices.

In [**20**], power modeling was approached using pattern analysis techniques. In this approach, Lee and others proposed segmenting users' logged data into a number of small time windows called "chunks." A chunk is a set of power-related data collected and computed during one percent of used battery capacity. Chunk data includes components such as average power, CPU utilization, frequency, and display activity. The researchers grouped the chunks based on the hardware components accessed. Chunks with standard small deviations were kept and others were discarded. Regression analysis was then performed to generate a model. In the reported work, only power models related to CPU and display were developed. Furthermore, in [**2**] the authors presented a logging system for Android smartphones. This system logs information about almost all components of the smartphone; however, it does not log "current" information, and hence could not measure power consumption, which meant

that power models could not be built. Usage samples were saved on the file system of the smartphones and uploaded to a database server for further analysis.

Table 1 shows a comparison between previously discussed work and our work.

Table 1: Comparison between reported work and proposed work

Work	Components	Power Measurement method	Logging method	Trace generation	Logger execution mode	Uploading Utility	Modeling techniques
[2]	All	N/A	Event + Polling	User	User	Yes	N/A
[15]	All	External setup	Polling	Script	User + Kernel	No	Regression (OLS)
[17]	All	Battery discharge curve	Polling	Script	Unknown	No	Regression
[18]	All	External setup	System calls +Polling	Script	User + Kernel	No	Finite State Machine
[19]	All	Smart Battery Interface	Polling + event	User	User + Kernel	No	Regression (TLS)
[14]	Internet + Computation	External setup	N/A	Script	N/A	No	State-transition diagram.
[20]	CPU + Display	Battery Level	Unknown	User	Unknown	No	Pattern Analysis
Proposed Model	All	Smart battery interface	Event + Polling	User	User	Yes	Regression + Artificial Neural Networks

In Table 1, the first column contains the paper reference, second column denotes the smartphone components included in the power model, and the *Power Measurement Method* column specifies the method used to measure the power. Several methods can be used to measure power. In an external setup, an external data acquisition unit is connected to the smartphone to measure power; however, this badly affects the portability of the smartphone. Battery discharge curves can be used to estimate the power using battery voltage and level; however, these curves are dependent on the type of battery, age, and other factors, which means one must construct the curve for each battery. A smart battery interface provides a reliable way to measure power; however, it exhibits some random errors which can be reduced when averaging the power values over a period of time.

Furthermore in the table, *logging method* denotes the techniques used to capture user behavior information. *Event-based* techniques have the lowest overhead to the tested system, since they are captured at the time of their occurrence. On the contrary, *polling-based* techniques have higher overhead to the tested system since the

logging application must be continuously running to capture any change in the logged information. *System calls* can be used to log information but the logger requires root (super-user) permission in order to capture them, which is not the case for most Android smartphones users.

Furthermore, *trace generation* specifies the method used to generate usage traces. Many researchers use *scripts* to simulate the usage behavior and hence force some usage behavior patterns, while other researchers use the traces generated by *users* while using their smartphones. These traces are more representative of the actual usage behavior of the smartphone.

Logger execution mode denotes the execution mode required by the logger application. *User* mode is the normal execution mode; all Android devices support this mode. *Kernel* mode is the super user mode (root). An Android device must be rooted to support this mode. *Uploading utility* is used to upload log files to a database server for further analysis rather than processing them locally on the smartphone.

Modeling techniques are the techniques used to model power consumption; these techniques include Regression Analysis, Finite State Machines, and State Transition Diagrams. In our work we used Regression Analysis and Artificial Neural Networks to model the power consumption of smartphones.

Chapter 3
The Data Logger

For this study, and in an effort to relate power consumption of a smartphones to users' utilization, we selected a sample of ten AUS students and monitored their smartphone activity with the assistance of a data logger program installed in their *Android-based* smartphones. The logger application continuously ran in the background without compromising the users' privacy. It ran continuously and started when the phone was turned on. We collected the maximum amount of usage-related data for synthesis and analysis as described in later sections. An overall layout of the system architecture used in this work and its various components is depicted in Figure 2. The operation and the interaction of the various entities will be described in later sections.

3.1 The Logger Application

Android is an operating system based on Linux with a Java programming interface. Android allows background processing, provides a rich user interface library, supports 2-D and 3-D graphics using the OpenGL libraries, and provides access to the file system and an embedded SQLite database. The Android Software Development Kit (Android SDK) provides all necessary tools to develop Android applications customized for various users' requirements, including research, which is the case in this work.

The logger developed for this work makes valuable use of a built-in android mechanism called broadcast/receiver. The operating system broadcasts messages about events that are taking place, such as a battery status change, the Wi-Fi connection being turned on, or the screens being switched off. However, some relevant and power-impacting usage parameters have no broadcast actions or messages associated with them, and in this case we use polling as a means of sampling changes in these parameters (examples, include "current" and "audio utilization").

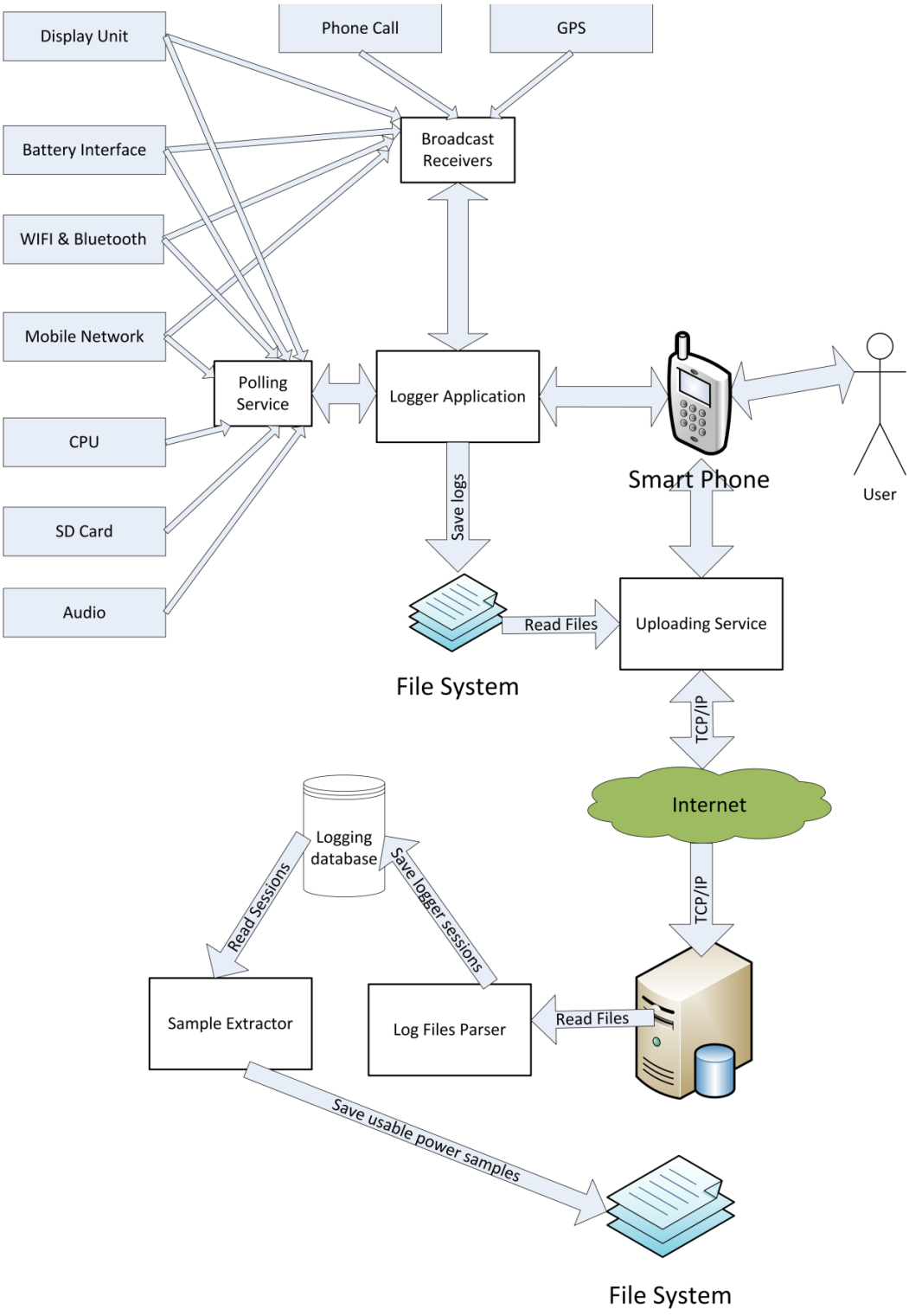


Figure 2: System Architecture

Figure 3 shows the classifications of variables collected using the logger application. Event-based variables are the variables associated with broadcast actions and hence we can capture their new values using the broadcast receivers. Other

variables that didn't have broadcast action associated with them are called Polling based variables; we read or sample their new values regularly. Some of the polled variables have associated operating system (OS) counters that can be read using Linux virtual files examples include CPU usage information, while others have no OS counters. We sample the variables with OS counters at a relatively low rate since any increment in their values will be monitored and saved by the OS. Variables without OS counters must be sampled at a relatively high rate in order to capture any change in their values.

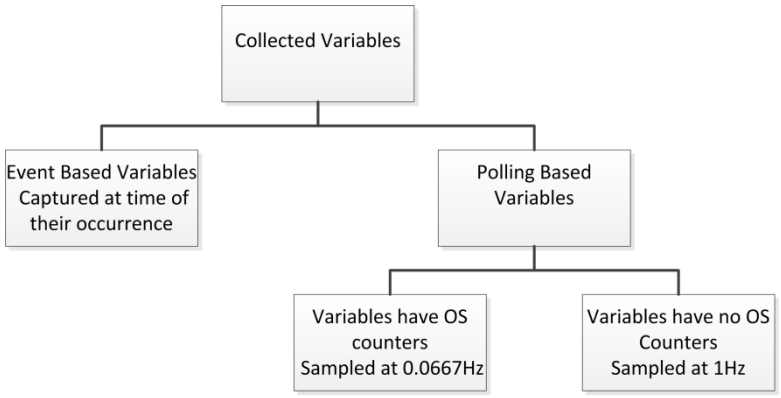


Figure 3: Variables Classification

To log the collected data on the SD card, we use the Android Logging wrapper for Log4J [**21**] Java library. It is a powerful logging tool for Java designed to be used in heavily multithreaded systems.

The format used to store captured event or polling-based user data is shown in Figure 4.

Figure 4: Logged Data Format

The first field is simply a time stamp highlighting the time of the action or event. The action code is one (1) for data samples collected using polling. Event-based variables are assigned different unique codes, for example, three (3) for the screen status, and so on. These codes are used to identify messages in the log file parser and sample extractor. A listing of all messages is provided in Table 2. The first column contains the name of the message. The second column contains a description of the triggering cause which can be main program counter used to sample all polling-

based messages, or any other event triggered by some change in the mobile environment. The third column contains the logged data format, starting with the timestamp (TS) at which the event occurred, followed by the action code and all other message information. The last column specifies whether the message is event-based or polled-based. The ALL-POLLED-VAR message contains all the information related to polled events (to be described later). A discussion of polled- and event-based messages follows.

Table 2: Logger Application Messages

Event Name	Triggering Cause	Format	Type
ALL-POLLED-VAR	Main Timer Action	TS,1,[data separated by ‘,’] contains all polling based variables information	Polling
SCREEN_STATUS	Screen switched on/off	TS,3,[new screen status]	event
POWER_CHANGED	Charger connected/disconnected	TS,4,[charging state]	event
WIFI_STATE_CHANGED	WIFI enabled/disabled	TS,5,[WIFI state]	event
WIFI_CONNEECTION_ CHANGED	WIFI connected/disconnected	TS,6,[WIFI connection state]	event
WIFI_RSSI_CHANGED	WIFI Received Signal Strength Indication changed	TS,7,[new WIFI RSSI]	event
CALL_STATE_CHANGED	Phone ringing/in call	TS,9,[Phone state(IDLE,ringing,Offhook)]	event
DATA_CONNECTION_ CHANGED	Mobile Data connected/disconnected	TS,10,[connection state],[network type]	event
DATA_ACTIVITY	Data traffic direction changed	TS,11,[data traffic direction]	event
PHONE_SERVICE_STATE	Phone becomes In-service/ Out-Of-Service/emergency	TS,12,[service state]	event
GSM_RSSI_CHANED	Mobile network Received Signal Strength Indication changed	TS,13,[success/failed],[new RSSI]	event
GPS_EVENT_CHANGED	GPS connected/disconnected	TS,14,[GPS state]	event
BT_STATE_CHANED	Bluetooth enabled/disabled	TS,15,[Bluetooth state]	event
BT_DISCOVERY_STARTED	Bluetooth discovery started	TS,16	event

BT_DISCOVERY_ENDED	Bluetooth discovery ended	TS,17	event
BT_DEVICE_CONNECTED	Bluetooth device connected	TS,18	event
BT_DEVICE_DISCONNECTED	Bluetooth device disconnected	TS,19	event
LOGGER_STARTED	Logger application started	TS,21	event
LOGGER_ENDEED	Logger application exited	TS,22	event
BATTERY_CHANGED	Some battery parameters changed	TS,23,[voltage],[level],[temperature],[status]	event

3.1.1 Event-Based Messages

In this section we describe the event based messages and the process of recording them. Our logging application registers an *asynchronous actions receiver* to log different broadcasted data, *PhoneStateListener* to log all data related to the phone's state, and *GPSListener* to log data regarding GPS usage. Figure 5 shows the UML designed for the aforementioned receivers and listeners.

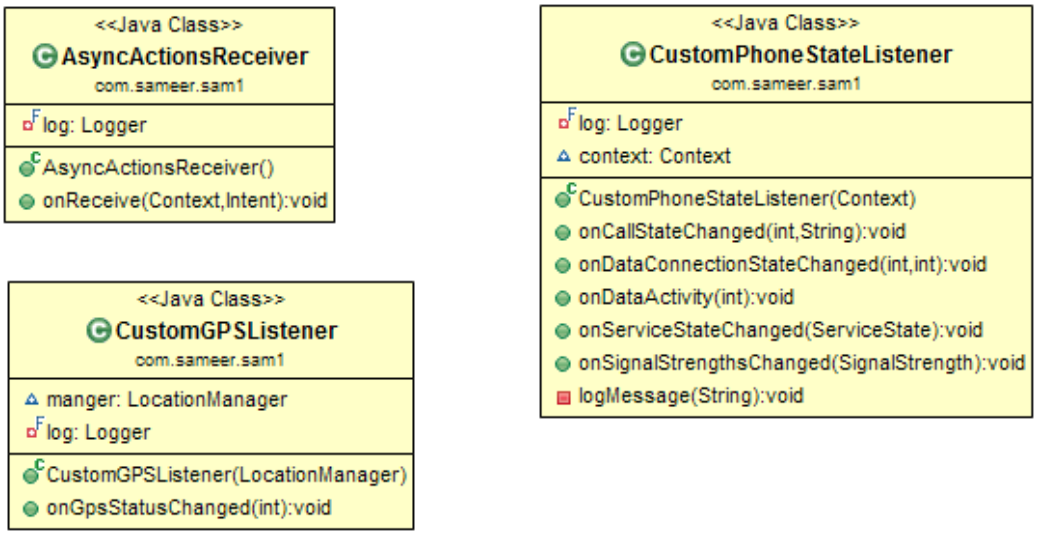


Figure 5: UML for Asynchronous Actions receivers/listeners

The *CustomPhoneStateListener* class is responsible for logging changes in the phone's state; it contains five handlers that are used to generate five different logger messages: CALL_STATE_CHANGED, DATA_CONNECTION_CHANGED,

DATA_ACTIVITY, PHONE_SERVICE_STATE and GSM_RSSI_CHANED using the corresponding functions.

The *CustomGPSListener* class is responsible for logging changes in the GPS status. When any change in the GPS status occurs, the *onGpsStatusChanged* method is called. Inside this method we check the new status of the GPS and log it using a GPS_EVENT_CHANGED logger message.

The *AsyncActionReceiver* class is responsible for logging all other event-based actions not handled by the previous two listeners, except the "all-plled-var" logger message which is generated using the Main Program Timer.

3.1.2 Polling Based Variables

CPU statistics and other data that cannot be logged using actions broadcasted by the Android OS are captured using a continuous timer in our logging application. To reduce the overhead incurred by our application, we set the time interval for the timer to 15 seconds. Choosing long time intervals for the main timer will not affect the overall accuracy of our average readings since the OS itself samples the system parameters at a relatively high rate (typically 10 msec). To calculate the values of these variables, we only take the difference between values reported by the OS. For example, virtual file */proc/stats* contains all CPU usage information. To read the total CPU IDLE time in some time interval, we subtract the value of CPU IDLE in */proc/stats* at the beginning of the time window from that at the end of the time window to obtain the total CPU IDLE time in that time window. Dividing the results by window time interval in seconds yields the average value of the CPU IDLE time within that window of time. This approach is used in collecting information about different aspects of the smartphone, simply defines variables to hold previous readings, and at the beginning of each time window read the new values, subtract old values from them and save the results, and finally updates old values to be equal to the new ones (new values becomes old values for next timer interval). Figure 6 shows all classes implemented to collect different statistics of our target system; they all contain an update method that is called at the beginning of each timer window to retrieve statistics and update previous values.

The *CPUStats* and *CoreStats* classes are used to collect information about the CPU usage; we gather this information using the */proc/stat* and

/sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state Linux virtual files. A sample output of the */proc/stat* virtual file is shown in Figure 7.

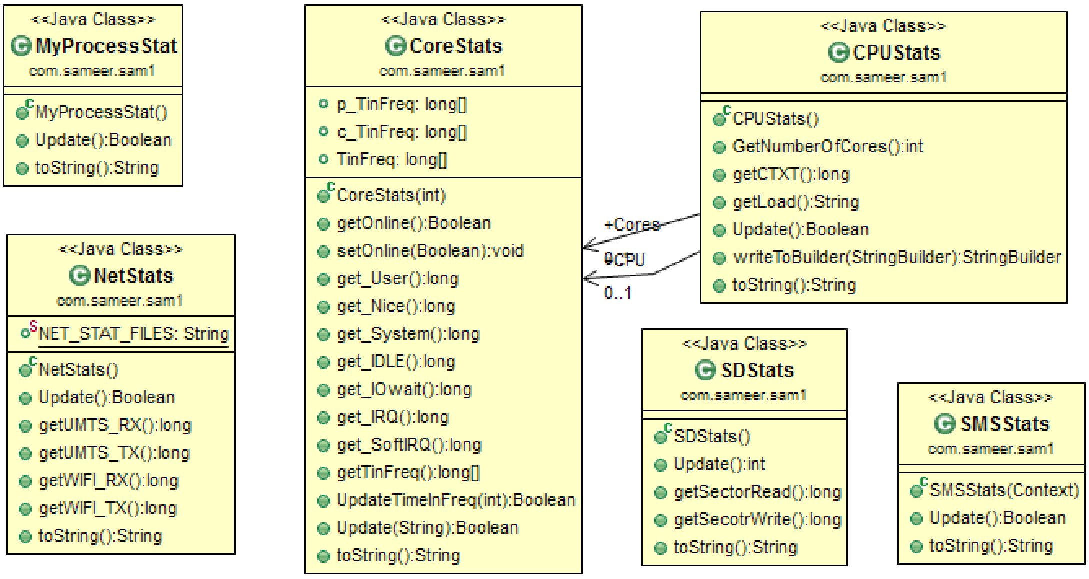


Figure 6: UML design for System Statistics collector classes.

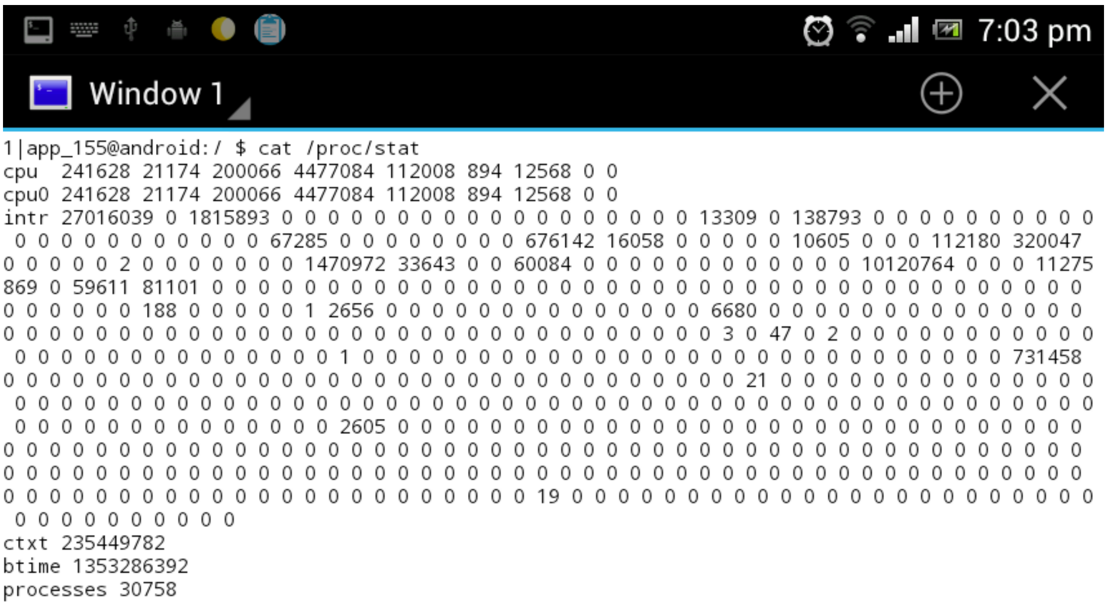


Figure 7: Sample output of the /proc/stat Linux virtual file.

The first line aggregates the numbers in all CPUs/Cores, the following line(s) are for different CPUs/Cores (in this example we have only cpu0), the explanation of the columns contents are as shown in Table 3 (from left to right).

Table 3: Illustration of the content of /proc/stat Linux virtual file

Execution mode	Description
User (column 2)	Number of time units the CPU consumed while executing user-space programs.
Nice (column 3)	Number of time units the CPU consumed while executing niced programs (low priority programs)
System (column 4)	Number of time units the CPU consumed while executing kernel level processes.
Idle (column 5)	Number of time units the CPU consumed doing nothing.
IOwait (column 6)	Number of time units the CPU consumed while waiting for IO to complete.
Irq (column 7)	Number of time units the CPU consumed while serving Interrupts.
Softirq (column 8)	Number of time units the CPU consumed while serving software interrupts.

The line starting with *ctxt* in Figure 7 contains the total number of context switching the OS performed.

The file *time_in_state* virtual file provides information about how much time the CPU core spent while operating at each frequency supported by that core. By subtracting values at the beginning of the time window from those at the end we obtain the time spent in each frequency during that window. Dividing values by the duration of the time window in second yields the fraction of time the core spent in each frequency.

The *NetStats* class provides statistics about network traffic; we use the */proc/self/net/dev* Linux virtual file to read network traffic data for all network interfaces. Following the same procedure in *CPUStats*, we calculate the average number of bytes sent/received per time window for each interface.

The *SDStats* class provides statistics about the SD traffic; we use the */proc/diskstats* Linux virtual file to read the SD traffic. Since most of the mobiles

contain both internal storage and external SD we are interested in measuring total traffic of both storages. These storages are denoted by *mmcblk0* and *mmcblk0*. In the same way used in the previous classes, we calculate the average number of sector read/write per time interval.

The *SMSStats* class provides statistics about SMS sent/received per time interval. We use Android content providers "*content://sms/sent"* and *"content://sms/inbox"* to give us the total number of sent and received SMS messages. From this information we can easily calculate the average number of SMS sent/received during a certain time window.

The *MyProcessStat* class provides statistics about our logger CPU usage. We used the */proc/[PID]* Linux file to read our process information and get the average CPU time usage during the time window.

Table 4 shows the "all-polled var" (from Table 2) message variables used to represent all polling-based information.

Table 4: Polled-Variables List

Field	Meaning	Range	Single/Dual Core
Current	Average current value during sample window.	Any value	Both
Screen Brightness	Screen Brightness during sample window.	0 - 255	Both
Audio On	Fraction of time Audio was on during the sample window.	0 - 1	Both
UMTS TX	Number of bytes sent over Mobile network during sample window.	≥ 0	Both
UMTS RX	Number of bytes received over Mobile network during sample window.	≥ 0	Both
WIFI TX	Number of bytes sent over WIFI network during sample window.	≥ 0	Both
WIFI RX	Number of bytes received over WIFI network during sample window.	≥ 0	Both
SD Sector Read	Number of SD sector read operations during sample window.	≥ 0	Both
SD Sector Write	Number of SD sector writes operations	≥ 0	Both

	during sample window.		
CPU Statistics status	Whether last CPU statistics inquiry success or failed	0 or 1	Both
CTXT	Number of context switching occurs during sample window.	≥ 0	Both
CPU User	Number of time units that CPU spent executing normal processes during sample window.	≥ 0	Both
CPU Nice	Number of time units that CPU spent executing niced processes during sample window.	≥ 0	Both
CPU System	Number of time units that CPU spent executing kernel-level processes during sample window.	≥ 0	Both
CPU IDLE	Number of time units that CPU spent doing nothing during sample window.	≥ 0	Both
CPU IO Wait	Number of time units that CPU spent waiting for I/O to complete during sample window.	≥ 0	Both
CPU IRQ	Number of time units that CPU spent servicing interrupts during sample window.	≥ 0	Both
CPU Soft IRQ	Number of time units that CPU spent servicing software interrupts during sample window.	≥ 0	Both
Core0 User	Number of time units that Core0 spent executing normal processes during sample window.	≥ 0	Dual Core only
Core0 Nice	Number of time units that Core0 spent executing niced processes during sample window.	≥ 0	Dual Core only
Core0 System	Number of time units that Core0 spent executing kernel-level processes during sample window.	≥ 0	Dual Core only
Core0 IDLE	Number of time units that Core0 spent doing nothing during sample window.	≥ 0	Dual Core only
Core0 IO Wait	Number of time units that Core0 spent waiting for I/O to complete during sample window.	≥ 0	Dual Core only

Core0 IRQ	Number of time units that Core0 spent servicing interrupts during sample window.	≥ 0	Dual Core only
Core0 Soft IRQ	Number of time units that Core0 spent servicing software interrupts during sample window.	≥ 0	Dual Core only
Load 1min	CPU and IO utilization during last one minute	≥ 0	Both
Load 5min	CPU and IO utilization during last 5 minutes	≥ 0	Both
Load 15min	CPU and IO utilization during last 15 minutes	≥ 0	Both
Running Processes	(Number of running processes)/(total number of process)	$\frac{\geq 0}{> 0}$	Both
Last Process	PID of the last accessed process	≥ 0	Both
Core0_TimeInFreq(0)	Number of time units first core operate at frequency 0 during sample window.	≥ 0	Both
Core0_TimeInFreq(1)	Number of time units first core operate at frequency 1 during sample window.	≥ 0	Both
……	……..		
Core0TimeInFreq (n-1)	Number of time units first core operate at frequency (n-1) during sample window.	≥ 0	Both
Core0TimeInFreq (n)	Number of time units first core operate at frequency (n) during sample window.	≥ 0	Both
SMS Sent	Number of SMS Sent during sample window.	≥ 0	Both
SMS Received	Number of SMS received during sample window.	≥ 0	Both
My Overhead (User)	CPU time that the logger application consume in User mode during sample window.	≥ 0	Both
My Overhead(kernel)	CPU time that the logger application consume in Kernel mode during sample window.	≥ 0	Both

3.1.3 The Auto-Uploading Service

The *log4j* logging API is configured to use *RollingFileAppender*. This appender is in turn configured to save logging files with a maximum size of 100 KB in the SD card. If the logging file exceeds the size limit, *log4j* will append a sequence number to its name and create another file for future logging. A service utility is next developed to upload finished logging files to a database residing in a remote server for future analysis.

Network communication is typically a power-hungry activity; therefore, the uploading service is designed to run automatically when two conditions are satisfied: the smartphone is connected to a charger, and a Wi-Fi connection exists. Otherwise, a user can select to activate a manual upload to the server.

We use HTTP protocol to upload the zipped files along with some information to a normal HTTP server. The server part of the logging operation is simple; it consists of a normal PHP page that saves the uploaded file along with other information into the database. We use the phone IMEI number to identify the device, while we use the manufacturer and model number to identify phone-specific parameters such as number of CPU cores. Figure 8 is a snapshot of our database.

ID	file_name	file_size	file_content	NC	FPC	MFG	MDL	IMEI	dTime
540	2loqwk.zip	114249	BLOB	2	23	Sony Ericsson	LT26w	353375050925097	2013-02-05 18:26:12
541	ed47n0.zip	23578	BLOB	2	23	Sony Ericsson	LT26w	353375050925097	2013-02-05 19:56:11
542	kr375z.zip	25841	BLOB	2	23	Sony Ericsson	LT26i	351710052514782	2013-02-05 19:57:04
543	q2lu31.zip	22476	BLOB	2	23	Sony Ericsson	LT26i	351710052514782	2013-02-05 21:57:03
544	g4c8jb.zip	17760	BLOB	2	23	Sony Ericsson	LT26w	353375050925097	2013-02-05 22:11:14
545	lzb0kw.zip	22543	BLOB	2	23	Sony Ericsson	LT26i	351710052514782	2013-02-05 23:57:02
546	7bb95r.zip	17381	BLOB	2	23	Sony Ericsson	LT26w	353375050925097	2013-02-06 00:11:17
547	y2rotn.zip	16906	BLOB	2	23	Sony Ericsson	LT26w	353375050925097	2013-02-06 02:11:16

Figure 8: Snapshot of the Log Files Database

In Figure 8 *file_name* denotes the zipped file name, *file_size* is the size of the file, *file_content* is the binary content of the file, *NC* is the number of CPU cores of the mobile device, *FPC* is total number of supported frequencies that each core can operate at, *MFG* is the smartphone manufacturer, *MDL* is the smartphone model, the

column labeled *IMEI* is the IMEI number of the mobile, and *dTime* is the server timestamp indicating the uploading time of the file.

3.2 Power Usage Samples Extraction

In this section we describe the programs developed to extract power usage samples from the uploaded raw data. These power samples will form the basis for the development of the power models. The raw data received from the logger application contains both polling- and event-based messages of users' activity-related data.

The *Log Files Parser* is responsible for fetching log files from the log files database and saving them in the messages database. The Sample Extractor reads the messages from the messages database and generates power samples as CSV files for each mobile.

3.2.1 Log Files Parser

As shown in the flow chart of Figure 9, the log file parser operation is quite simple: it fetches all compressed files that correspond to some IMEI number; un-compresses them; reads all files and adds their contents of messages tagged with an IMEI number, Time Stamp, and Action (Type of the message); and finally saves them in the messages database.

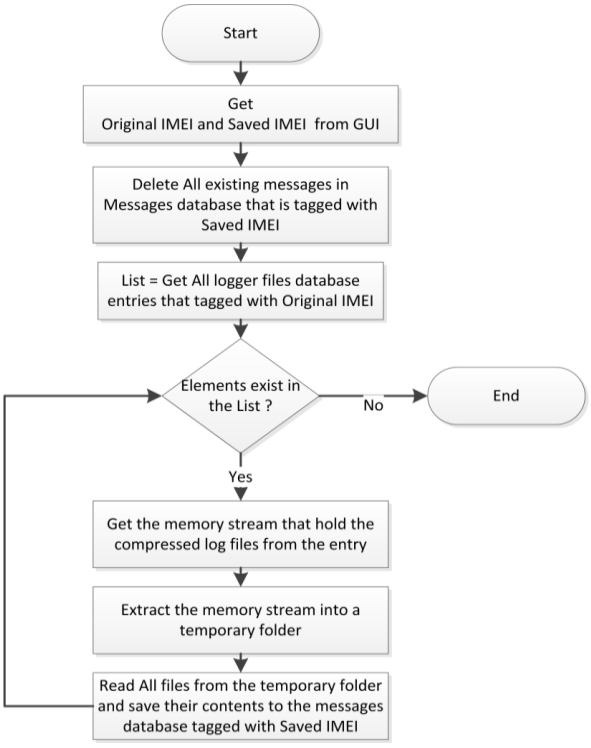


Figure 9: Flow chart of the Log Files Parser

3.2.2 Sample Extractor

After parsing log files and saving all messages in the messages database, we next run an application developed to extract power usage samples from the messages. Some of the sample parameters are trivial, for example, “screen brightness,” so no extra processing is needed for this parameter. On the other hand, other parameters such as “Data Activity” require normalizing with respect to the unit of time. We receive the information per sample time but we normalize this value to a one-second interval.

Processing Categories:

We divide the processing required to extract sample parameters into the following categories:

1. No processing:

   In this case we use the parameter values exactly as in the log file, for example the “screen brightness” parameter.
2. Normalization per 1 second:

   In this case we have the parameter value per sample window time, but since the sample window time is not constant, we need to normalize the parameter value per 1 second. An example for this case is the *CPU User* parameter.
3. Weighted Averaging:

   In this case the parameter values may change several times during the sample window; hence we calculate the weighted average of it. Figure 10 shows an example of this case for the RSSI (received signal strength indication) parameter.

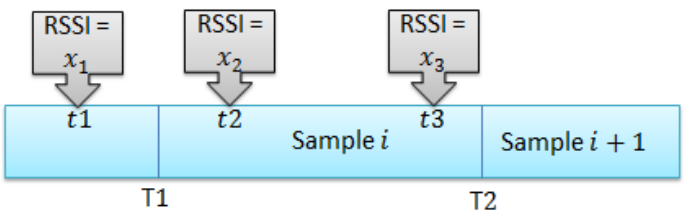


Figure 10: Weighted Averaged Parameter

In Figure 10, x_i denotes the new value of the RSSI, t_i is the time at which each signal change occurs, and T1 and T2 are start and end times of the Sample i window. We calculate the average RSSI as follows:

$$RSSI_{avg} = \frac{x_1 \times (t_2 - T_1) + x_2 \times (t_3 - t_2) + x_3 \times (T_2 - t_3)}{T_2 - T_1}$$

The key idea is to multiply the value of each parameter (x_i) by the fraction of time during which the variable is assumed to have this value within a certain time window.

4. Time Fraction Calculations:

   Some messages convey no numeric values and are only binary (On/Off). For these types of messages, we are only interested in the time fraction the corresponding parameters were on during the sample window. An example of this type of parameter is the *Screen ON* parameter. Figure 11 shows an example of this case.

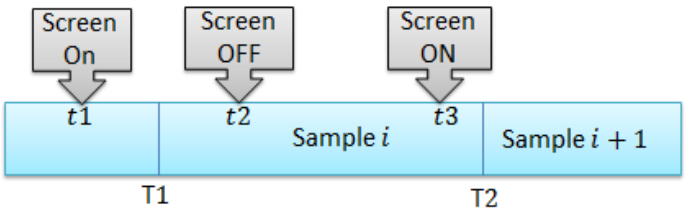


Figure 11: Time Fraction Parameter

We can calculate the Screen On time fraction as the following:

$$Screen\ On = \frac{(t_2 - T_1) + (T_2 - t_3)}{T_2 - T_1}$$

The key idea is to add up the time that the corresponding parameter was ON and divide it over total sample time window.

The smartphone family we target in this study uses the Qualcomm MSM8260 Snapdragon processor, which is an asynchronous symmetric dual-core processor. Either core can be scaled in terms of frequency and voltage independently. This causes complications to our logging system since the OS scheduler tends to put the second core in sleep mode whenever the system load is low, and we can't record any information regarding the second core while it in sleep state.

Figure 7 is a sample output of the */proc/stat* Linux virtual file used to obtain CPU time used in each mode per core. We can see that the first line contains the total CPU time for each mode, while the second line contains the same information but for the first core. If the second core is not in sleep mode, a line of its information will appear below this line. If we add the values for each mode in Core0 with the corresponding values in Core1, we find that they equal the values in the total CPU values (first line). To overcome the problem of the second core, we only log values for the CPU and Core0. Values for Core1 are calculated using equations (1) to (6). The variables used in these equations are defined in Table 3.

$$User_{Core1} = User_{CPU} - User_{Core0} \quad (1)$$

$$Nice_{Core1} = Nice_{CPU} - Nice_{Core0} \quad (2)$$

$$System_{Core1} = System_{CPU} - System_{Core0} \quad (3)$$

$$IDEL_{Core1} = IDLE_{CPU} - IDLE_{Core0} \quad (4)$$

$$IRQ_{Core1} = IRQ_{CPU} - IRQ_{Core0} \quad (5)$$

$$SoftIRQ_{Core1} = SoftIRQ_{CPU} - SoftIRQ_{Core0} \quad (6)$$

Asynchronous operations of the cores caused a major problem in the operation of the CPUFreq Linux driver. The *CPUFreq* driver automatically flushes *time_in_state* counters for *Core1* whenever the core goes to sleep. Unlike */proc/stat*, time*_in_state* does not have counters regarding the whole processor, so we cannot obtain counter values for *Core1*.

We tried to solve the previous problem by sampling *time_in_state* at double of the system HZ constant (100 Hz in current Android kernel), which mean running a timer with 5 milliseconds interval. But this caused the battery to drain to zero of its capacity within a few hours. Also, the application overhead itself affected our results. Thus we discarded this solution.

We assume that Core0 and Core1 operate at the same frequency at a time, so calculating the time fraction for each frequency for Core0 will be enough to model the frequency effect for both cores.

Data Sample Parameters:

Table 5 includes all parameters of power samples resulting from the extraction process. Note that the power is the product of voltage by current ($Voltage \times Current$). Current values are averaged in the logger application, so no processing is required to compute it. On the other hand, voltage is given as different messages that indicate changes in voltage, so we need to calculate the weighted average of the voltage during the sample window.

Table 5: Parameters list extracted from the logger messages using our sample generator application.

Parameter	Description	Processing used	Range
Power	Average power consumed during sample time window	Weighted Averaging (for voltage)	≥ 0
Screen_ON	Fraction of time screen was on	Time Fraction Calculations	0-1
Brightness	Screen brightness	Nothing	0-255
Audio_On	Fraction of time audio device was on	Nothing	0-1
Data_Activity	Number of bytes sent/received per second using mobile data service.	Normalization per 1 second	≥ 0
Data_Activity_On	Fraction of time phone sending/receiving data using mobile network	Time Fraction Calculations	0-1
Data_Conn_On	Fraction of time phone is connected to some mobile data network	Time Fraction Calculations	0-1
Phone_Ringing	Fraction of time phone is ringing	Time Fraction Calculations	0-1
Phone_OffHook	Fraction of time phone is in a call	Time Fraction Calculations	0-1

Phone_In_Service	Fraction of time phone is in service	Time Fraction Calculations	0-1
GSM_RSSI	Average Received signal strength indication of the received signal	Weighted Averaging	(-113)-(-51)
NetUnknown	Fraction of time phone is connected to an Unknown network type	Time Fraction Calculations	0-1
NetGPRS	Fraction of time phone is connected to a GPRS network	Time Fraction Calculations	0-1
NetEDGE	Fraction of time phone is connected to an EDGE network	Time Fraction Calculations	0-1
NetUMTS	Fraction of time phone is connected to a UMTS network	Time Fraction Calculations	0-1
NetHSPA	Fraction of time phone is connected to an HSPA network	Time Fraction Calculations	0-1
NetHSPAP	Fraction of time phone is connected to an HSPA+ network	Time Fraction Calculations	0-1
NetHSUPA	Fraction of time phone is connected to an HSUPA network	Time Fraction Calculations	0-1
SMS_Activity	Number of SMS sent/received per second	Normalization per 1 second	≥ 0
SD_Activity	Number of sector read/write per second	Normalization per 1 second	≥ 0
WIFI_ON	Fraction of time WIFI adapter was on	Time Fraction Calculations	0-1
WIFI_Conn_ON	Fraction of time phone is connected to WIFI Access Point	Time Fraction Calculations	0-1
WIFI_Activity	Number of bytes sent/received per second using WIFI adapter.	Normalization per 1 second	≥ 0
WIFI_RSSI	Average received signal strength indication of the received signal	Weighted Averaging	(-100)-(-55)
GPS_On	Fraction of time GPS module is in use	Time Fraction Calculations	0-1
BT_On	Fraction of time Bluetooth	Time Fraction	0-1

	adapter is ON	Calculations	
BT_Discovery	Fraction of time Bluetooth discovery service is running	Time Fraction Calculations	0-1
BT_Device_On	Fraction of time Bluetooth adapter is connected to some device	Time Fraction Calculations	0-1
Bat_Temp	Average battery temperature	Weighted Averaging	any value
Bat_Level	Average battery level	Weighted Averaging	0-100
Core0_Utilization	Utilization of the first CPU Core	Time Fraction Calculations	0-1
Core1_Utilization	Utilization of the second CPU Core	Time Fraction Calculations	0-1
Core0_TimeInFreq(0)	Fraction of time first core operate at frequency 0	Time Fraction Calculations	0-1
Core0_TimeInFreq(1)	Fraction of time first core operate at frequency 1	Time Fraction Calculations	0-1
……	……..		
Core0TimeInFreq(n-1)	Fraction of time first core operate at frequency n-1	Time Fraction Calculations	0-1
Core0TimeInFreq(n)	Fraction of time first core operate at frequency n	Time Fraction Calculations	0-1

3.3 Current Measurement

The power value at any time instant is defined by $P = V \times I$, where V and I are the voltage and current at any instant in time, respectively. We can obtain the voltage value at any time using an Android API; however, the current value reading is not trivial. Next we will describe the current measurement approach followed in this work.

Android APIs do not provide any functionality for current measurement; also almost all smartphones do not have built-in current sensors, which is why most previously reported research depends on external setups to measure current. However, this is not a practical since we need to model smartphone power consumption based on real life usage data and restricting the mobility by requiring external instruments connected to the mobile phone while in use is not practical.

However, the smartphone model selected for this work contains modern Lithium-Ion batteries with a smart battery interface that monitors the charging and discharging process to protect it. It also provides information about battery capacity, current, voltage, and temperature to the power management routine in the operating system. Some smartphone manufactures provide drivers to these interfaces as Linux virtual files to read different parameters of the battery status. Most of the recent SonyEricsson/Sony family of smartphones are provided with this sensor, which is one of the main reasons for choosing the Sony Acro S smartphone as a testing platform for our research.

We target the TI BQ27520 Battery Fuel Gauge IC [**22**]. This gauge resides on the system main board and uses a 400-kHz I^2C™ Interface for connection to microcontroller Port. It is capable of measuring battery level, voltage, current, and temperature. According to the datasheet, we can read the instantaneous current and the average current through the I^2C interface. The value of the average current is updated every second, so sampling the current at 1 Hz is enough to capture the overall average current passed through the gauge and though the whole system.

The Linux kernel used by Android phones that support BQ27520, such as the Acro model, provides a virtual file system driver for the phone through the "/sys/class/power_supply/bq27520/" folder. Reading the file "current_now" will give us the instantaneous current, while reading "current_avg" will gives us the average current. In the logger application we define a timer (Application Timer) to invoke a function every 1 second; in this function we sample the current measurements and audio utilization since these variables do not have OS counters. The average current during a certain time window is calculated using equation (7).

$$Current(Average) = \frac{CurrentSum}{SamplesCount} \quad (7)$$

$$Audio\ Utilization = \frac{AudioOnCount}{SamplesCount} \quad (8)$$

Note that at the end of each sample window all sums and counts are reset.

3.4 The Logger Performance

The logger application is implemented in Java using more than 2300 lines of code. To minimize the overhead of our logger application, we design it such that event-based messages are logged at the time of their occurrence while polling-based information is scheduled to be captured every 15 seconds using a timer interrupt. However, it seems when there is no activity, the OS on its own chooses to elongate the sampling period set for the counter. In Figure 12, we plot the Cumulative Distribution Function (CDF) of sample periods of a set of 46517 data samples. We find that 40% of the data samples have sample periods equal or less than 15 seconds, while 70% have sample periods equal or less than 17 seconds, and 94% have sample periods less than or equal to 100 seconds. We hypothesize that variations in sample periods are due to OS scheduling of our timer interrupts. When the smartphone is in sleep mode, or in heavy use mode, the OS may delay timer interrupts to save resources.

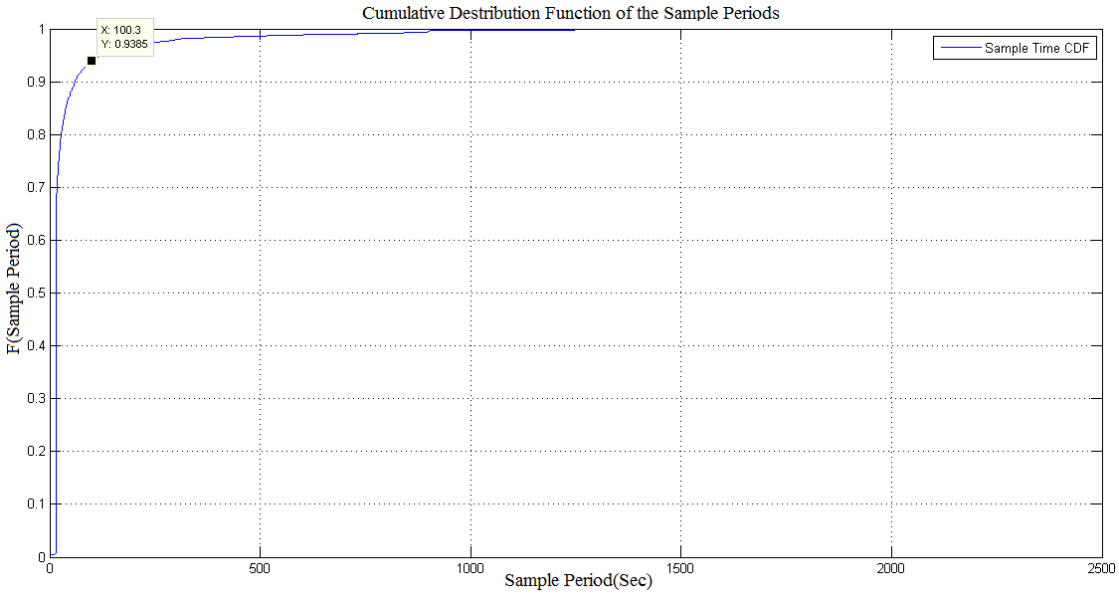


Figure 12: Cumulative distribution function of data sample periods

We also studied the total CPU time consumed by the logger application. The /proc/[PID] Linux virtual file provides all execution information of the logger application. We extract CPU time consumed by the application per sample in Jiffies time units. Knowing that the HZ constant in our Linux kernel is 100 (obtained using SYSTEM_CLK_TCK constant), the time unit duration will be 0.01 second; therefore,

multiplying time units by time unit duration yields the total CPU time consumed by our application in seconds. Figure 13 shows the Cumulative Distribution function of the CPU time taken by our logger application per sample.

From Figure 13, we note that the logger CPU time is less than or equal to 0.08 seconds per sample for approximately 73.21% of the samples, and less than or equal to 0.19 seconds per sample for about 99.55% of the samples. We also analyze Figure 14, the overhead; in terms of CPU time, the average logger overhead (logger CPU Time/Total CPU Time) is 0.62%. From Figure 14, we also conclude that 78.14% of the samples have logger overhead less than or equal to 0.6667%, and 99.5% of the samples have logger overhead less than or equal to 5%.

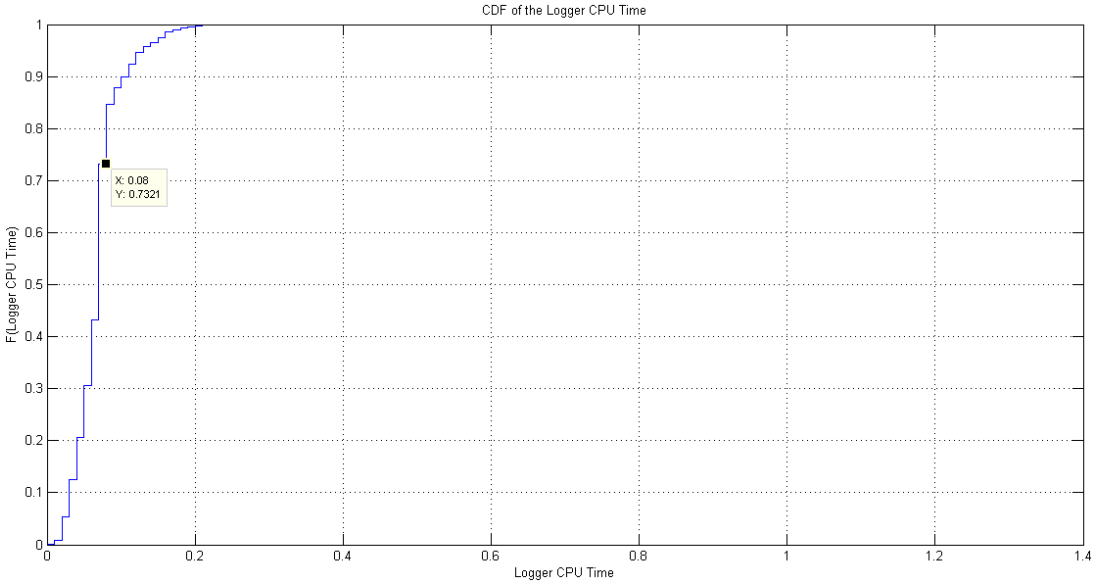


Figure 13: CDF of the Logger CPU Time

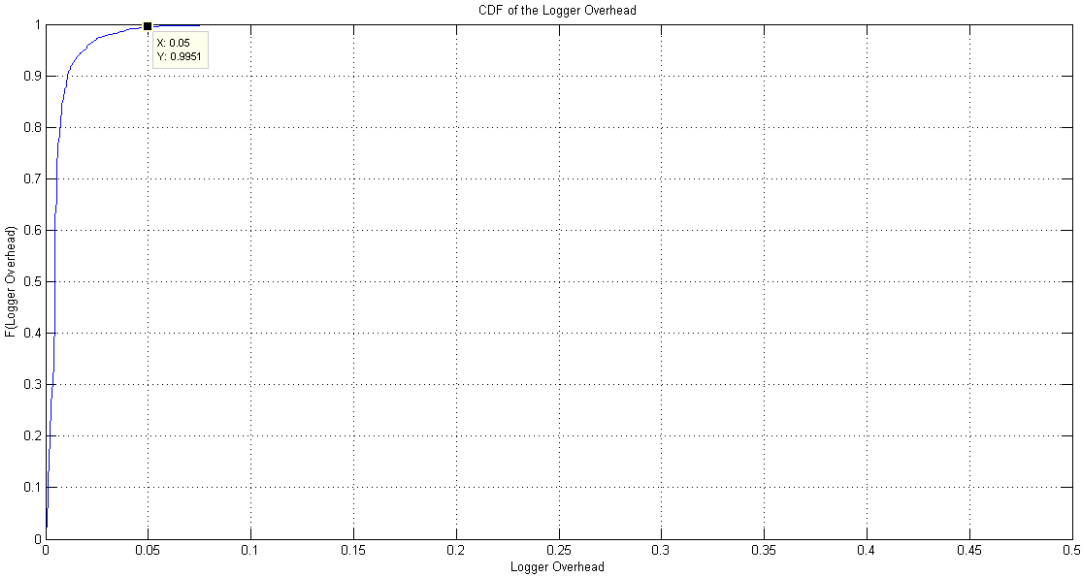


Figure 14: CDF of the Logger Overhead

Chapter 4
Power Regression-Based Modeling

Regression analysis is a statistical technique used to investigate and model the relationship between two or more variables. In a simple relationship there are two variables, an independent variable (also called the predictor variable), and a dependent variable. In a multiple relationship (multiple regressions) two or more independent variables are used to predict one dependent variable [**23**].

In general, the dependent variable or response Y may be related to k independent or regressor variables. The model

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots\ldots\ldots\ldots\ldots + \beta_k x_k + \varepsilon \quad (9)$$

is called a multiple linear regression model with k regressor or predictor variables.

The parameters β_j, j=0, 1, …, k are called the regression coefficients and ε is a random error term. These models can be used as approximating functions where the exact functional relationship between Y and $x_1, x_2, \ldots, x_k$ is unknown. However, over certain ranges of the independent variables the linear regression model provides an adequate approximation [**23**].

References [**24**] [**25**] [**26**] [**27**] are examples of recent engineering research that apply regression principles in modeling.

The aim of regression analysis in this work is to determine simple and sufficiently accurate models for predicting smartphones' user-specific energy requirements, as well as to develop a generalized power model. Both models are data-driven since the generation of the various parameters of the models is obtained using the samples logged during users' usage of their smartphones.

4.1 Predictors Selection

In developing a regression-based model, a critical step is to determine if all predictors are required or whether a subset of predictors would suffice to develop a reliable model. The predictor variable should be highly correlated with the criterion variable Y, and have a low correlation with other predictor variables. The elimination of unnecessary predictors will lead to a simplification in the data-gathering phase and an enhancement to the model; it also eases the interpretation of results. In this work and in an effort to reduce the number of predictors, *Heat Maps* showing the correlations between the predictors are first generated.

Table 5 of Chapter 3 includes all the parameters extracted during smartphone activity. The parameters are grouped together using functionality as a criterion and then corresponding heat maps are used to eliminate some of the parameters from the predictors subset.

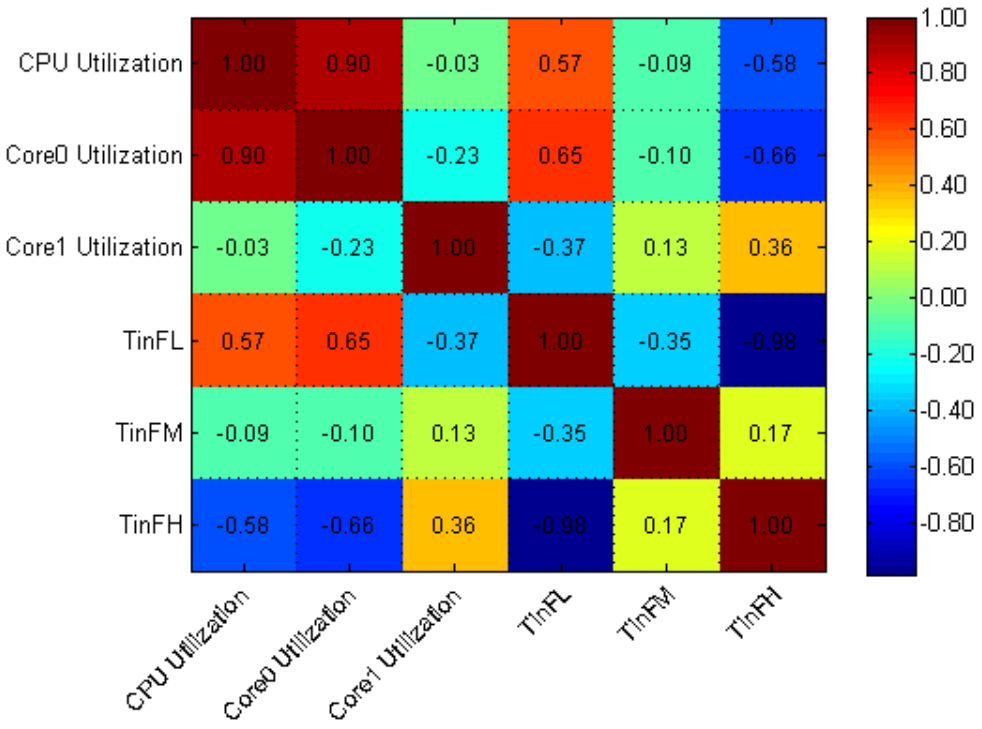


Figure 15: Heat Map relating CPU Parameters

The heat map of Figure 15 shows the relationship between the various parameters directly related to the CPU. Each predictor is completely correlated with itself (the brown diagonal). It is clear that CPU utilization (average CPU utilization) is

highly correlated with Core0 utilization (utilization of the first core), which means CPU utilization is highly dependent on Core0 utilization and vice versa. Therefore, including one of these predictors in the regression model would suffice.

In the smartphone family targeted in the course of this work, each processor core supports up to 23 different frequencies using on-demand dynamic frequency scaling algorithms. It is not practical to include all of them in the power model, and hence, we categorize the frequency range into 3 groups.

TinFL is the fraction of time the CPU spent operating at frequencies that belong to the low-frequency-range category. Similarly, TinFM and TinFH are the fractions of time spent in the middle and high frequencies respectively.

From Figure 15, TinFL is almost completely negatively correlated with TinFH, This implies that our CPU tends to utilize either low frequencies or high frequencies. These two variables are negatively correlated since the CPU can operate at one frequency at a time, so if the low frequencies are heavily used then the high frequencies will be lightly used and vice versa.

We also noted that TinFL, TinFH, CPU Utilization, Core0 Utilization, and Core1 Utilization are highly correlated with each other. This seems logical since all of them are trying to measure the CPU performance in different ways. The aforementioned predictors' values are calculated using equations 10 to 16:

$$CPU\ Utilization = \frac{Total\ CPU\ time - IDLE\ time}{Total\ CPU\ Time} \tag{10}$$

$$Core0\ Utilization = \frac{Total\ first\ Core\ time - Total\ IDLE\ time\ for\ the\ first\ core}{Total\ first\ core\ Time} \tag{11}$$

$$Core1\ Utilization = \frac{Total\ second\ Core\ time - Total\ IDLE\ time\ for\ the\ second\ core}{Total\ second\ core\ Time} \tag{12}$$

$$TinFL = \frac{Total\ Time\ CPU\ operate\ at\ the\ low\ frequency\ range}{Total\ CPU\ time} \tag{13}$$

$$TinFM = \frac{Total\ Time\ CPU\ operata\ at\ the\ middle\ frequency\ range}{Total\ CPU\ time} \tag{14}$$

$$TinFH = \frac{Total\ Time\ CPU\ operate\ at\ the\ high\ frequeny\ range}{Total\ CPU\ time} \tag{15}$$

$$TinFL + TinFM + TinFH = 1 \quad (16)$$

All previous equations are actually correlated with each other. The denominator for CPU Utilization, TinFl, TinFM, and TinFH is the same; the summation of the Core0 Utilization and Core1 Utilization denominator is equal to Total CPU time. The question now is how to model the CPU usage in terms of CPU utilization and frequency usage with low correlation.

We dropped TinFL predictors since they are equal to $1 - TinFM - TinFH$ and include CPU Utilization. We removed Core0 Utilization and Core1 Utilization to make our equation applicable to any phone (not only dual core). TinFM, TinFH, and CPU Utilization is still correlated, but we need some logical relation between them. TinFM and TinFH is a measure of the fraction of time the CPU operate at middle and high frequency ranges respectively, and this time is not totally utilized. CPU utilization is a measure of the fraction of time the CPU is doing useful work. Multiplying CPU Utilization by TinFM and TinFH yields the fraction of time the CPU is doing useful work at middle and high frequencies respectively (MF Utilization and HF Utilization). The equations below highlight these computations.

$$MF\ Utilization = TinFM \times CPU\ Utilization \quad (17)$$

$$HF\ Utilization = TinFH \times CPU\ Utilization \quad (18)$$

The correlation between MF Utilization and HF Utilization is only 0.17 which is acceptable.

In our logger application we log information about all mobile networks, which include GPRS, EDGE, UMTS, HSDPA, HSPA, HSPA+, HSUPA and UNKNOWN network when the mobile is not connected to any network or is connected to some unknown network. From the logged data, we observe that only GPRS, EDGE, and HSDPA are in use in the UAE.

Figure 16 shows that HSDPA network type predictors are negatively correlated with all other networks. This seems logical since the average values for all network type predictors are 0.0975, 0.0046, 0.0453, and 0.8525 respectively. It is clear that HSDPA network is the dominant network and other networks are used only

when the phone not able to connect to it. The summation of all network predictors' usage in a sample is 1; hence they are linearly dependent, and we should get rid of one of them. The first choice could be removing the HSDPA network predictor since all other network predictors are negatively correlated with it. However, we prefer to keep it since it's the dominant network type. Thus, we removed the Unknown network type to break the linear dependency between predictors. $NetUnknown$ predictors can be computed using equation 19:

$$NetUnknown = 1 - NetEDGE - NetGPRS - NetHSDPA \quad (19)$$

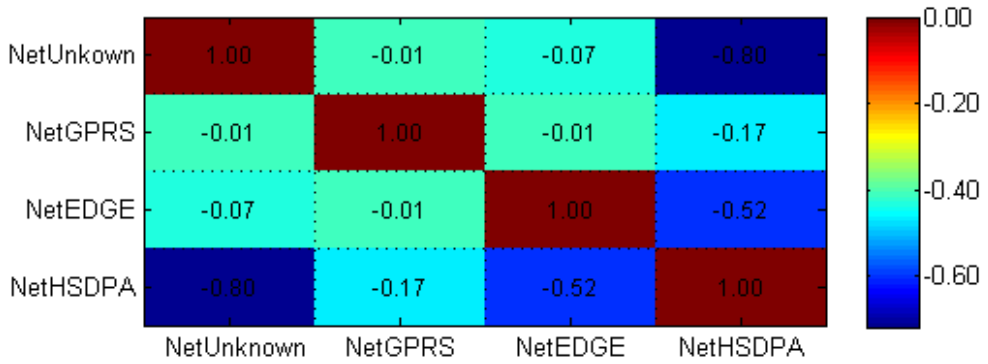


Figure 16: Heat Map for Network Type Predictors

To model the screen, we have Screen On and Screen Brightness predictors. We ran a correlation study between them and they were not linearly dependent (correlation coefficient was 0.05). To model phone calls we have Call_Ringing and Call_OffHook predictors which are linearly independent. We have SD_Traffic, Audio_On, and GPS_ON to model SD Card, audio, and GPS, respectively.

Nowadays, most students are using Internet-based services, such as Whatsapp, and Google Talk, for text messaging, so SMS Activity is very low. We logged only 64 non-zero SMS activities in our data set. This number of samples is not sufficient for modeling this activity, so we did not include SMS Activity as a predictor in our model. Also, Bluetooth is rarely used in our dataset so we did not include any Bluetooth predictors in our dataset.

For Wi-Fi communication we have WIFIOn, WIFIConON, WIFIActivity and WIFI RSSI. Figure 17 shows the heat map for all WIFI predictors. It's clear that

WIFIOn and WIFIConOn are highly correlated (correlation coefficient is 0.75). This means, in our dataset, whenever Wi-Fi is ON, a connection is established to some Wi-Fi hotspot. In our study, since we target university students and Wi-Fi is available everywhere on campus, the phone is very likely to be connected to the Wi-Fi as soon as Wi-Fi adapter is turned on. Because we should get rid of one of these predictors, we chose to remove the WIFIConOn predictor. Also, WIFIRSSI is linearly dependent on WIFIOn, so we removed it as well.

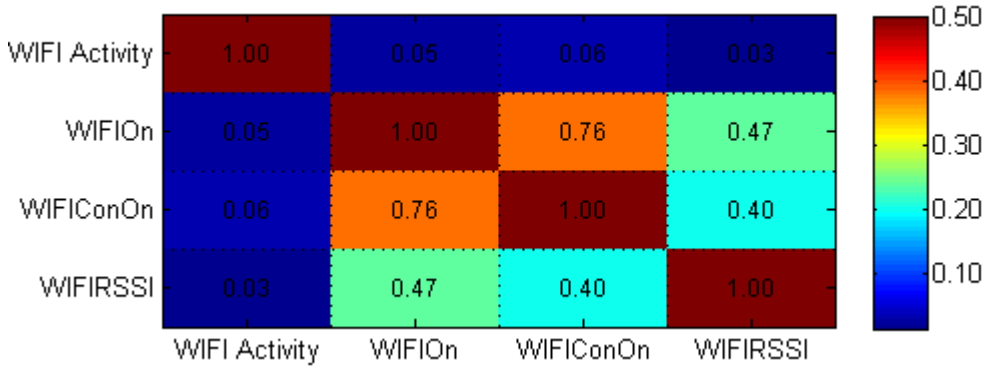


Figure 17: Heat Map of the WIFI predictors

For modeling mobile data communications, we have Data Activity, Data Activity On, DataConOn and GSMRSSI predictors. Figure 18 shows the heat map for the previously mentioned predictors. It is clear that Data Activity On and DataConOn are correlated. Logically, Data Activity and Data Activity On are also correlated, since there will be no Data Activity unless Data Activity On is greater than 0. We prefer to remove Data Activity On since it is correlated with the two other predictors.

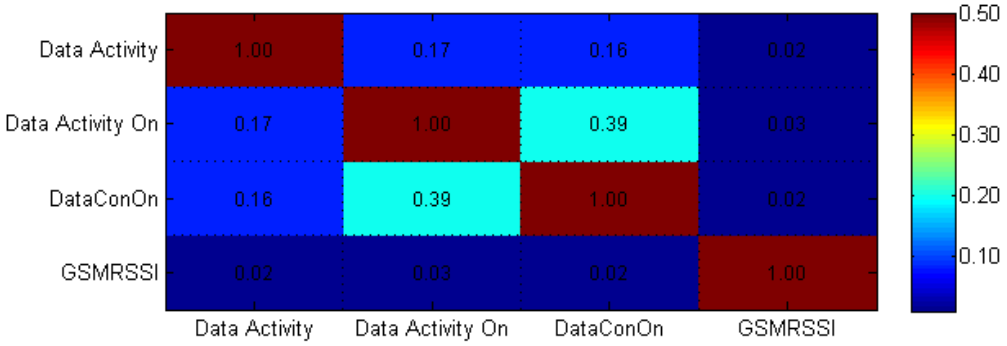


Figure 18: Heat Map for Mobile Data Communication Predictors

After excluding unnecessary predictors, the subset of predictors selected for power model generation is given in Table 6. The Generation Method describes the technique used to obtain the value of the predictor, either using Android broadcast Actions/Receivers or polling OS counters at a low rate, or polling device information at a high rate.

Table 6: Predictors Selected for Model Development

Predictor Name	Description	Generation Method	Range
MF Utilization	CPU utilization while operating at the Median frequency rang	Variables Polling with OS counters	0-1
HF Utilization	CPU utilization while operating at the high frequency range	Variables Polling with OS counters	0-1
Screen ON	Fraction of time Screen was on	Broadcast receivers	0-1
Screen Brightness	Average Screen brightness	Variables Polling	20-255
Call Ringing	Fraction of time smartphone was ringing	Broadcast receivers	0-1
Call Off Hook	Fraction of time smartphone was in call	Broadcast receivers	0-1
Data On	Fraction of time smartphone was connected to some mobile network	Broadcast receivers	0-1
Data Traffic	Average number of bytes sent/received through mobile network per second	Variables Polling with OS counters	≥ 0
WIFI ON	Fraction of time phone is connected to some WIFI network	Broadcast receivers	0-1
WIFI Traffic	Average number of bytes sent/received through WIFI interface per second	Variables Polling with OS counters	≥ 0
SD Traffic	Average number of sectors read/written per second	Variables Polling with OS Counters	≥ 0
Audio ON	Fraction of time Audio device was active	Broadcast receivers	0-1
GSM RSSI	Average mobile Received Signal Strength Indication	Broadcast receivers	(-113) – (-48) dBm
NET HSDPA	Fraction of time mobile connected to HSDPA network (3G)	Broadcast receivers	0-1

NET EDGE	Fraction of time mobile connected to EDGE network (2.5 G)	Broadcast receivers	0-1
NETGPRS	Fraction of time mobile connected to GPRS network (2G)	Broadcast receivers	0-1
GPS ON	Fraction of time GPS adapter is ON	Broadcast receivers	0-1

4.2 Linear Regression in Context

In the previous section we identified a set of 17 predictors to model our systems. In this section we apply regression analysis to develop a power model for our smartphone. We can use an ordinary least squares (OLS), or total least squares (TLS) regression method to model power consumption of smartphone. In [**28**] the authors showed that OLS is better suited than TLS for power modeling problems, as TLS assumes that all inputs have errors and need to be adjusted. In our case not all inputs have errors and the relative errors of the inputs are not known in real time. Hence we opt to apply OLS in the work discussed here.

For a single sample i, if the measured value of the $Predictor_j$ is $A_{i,j}$ then the power contribution of $Predictor_j$ to the total power of the sample is given by:

$$p_{i,j} = A_{i,j}.\beta_j \tag{20}$$

where β_j is the regression coefficient for $Predictor_j$. The total power of sample i with n predictors is given by:

$$P_i = k + \left(p_{i,0} + p_{i,1} + \cdots + p_{i,n}\right) \tag{21}$$

$$P_i = k + \Big(\left(A_{i,0}.\beta_0\right) + \left(A_{i,1}.\beta_1\right) + \cdots + \left(A_{i.n}.\beta_n\right)\Big) \tag{22}$$

where k is a constant that includes power not attributable to any of our predictors. Assuming $x_i = \left(A_{i,0}, A_{i,1} \ldots A_{i,n}\right)$ for each sample i, and $b = (\beta_0, \beta_1 \ldots \beta_n)$, we can reduce equation 22 to:

$$P_i = k + x_i.b \tag{23}$$

For a dataset of m samples, the general mode is given by:

$$\begin{pmatrix} P_0 \\ P_1 \\ \vdots \\ P_m \end{pmatrix} = k.\begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} + \begin{bmatrix} A_{0,0} & \cdots & A_{0,n} \\ A_{1,0} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,0} & \cdots & A_{m,n} \end{bmatrix}\begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{pmatrix} \quad (24)$$

Letting $P = \begin{pmatrix} P_0 \\ P_1 \\ \vdots \\ P_m \end{pmatrix}, X = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n} \\ A_{1,0} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,0} & \cdots & A_{m,n} \end{bmatrix}$ and $e = (1 \ldots 1)^T$ yields:

$$P = k.e + X\beta \quad (25)$$

Values of k and β can be calculated using the *LinearModel.fit* function in MATLAB. Once calculated, the power for any sample of the system measurements x_i can be estimated using Equation 23. The power contributed by each system predictor can be estimated using Equation 20.

The average power for a dataset of m samples can be estimated by:

$$P_{dataset} = \frac{1}{m}\sum_{i=1}^{m} P_i \quad (26)$$

Given the duration of the sample i to be T_i, the energy consumed during sample i can be estimated using:

$$Energy_i = P_i \times T_i \quad (27)$$

The total energy consumed by a dataset of m samples can be estimated by:

$$Energy_{total} = \sum_{i=1}^{m} Energy_i \quad (28)$$

The squared error in sample i is given by:

$$E_i = (\overline{P_i} - P_i)^2 \quad (29)$$

where $\overline{P_i}$ is the measured power for sample i. The mean squared error of dataset of m samples is given by:

$$MSE = \frac{1}{m} \times \sum_{i=1}^{m} E_i \quad (30)$$

The root mean squared error (RMSE) is given by:

$$RMSE = \sqrt{MSE} \quad (31)$$

In the following two sections we discuss the development of the power models per user and per smartphone type.

4.2.1 User-Level Power Modeling

Previously, we concluded that 17 predictors are sufficient to build the regression model; these predictors are used to model the different components of the smartphone. In the course of the users' data gathering and extraction, we noted that some users do not use or exercise the functionality of all components comprising the smartphone. For example, some of them did not use the GPS at all, and some did not have a data package subscription with an Internet Service Provider, and hence had no mobile data activity. To overcome this limitation while developing the per user models, we remove any predictors that are not used. We check the value of variance of the all values of the predictors. If the value is zero, we remove it from the model. This operation is performed automatically using a program developed for this work within the MATLAB environment.

To generate user-based regression models, we divide the dataset for each user into a training dataset of 85% of samples and a testing dataset of 15% of samples.

Table 7: User-level models' performance parameters

User	Mobile Model	Duration (days)	Number of Predictors	R^2 (Coefficient of determination)	RMSE (Root Mean Square Error)
User1	Xperia Acro S	12.849	16	0.82	0.2220
User2	Xperia Acro S	7.008	13	0.823	0.2780
User3	Xperia Acro S	10.159	13	0.866	0.0965
User4	Xperia Acro S	14.442	14	0.925	0.1916
User5	Xperia Acro S	11.744	14	0.826	0.1124
User6	Xperia Acro S	9.123	17	0.899	0.2279
User7	Xperia Acro S	9.513	15	0.905	0.2700
User8	Xperia Acro S	8.516	13	0.836	0.2091
User9	Xperia S	15.849	17	0.879	0.2221
User10	Xperia S	39.787	14	0.781	0.2541

Figure 19 contains plots of the first 6 users' regression model performance. We used the MATLAB *plotregression* function to plot these figures; this function plots the linear relationship between output, which is the estimated power P_i , for each sample i and the target which is the measured power $\overline{P_i}$ for each sample i. The ideal model should have all points located on the diagonal line Y= T. The blue line (Fit) is the curve fitting of the output with respect to the target; the closer this line to the diagonal line, the more accurate our model is. In the y-axis we have the regression equation $Output = Slope \times Target + Offset$. In ideal cases $Slope$ should be 1 and the offset should be zero, which means $Output = Target$.

We note that the models for user 4 and user 6 fit the measured power very well; the fit line (blue) is very close to the diagonal line (dashed). The models of users 1 and 2 have the lowest R^2 since they have long tails; it seems that our modeling technique did not perform well with the high power samples. Users 1 and 2 have a relatively large number of high power samples, which is why the tail appears. User 3 generates a very low number of samples despite this user's long usage duration (10 days). It seems that he/she uses the Android-based mobile phone as secondary mobile, since most of the time the phone was idle. User 5's power usage is between low and medium, meaning that either the phone is idle or is used normally without any computation-hungry applications.

Table 7 shows the performance of the user based regression models. The second column is the mobile model, the third column is the logging duration in days, the fourth column is the number of predictors used to develop the model (other predictors do not change across so are removed), the fifth column is the R-squared (coefficient of determination), and the last column is the root mean squared error of the testing dataset.

The closer the value of R^2 to 1, the more accurate the model is. Our models' R^2 are between 0.781 and 0.9265 which are relatively close to 1. The root mean squared errors of our models are between 0.0965 and 0.2780. The closer this number is to 0, the more accurate the model.

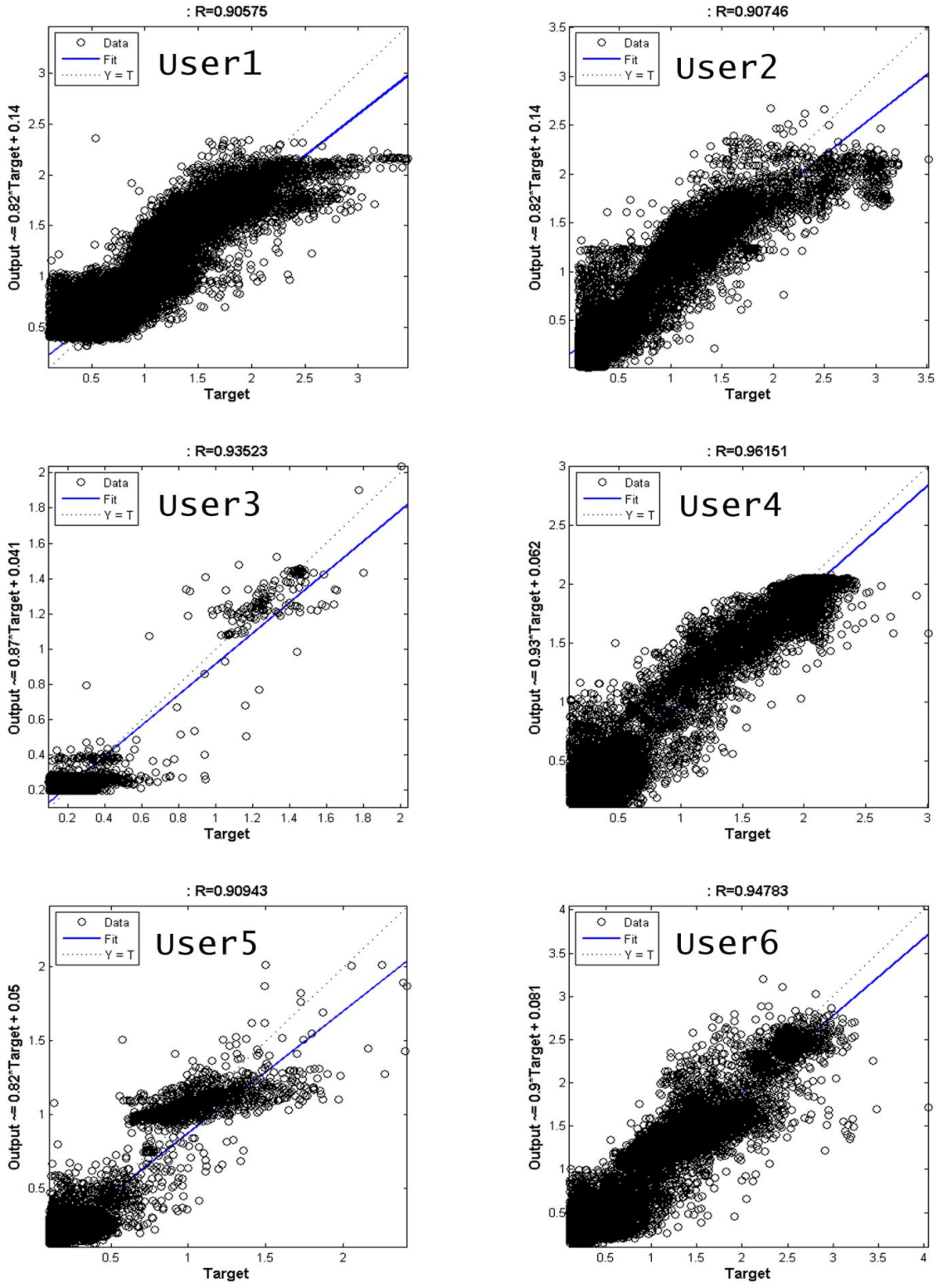


Figure 19: Regression plots for the first 6 users

4.2.2 Device Level Power modeling

In this section we build a generalized power model based on the device and presumably applicable to all users. We have included two devices in our research: the Sony Xperia Acro S and Sony Xperia S smartphones. For the Xperia Acro S we collected data from 8 distinct users with log intervals of at least one week per user. For the Xperia S we collected data from only two distinct users. This number of user may be not sufficient to model the power for the device; therefore, we attempted to build a power model for the Sony Xperia Acro S device only.

We use MATLAB's *LinearModel* toolbox to generate the power model for the device. We use 85% of our data set as a training set and 15% as a testing set to make sure that we do not have over-fitting. The generated model fit our data set with $R^2 = 0.835$ and RMSE = 0.2570 for the training dataset and 0.2560 for the validation dataset. We use root mean square error (RMSE) as a performance measure for our model.

Table 8 shows the predictor estimates for the proposed regression model, where the constant term (k) is equal to 0.1262. Equations 20 to 31 are all applied to this regression model to estimate the power per sample and per dataset along with error estimation. The number of predictors in this case is $m = 17$.

Table 8: Regression Model Predictor Coefficients

j	**Predictor j**	β_j	**Range of Predictor j**
1	MF Utilization	0.7811	0-1
2	HF Utilization	0.81541	0-1
3	Screen ON	0.83291	0-1
4	Screen Brightness	0.0015905	20-255
5	Call Ringing	0.35411	0-1
6	Call Off Hook	0.52635	0-1
7	Data On	0.21864	0-1
8	Data Traffic	2.7274×10^{-6}	≥ 0
9	WIFI ON	0.017464	0-1
10	WIFI Traffic	2.7686×10^{-7}	≥ 0
11	SD Traffic	9.2416×10^{-6}	≥ 0
12	Audio ON	0.2654	0-1
13	GPS ON	0.31748	0-1

14	GSM RSSI	0.0011159	(-113) – (-48) dBm
15	NET HSDPA	0.039442	0-1
16	NET EDGE	0.031213	0-1
17	NET GPRS	0.15199	0-1

We noted that the major energy consumers are display (Screen_ON) followed by high frequency utilization and median frequency utilization. We also noted that smartphones consume more energy in the calling state than in the ringing state. For data communication, mobile data networks consume much more energy than Wi-Fi networks; in addition, 2G networks (GPRS) consume much more power than newer networks (EDGE and HSDPA).

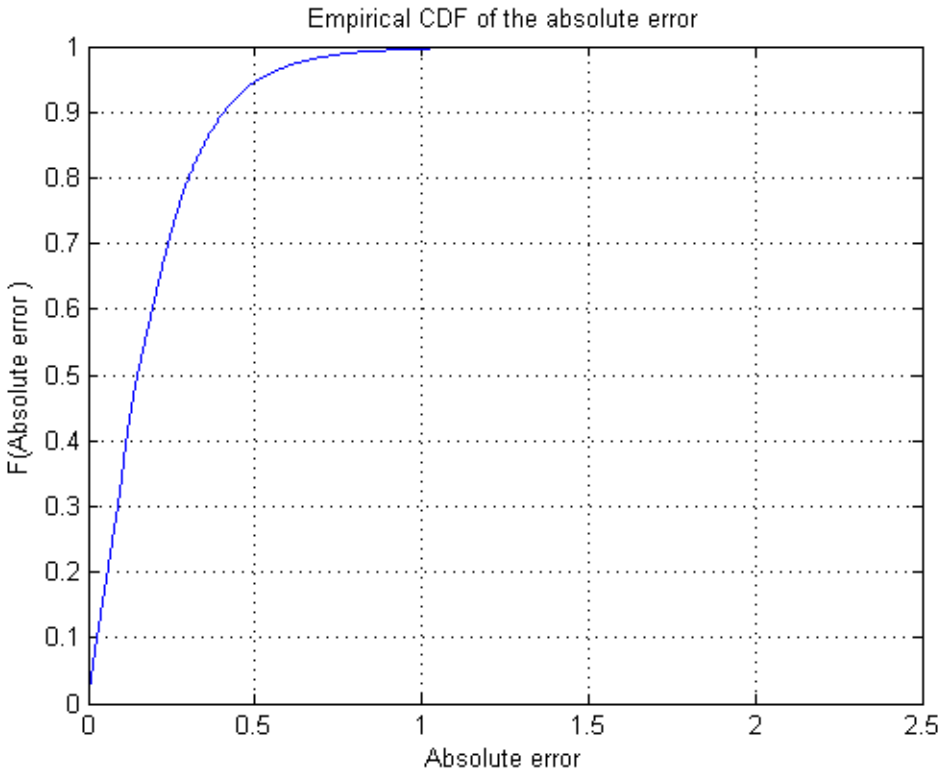


Figure 20: Empirical CDF of absolute error of our dataset

From Figure 20 we notice that 80% of samples have errors less than or equal to 0.3011 watts, 95% of samples have errors less than or equal to 0.514 watt, and 99% of samples have errors less than or equal to 0.7842 watts. The error histogram is shown in Figure 21, and it is clear that it follows a normal distribution.

In Figure 22 we use the MATLAB function *plotregression* to evaluate our regression model's performance. We find that the correlation coefficient between them (R) is 0.91379, which is relatively close to the unity.

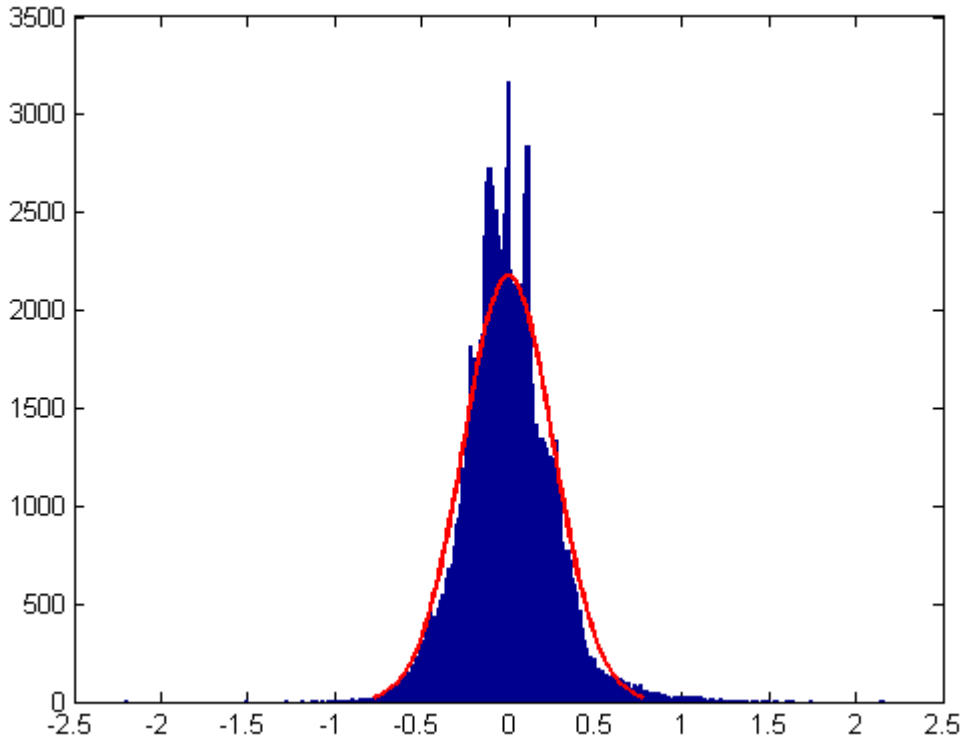


Figure 21: Error histogram the results of the regression based model

In the ideal regression model, all points should be located at the dashed line, which means the estimated power is equal to the measured power. We cannot build an ideal model, but we can try to build a model close to it. The more points there are close to the dashed line, the more accurate the regression model is. It's clear that our model has its lowest accuracy in both extremes values, from 0 to 0.5 watts and from 2.25 to 3.5 watts. The accuracy in the range from 2.25 to 3.5 is acceptable.

The cause of the error in the low power values could be the result of idle states, or sleep states, of the mobile system. We did not log information related to the power state of the system. In the next version of the logger we intend to use *wackelooks* for additional data collection. Mobiles tend to use a constant amount of power while in the sleep or idle state, so in these states there is no need of regression. Estimating the average power will be more than enough.

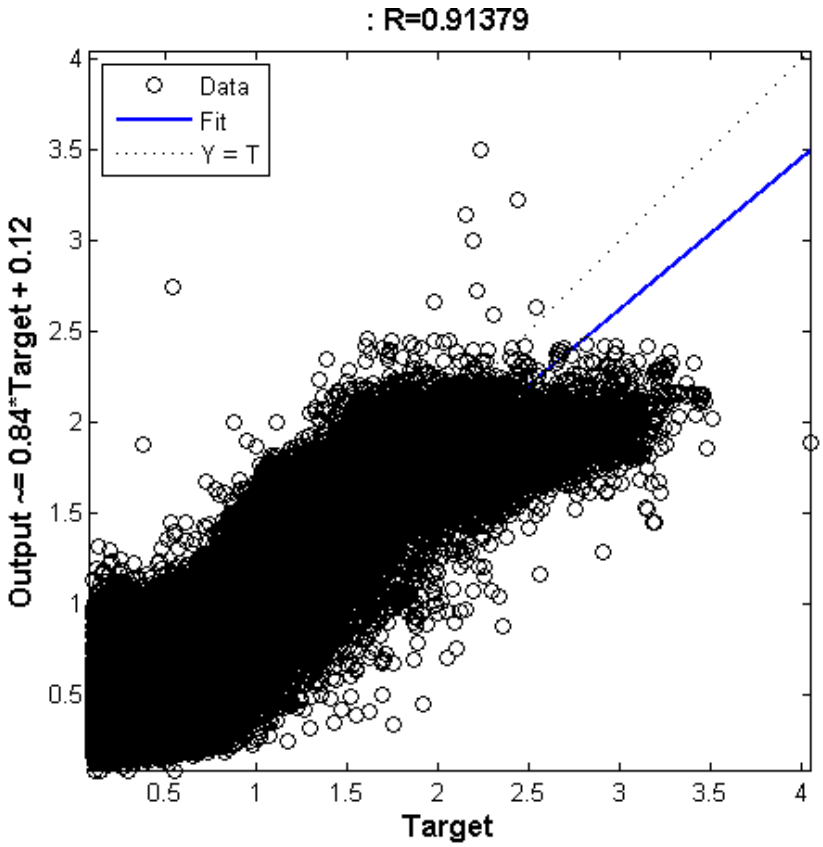


Figure 22: Regression model performance plot

The cause of the error in the high power region could be attributed to the low number of samples that have high power. We relied on power samples collected from users to develop our model. We did not force the use of all predictors' values as in lab settings, so data collected from the users may not have contained enough high power samples for our model to converge.

We test the device level model with all users of the Sony Xperia Acro S mobile phone. Figure 23 shows the regression performance plots for the first 6 users. Users 3 and 5 have the lowest accuracy since they have the lowest number of power samples; for other users with a higher number of samples the model performance is acceptable.

Table 9: Regression performance for mobile users using user-level and device-level modeling

user	R (user level)	R (device level)	RMSE (user level)	RMSE (device level)	$\frac{R(device\ level)}{R(\ user\ level)}$
User 1	0.90575	0.88582	0.2220	0.2645	0.978
User 2	0.90746	0.88867	0.2780	0.3391	0.979
User 3	0.93523	0.62995	0.0965	0.3824	0.674
User 4	0.96151	0.93348	0.1916	0.266	0.971
User 5	0.90943	0.77026	0.1124	0.2147	0.847
User 6	0.94783	0.91824	0.2279	0.289	0.969
User 7	0.95097	0.94076	0.2700	0.3891	0.989
User8	0.91312	0.88319	0.2091	0.2516	0.967

Table 9 shows the performance of the power model of mobile phone users when using user-level power modeling and device-level power modeling. It is clear that for all users, user-level power modeling has higher performance than device-level modeling. Also, we note that for users 3 and 5 where the number of samples is limited we have the lowest performance of device-level modeling compared to user-level modeling $\left(\frac{R_{device\ level}}{R_{user\ level}}\right)$. For other users with sufficient samples, the device-level model performance is relatively good.

Results indicate that the RMSE of the user level models is always lower than device level models, so we can conclude that user-level modeling is more accurate than device-level modeling. Device-level models are still useful since we model the power consumption of a device without relying on the particular usage pattern of individual users and the modeling aspects are approached from a global perspective.

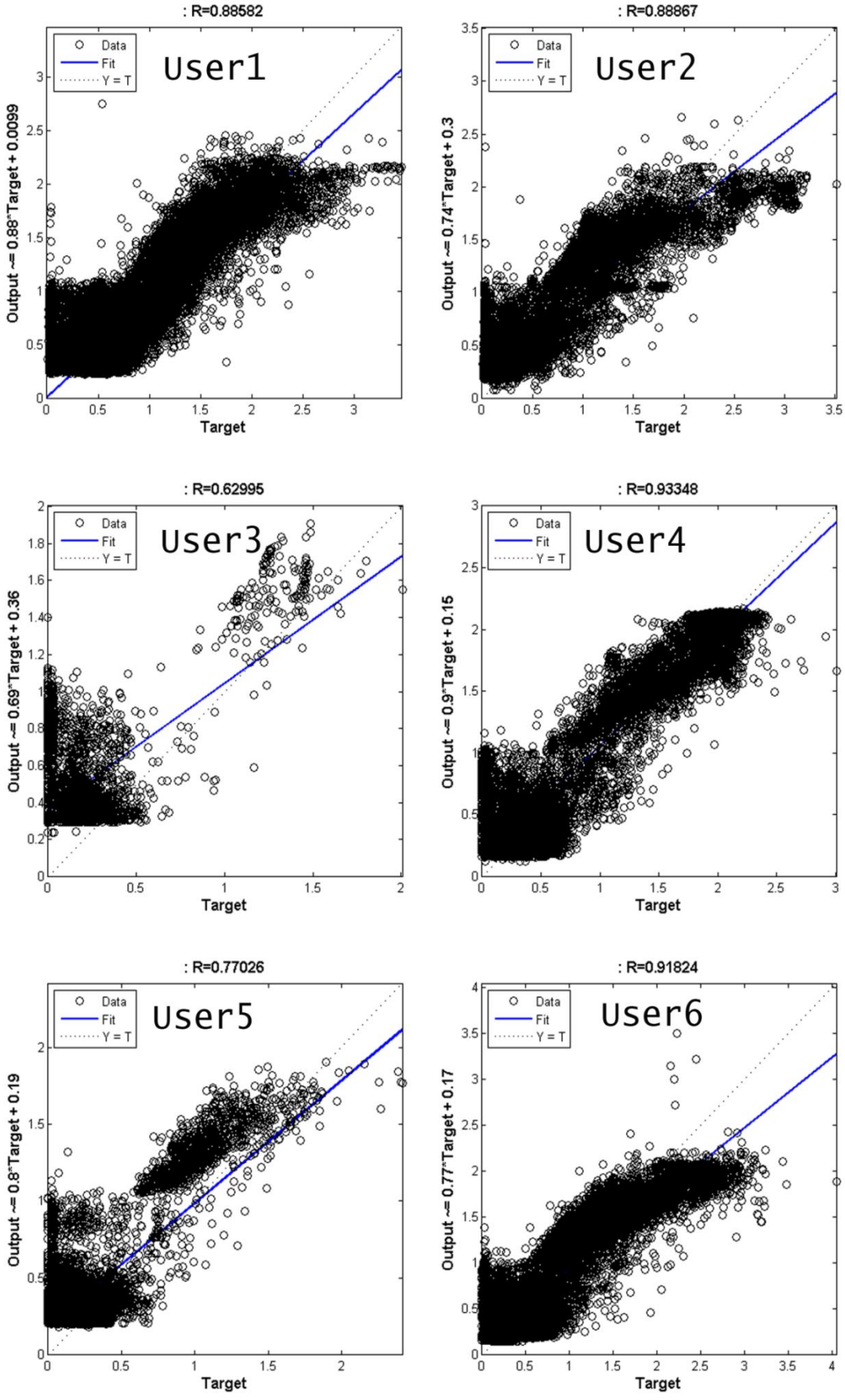


Figure 23: User-level regression performance when using the device-level model

Chapter 5
Neural Network based Power Modeling

An artificial neural network (ANN), often simply referred to as a neural network (NN), is a mathematical model inspired by biological neural networks. A neural network consists of an interconnected group of artificial neurons that processes information using a connectionist approach to computation [**29**].

The origin of the modern artificial neural networks science was the work published by Warren McCulloch and Walter Pitts [**30**], who showed that artificial neural networks could, in principle, compute an arithmetic or logical function.

The elementary element of the ANN is the artificial neuron. Figure 24 shows the artificial neuron.

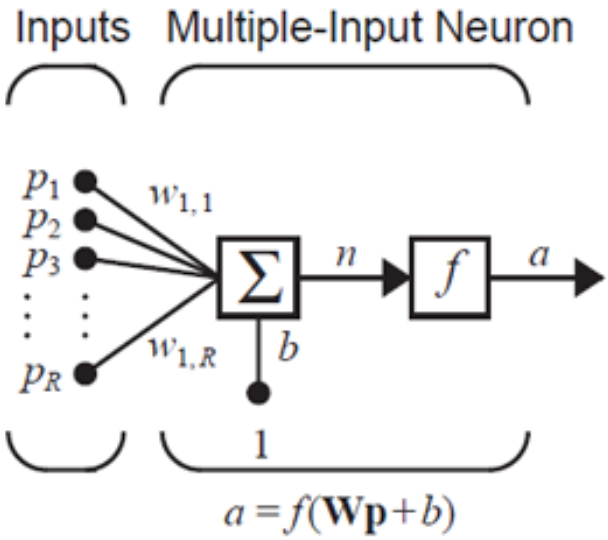


Figure 24: Artificial Neuron [31**]**

The individual inputs $p_1, p_2, \dots, p_R$ are each weighted by corresponding elements $w_{1,1}, w_{1,2}, \dots, w_{1,R}$ of the weight matrix $\boldsymbol{W}$. The net input n can be calculated by:

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b \qquad (32)$$

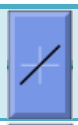
where b is the bias of the neuron. Equation 32 can be written in a matrix form as:

$$n = \boldsymbol{W}\boldsymbol{p} + b \qquad (33)$$

Now, the neuron output can be written as $a = f(\boldsymbol{W}\boldsymbol{p} + b)$ where f is the transfer function.

The transfer function f may be a linear or non-linear function of the net input n. Our power model generation is considered as a fitting problem. Usually three types of transfer functions are used for these kind of problems; others are usually used for classification problems. Table 10 lists these three transfer functions.

Table 10: Transfer functions

Name	Functionality	Icon	MATLAB Name
Linear	$a = n$		**purelin**
Log-Sigmoid	$a = \frac{1}{1+e^{-n}}$		**logisg**
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$		**tansig**

The selection of the transfer function and the number of inputs will define the structure of the neuron; the process of choosing weights and bias to generate the right output is called the training of the neuron.

A single neuron has very limited modeling capabilities. Usually, multiple layers of many neurons are used for modeling. The number of layers and neurons per layer and the type of connections between them defines the neural network architecture.

In recent years, neural networks have been applied to model and solve various problems in the engineering field [**32**] [**33**]. In the following section, we describe some of the widely used neural network architectures that can be used to model our smartphone usage.

5.1 Neural network architectures

In this work, we will use MATLAN Neural Network Toolbox. It supports many neural network architectures. In the following subsections we will briefly describe some neural network architectures; most of the definitions in the following subsections are taken from MATLAB documentations [**34**].

5.1.1 Feed-forward back-propagation (FFBP)

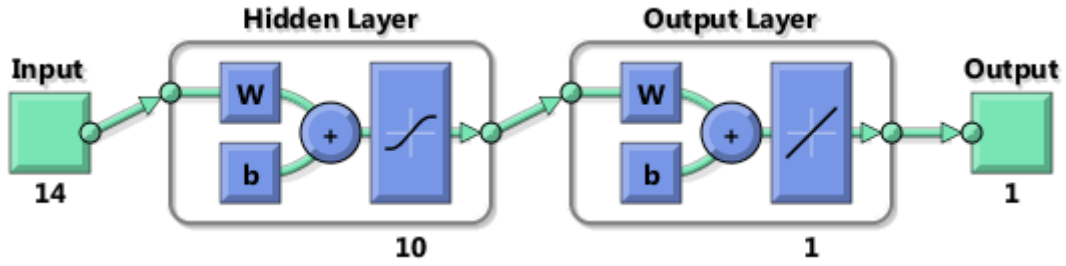


Figure 25: An example of the feed-forward back-propagation neural network.

The feed-forward neural network was the first and simplest type of artificial neural network. The connections between units of the feed-forward neural network do not form a directed cycle. This is different from recurrent neural networks.

Figure 25 shows an example of this network (14 inputs, 1 hidden layer with 10 neurons, and 1 output layer with 1 neuron). It is clear that the information moves in only one direction: forward, from the input nodes, through the hidden nodes, to the output nodes. There are no cycles or loops in the network [**35**]. Given enough hidden neurons, a two or more layer feed-forward network can learn any finite input-output relationship relatively well.

5.1.2 Cascade feed-forward back-propagation (CSFFBP)

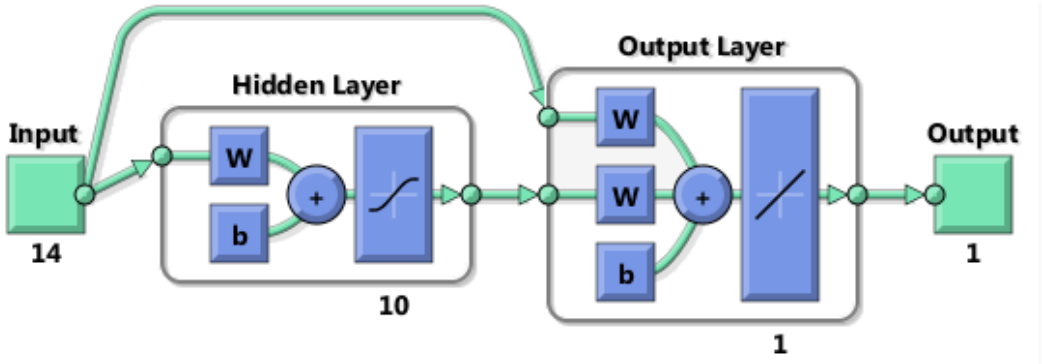


Figure 26: An example of a cascaded feed-forward back propagation network

This network is similar to feed-forward networks, but includes a connection from the input and every previous layer to following layers. Figure 26 shows an example of this network architecture (14 inputs, 1 hidden layer with 10 neurons, and one output layer with 1 neuron). Like the FFBP network, a two or more layer cascaded feed-forward network can learn any finite input-output relationship relatively well, given enough hidden neurons.

5.1.3 Feed-forward time delay (FFBPTD)

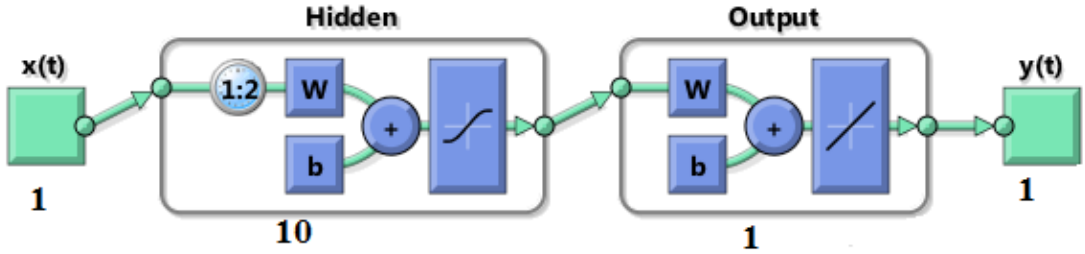


Figure 27: An example of a feed-forward time delay network

Time delay networks are similar to feed-forward networks, except that the input weight has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. Figure 27 shows an example of this network with 1 input, 1 hidden layer with 10 neurons, and 1 output layer with 1 neuron.

5.1.4 Elman back-propagation (ELMAN)

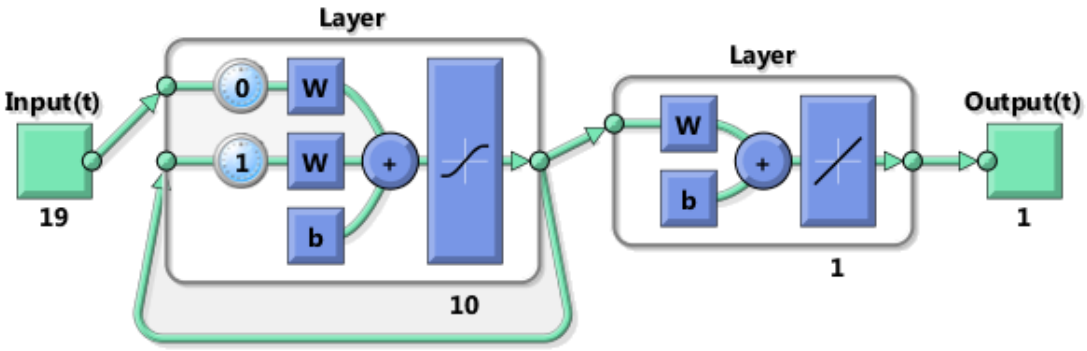


Figure 28: An example of Elman back-propagation network

Elman networks are feed-forward networks with the addition of layer recurrent connections with tap delays.

The Elman network is no longer recommended, except for historical and research purposes, since it uses simplified derivative calculations at the expense of less reliable learning. For more accurate learning, feed-forward time delay networks should be used. An example of an Elman network is shown in Figure 28. This network consists of 19 inputs, 1 hidden layer with 10 neurons, and 1 output layer with 1 neuron.

5.2 Neural network training algorithms

To train the neural network to produce the most accurate results, we need to adapt all weights and biases in the network with the intention of finding a model that works correctly for all inputs. The procedure to achieve this goal is called the training phase.

MATLAB Neural Network Toolbox has a number of training functions to train a network. A list of these functions is provided in Table 11; the formal definitions and implementations of these algorithms are outside the scope of our study.

Table 11: Training Algorithms list

Function name in MATLAB	Training algorithm
trainbfg	BFGS quasi-Newton back-propagation
trainbr	Bayesian regularization
traincgb	Powell –Beale conjugate gradient back-propagation
traincgf	Fletcher-Powell conjugate gradient back-propagation
traincgp	Polak-Ribiere conjugate gradient back-propagation
traingd	Gradient descent back-propagation
traingdm	Gradient descent with momentum back-propagation
traingda	Gradient descent with adaptive lr back-propagation
traingdx	Gradient descent w/momentum & adaptive lr back-propagation
trainlm	Levenberg-Marquardt back-propagation
trainoss	One step secant back-propagation
trainrp	Resilient back-propagation (Rprop)
trainscg	Scaled conjugate gradient back-propagation

5.3 Inputs and outputs of the neural network

In our study we are trying to model the power consumption of mobile phones using NNs, where the inputs to the model are the 17 variables identified in in chapter 3 (Table 6). The output of the model is the power consumption.

All neural network power models developed in this study have 17 inputs with one or more hidden layers with different transfer functions and one output layer. For any dataset, 70% of it is randomly selected as a training dataset, 15% is selected as validation dataset, while the rest (15%) is randomly selected as testing dataset.

5.4 Selecting best NN structures and training algorithms

In an effort to identify the most suitable architecture, we experimented with all types of previously mentioned neural network structures using various training algorithms while computing the RMSE. The results are listed in Table 12.

Table 12: RMSE of different network structures and training functions

Training function	Network structure							
	FFBP		CFFBP		FFBPTD		ELMAN	
	n=10	n=20	n=10	n=20	n=10	n=20	n=10	n=20
trainbfg	0.209797	0.206758	0.216204	0.199763	0.632268	0.632285	0.630952	0.63095
trainbr	0.193328	0.175663	0.184413	0.174294	0.632267	0.632267	0.63095	0.630951
traincgb	0.218694	0.217632	0.21954	0.217823	0.632268	0.632271	0.630949	0.630949
traincgf	0.229647	0.226134	0.227723	0.213663	0.632267	0.632267	0.630949	0.630949
traincgp	0.225715	0.219247	0.227335	0.217196	0.632269	0.632268	0.630956	0.630951
traingd	0.353894	1.192112	0.336454	5.590423	0.632267	0.63227	0.630949	0.63095
traingdm	0.966383	0.845086	3.510545	5.550865	0.883342	0.654927	0.632103	0.630972
traingda	0.332997	0.39005	0.454985	0.482595	0.632267	0.632568	0.630949	0.630961
traingdx	0.286482	0.284275	0.306612	0.317421	0.632344	0.632833	0.631744	0.63095
trainlm	0.188031	0.176105	0.190761	0.180157	0.632267	0.632283	0.630973	0.630949
trainoss	0.233728	0.236423	0.223991	0.234473	0.632361	0.632269	0.630952	0.630949
trainrp	0.231186	0.239729	0.242783	0.247439	0.632274	0.806998	0.630973	0.630949
trainscg	0.22851	0.229321	0.232129	0.218688	0.632267	0.632268	0.63095	0.63095

In the above table, the first column specifies the training algorithm used to train the neural network, columns 2 and 3 are for the FFBP network structure with 10 and 20 neurons in the hidden layer respectively, and columns 4 and 5 are for the CFFBP network structures with 10 and 20 neurons in the hidden layers respectively. The same order is applied for the FFBPTD and Elman network structures.

It is clear that *trainbr* and *trainlm* training algorithms have the best performance over all other algorithms. The FFBPTD and Elman network architectures did not perform well with any training algorithm (high and not-improving RMSE); hence, we will not include them in any further analysis.

The RMSE improves by increasing the number of hidden nodes, but what is the optimal number of hidden nodes to model our system? We can use either *trainbr* or *trainlm* with FFBP or CFFBP network structures. We need to answer the following question: “which combination is the best?”

Table 13 contains the training time required to train each network. The organization of this table is the same as Table 12 except that the entries in this table are the training time in seconds, not the RMSE values. Table 12 shows that *trainbr* has better performance over *trainlm*; however, results in Table 13 show that *trainbr* requires two times the training time of *trainlm* with an RMSE maximum improvement of only 3.5%. Finally, *traingdm* requires the least time to train the network, but also has the worst performance.

Table 13: Training time required to train networks of different architectures using different training functions

Training function	Network structure							
	FFBP		CFFBP		FFBPTD		ELMAN	
	n=10	n=20	n=10	n=20	n=10	n=20	n=10	n=20
trainbfg	560.1874	577.9164	488.3009	885.5299	21.76374	28.44837	9.300738	16.20378
trainbr	529.1933	661.35	709.0895	944.2664	117.9369	633.9807	73.32792	106.6371
traincgb	685.6228	715.9561	686.6167	762.2569	15.20419	19.50082	23.12666	22.01755
traincgf	728.115	640.3829	1034.791	1118.346	24.88824	19.42838	16.77378	26.91041
traincgp	683.7995	674.8491	507.0572	1003.557	13.84029	24.83923	15.82576	18.97255
traingd	252.4434	9.03335	397.8197	9.829319	15.08716	29.97665	23.50376	7.26809
traingdm	10.21414	19.31096	8.943479	23.59974	11.39161	18.7903	12.00109	11.2225
traingda	73.37219	124.8304	168.5904	186.7978	16.62864	38.57245	17.5082	9.93892
traingdx	149.7136	160.7029	137.9587	162.3984	24.48057	26.59189	15.66115	18.95144
trainlm	268.3789	380.7035	576.4763	929.8259	8.999159	22.90891	6.599662	9.924979
trainoss	375.6005	345.4326	854.437	658.3433	20.60792	17.29615	13.08494	10.94395
trainrp	253.1726	268.0829	407.9637	418.6255	36.97882	10.3354	28.49819	23.9333
trainscg	393.1616	483.8855	295.2382	835.7505	30.25095	28.35549	8.889883	10.06805

We next study all possible combinations of network architectures and training algorithms with different numbers of neurons in the hidden layer.

5.5 Selecting transfer function and number of neurons

In the previous section we showed that FFBP and CSFFBP networks trained with either *trainlm* or *trainbr* produced the best performance. In this section we will study all the combinations of these network structures and training algorithms along with different transfer functions. The main goal is to select the neural network that yields the most accurate power model.

We keep the transfer function of the output layer (*purelin*) while changing the transfer function of the hidden layer to either *logsig* or *tansig* and changing the number of neurons in the hidden layer for both FFBP and CFFBP networks. We plot the RMSE of the final model and its training time in Figure 29 and Figure 30, respectively. In these figures, LS denotes the *logsig* transfer function, TS denotes the *tansig* transfer function, BR denotes *trainbr* training algorithms, and LM denotes the *trainlm* training algorithm.

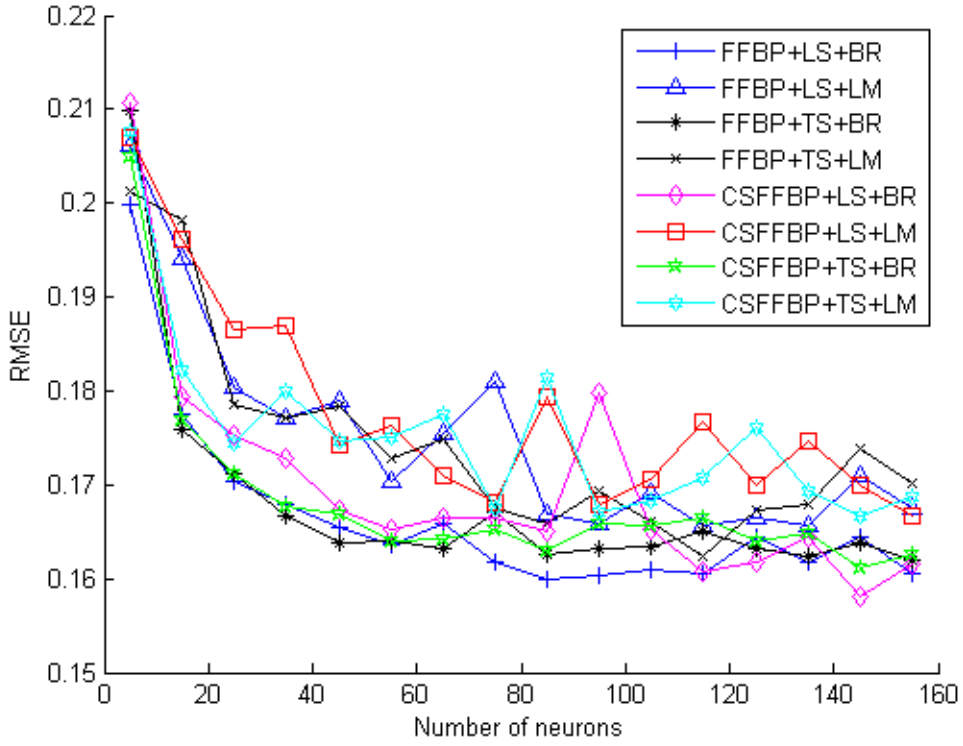


Figure 29: RMSE of the neural network power model for various network configurations

It is clear that *trainlm* has some random nature; the RMSE did not improve with increasing the number of neurons. Furthermore, *trainbr* is more stable than *trainlm*; the RMSE improves with increasing number of neurons. We also note that FFBP has a slightly better performance than CSFFBP. We therefore select as the best NN the FFBP NN trained using *trainbr* with a hidden layer consisting of 85 neurons each using *logsig* as the transfer function.

Figure 30 shows the time required to train different network configurations. We can see that training time when using *trainlm* is lower than when using *trainbr*, but it exhibits some randomness. For our selected configuration, the time required to train the network is about 3000 seconds (50 minutes); this is an affordable time when we want to build the model once using a typical computer. If we want to build the model for each user using his/her own mobiles as computation platforms, it is better to use simpler and lower-training-time configurations even at the cost of expected accuracy.

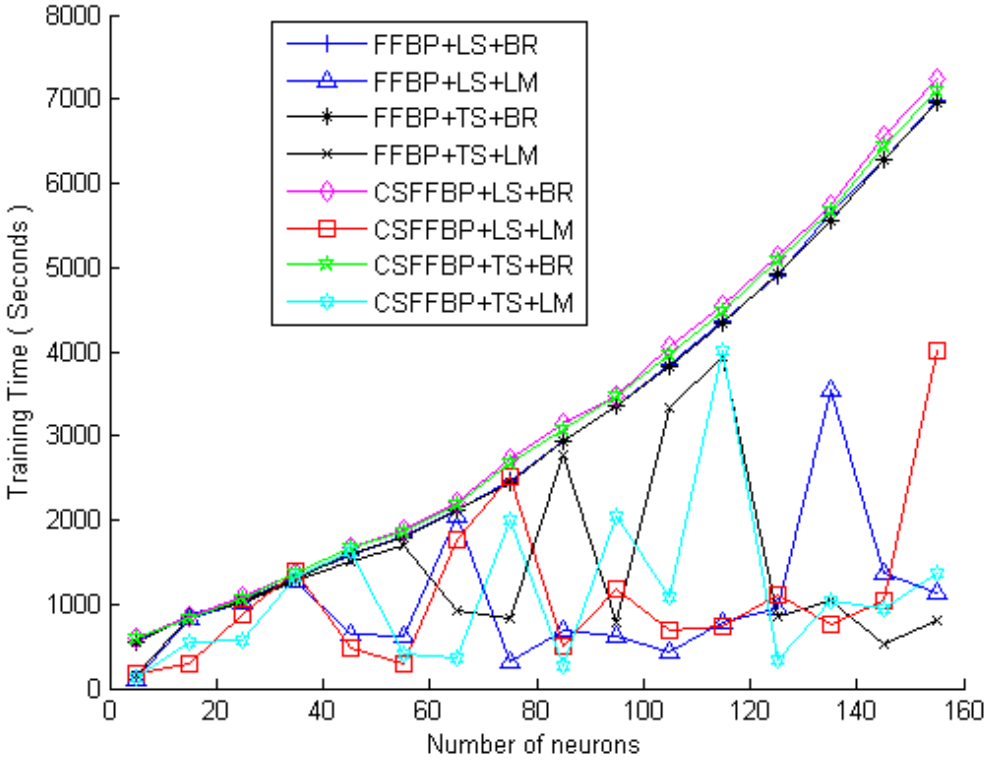


Figure 30: Training time of neural networks power models for various configurations

Figure 31 shows the selected structure for our final neural network power model. The training algorithm for this network is *trainbr*. The hidden layer consists of 85 neurons each using the *logsig* transfer function, while the output layer contains only one neuron that uses the *purelin* transfer function.

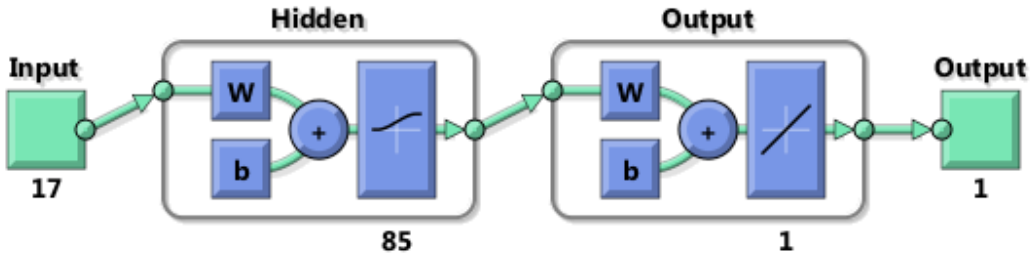


Figure 31: Selected network structure

Figure 32 shows the performance of our neural network power model. It is clear that the accuracy achieved is very good, with an R value of 0.96747 which is very close to the ideal value of unity.

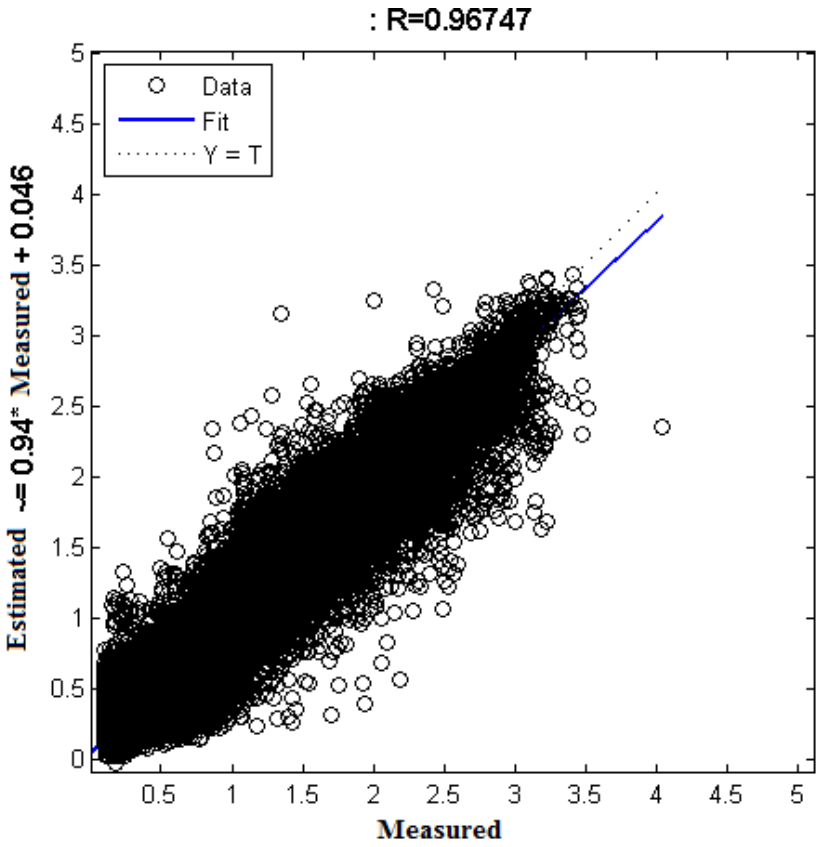


Figure 32: Performance plot of single hidden layer neural network

5.6 Effect of the number of hidden layers

In the previous section we used a neural network with one hidden layer to model the smartphone power consumption. In this section we will examine the effect of adding another hidden layer to the neural network. Figure 33 shows the proposed network structure, where *m* denotes the number of neurons in the first hidden layer, *n* is the number of neurons in the second hidden layer, T1 is the transfer function for first hidden layer, and T2 is the transfer function for the second layer. In Table 14, we list the RMSE and training time of the networks with different values of *m*, *n*, T1, T2, and training algorithms.

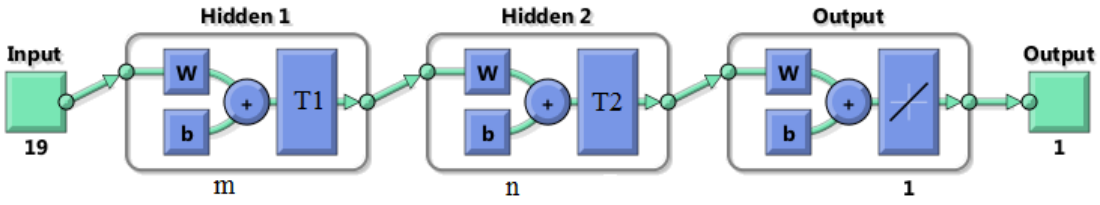


Figure 33: NN Architecture with 2 hidden layers

Table 14: Performance and training time for 2 hidden layer neural networks

T1	T2	m	TRAINBR Algorithm							
			RMSE				Training Time (Sec)			
			n=5	n=25	n=45	n=65	n=5	n=25	n=45	n=65
tansig	**tansig**	5	0.1904	0.1778	0.1807	0.1789	610	816	986	1187
		25	0.1639	0.1573	0.1545	0.1567	1189	1877	3019	4334
		45	0.1589	0.1528	0.1516	0.1577	1744	3782	6582	11032
		65	0.1594	0.1521	0.1532	0.1546	2631	6409	13003	23524
tansig	**logsig**	5	0.1991	0.1803	0.1816	0.1741	597	808	981	1213
		25	0.1610	0.1545	0.1555	0.1537	1209	1933	2988	4340
		45	0.1587	0.1501	0.1528	0.1499	1753	3776	6628	11194
		65	0.1577	0.1518	0.1506	0.1516	2641	6498	13059	23671
logsig	**tansig**	5	0.1917	0.1815	0.1782	0.1778	603	810	993	1199
		25	0.1635	0.1604	0.1561	0.1652	1169	1913	3041	4370
		45	0.1621	0.1580	0.1607	0.1578	1746	3822	6685	11057
		65	0.1556	0.1584	0.1610	0.1683	2633	6369	13025	16551
logsig	**logsig**	5	0.1887	0.1816	0.1758	0.1719	626	839	989	1216
		25	0.1624	0.1580	0.1583	0.1610	1212	1924	3068	4355
		45	0.1590	0.1576	0.1588	0.1571	1782	3840	6668	11135
		65	0.1581	0.1655	0.1536	0.1663	2656	6425	13185	24358

The first two columns contain all possible enumerations of the transfer functions used in the first hidden layer (T1) and second hidden layer (T2); the third column contains different numbers of neurons in the first hidden layer (*m*) for each enumeration of the transfer function. The next 4 columns contain a different number of neurons in the second hidden layer (*n*). The sub-matrix for each transfer function enumeration contains all RMSE values for a different number of neurons in the first and second hidden layers specified by *m* row and *n* column values. The same order is applied for the next 4 columns, but the training time required for each configuration is in seconds.

We train the network in Figure 33 with different training parameters. The RMSE of the final network and the training time is shown in Table 14. It is clear that we achieve the best results using the network configuration T1=*tansig* and T2=*logsig*. We choose the number of neurons in the hidden layers to be 45 and 25 in the first and second hidden layers, respectively. The time required to train this networks is 3776 seconds (63 minutes).

Figure 34 shows the performance comparison between the 2-hidden-layer neural network and 1-hidden-layer neural network. We note that the performance of the 2-hidden-layer networks is slightly better than that of the 1-hidden-layer network.

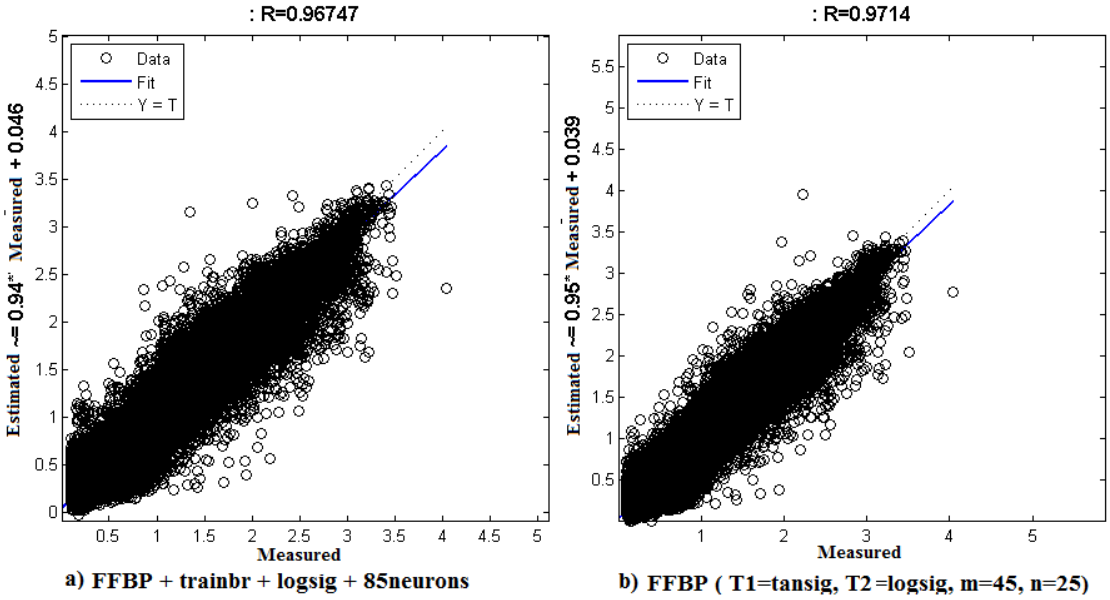


Figure 34: Performance comparison between 1-hidden-layer and 2-hidden-layer neural networks

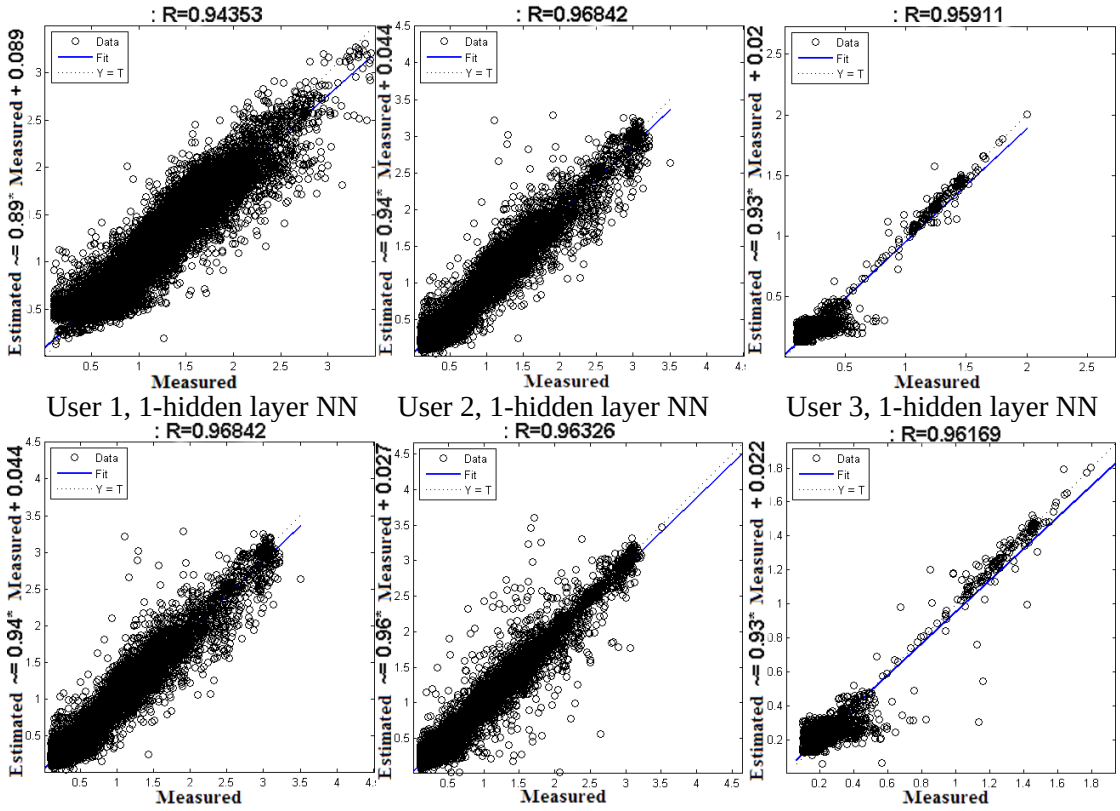
5.7 User level power modeling

Here we build two NN candidate models. The first NN candidate consists of 17 input variables, 1 hidden layer with 85 neuron using *logsig as* the transfer function, and 1 output layer (power) using the *purelin* transfer function. The second NN candidate model consists of 17 input variables, 2 hidden layers with 45 neurons for the first layer (*tansig* transfer function) and 25 neurons for the second layer (*logsig* transfer function), and 1 output layer using the *purelin* transfer function.

We developed user-level power models for each user using his/her power samples only. The samples are divided randomly into a training dataset (70%), validation dataset (15%), and testing dataset (15%). We build the power models using previously identified 1-hidden-layer NN architecture and 2-hidden-layer architecture. Figure 35 contains the plots of the estimated power (Estimated) vs. measured power (Measured) for the first 3 users using a 1-hidden-layer NN and a 2-hidden-layer NN. The performance of the models for all users will be further discussed in Chapter 6.

In the ideal case, estimated power would be equal to measured power, which means all points should be located on the diagonal line Y=X. The correlation

coefficient (R) indicates how much measured power and estimated power are close to each other. The closer the value of R to unity, the more the accurate the model.



User 1, 1-hidden layer NN User 2, 1-hidden layer NN User 3, 1-hidden layer NN

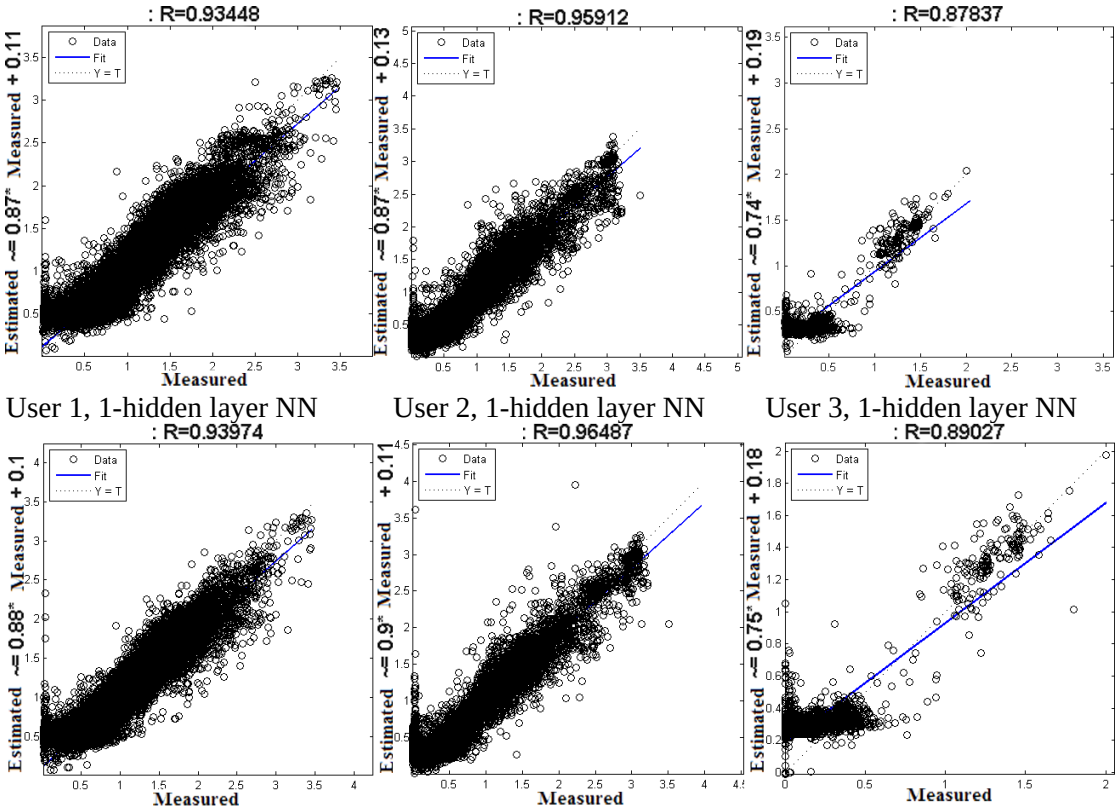
User 1, 2-hidden layers NN User 2, 2-hidden layers NN User 3, 2-hidden layers NN

Figure 35: User level performance plots for first 3 users

Figure 35 shows that the 2-hidden-layer NN's performance is slightly better than the 1-hidden-layer NN for users 1 and 3 (the R value we got when using the 2-hidden-layer models is a little higher than when using the 1-hidden-layer model). For user 2, the 1-hidden-layer NN model performance is better than the 2-hidden-layer NN model.

5.8 Device level power modeling

We previously developed power models using 1-hidden-layer and 2-hidden-layer networks (see Figure 31 and Figure 33 with *m*=45, *n*=25, T1=*tansig*, T2=*logsig*). These two models are the device-level power model. Figure 36 shows the performance plot when using these two models to model the power consumption for the first 3 users. Performance plots for the other users will be discussed in Chapter 6.



User 1, 1-hidden layer NN User 2, 1-hidden layer NN User 3, 1-hidden layer NN

User 1, 2-hidden layers NN User 2, 2-hidden layers NN User 3, 2-hidden layers NN

Figure 36: Device level performance plots for first 3 users

Figure 36 shows that the 2-hidden-layer performance is better than the 1-hidden-layer for user 1 and user 3, while user 2's 1-hidden-layer NN model has slightly better performance than the 2-hidden-layer NN model. Referring to Figure 35, we can conclude that user-level power models have better performance than device-level power models.

Chapter 6

Results Comparison and Discussion

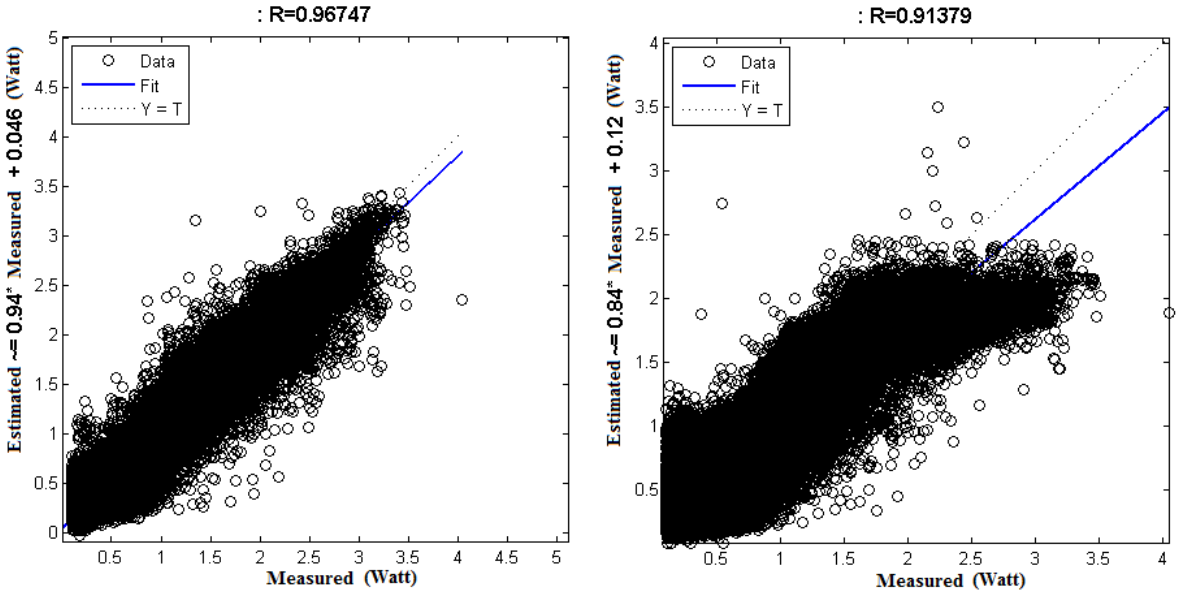
In this chapter, we discuss and compare the results obtained using the following approaches:

- Regression-based modeling at the user level
- Regression-based modeling at the device level
- Neural network-based modeling at the user level
- Neural network-based modeling at the device level

The comparison is in terms of the correlation coefficient (R), and the Root Mean Squared Error (RMSE).

6.1 Comparison of Results at the Device Level

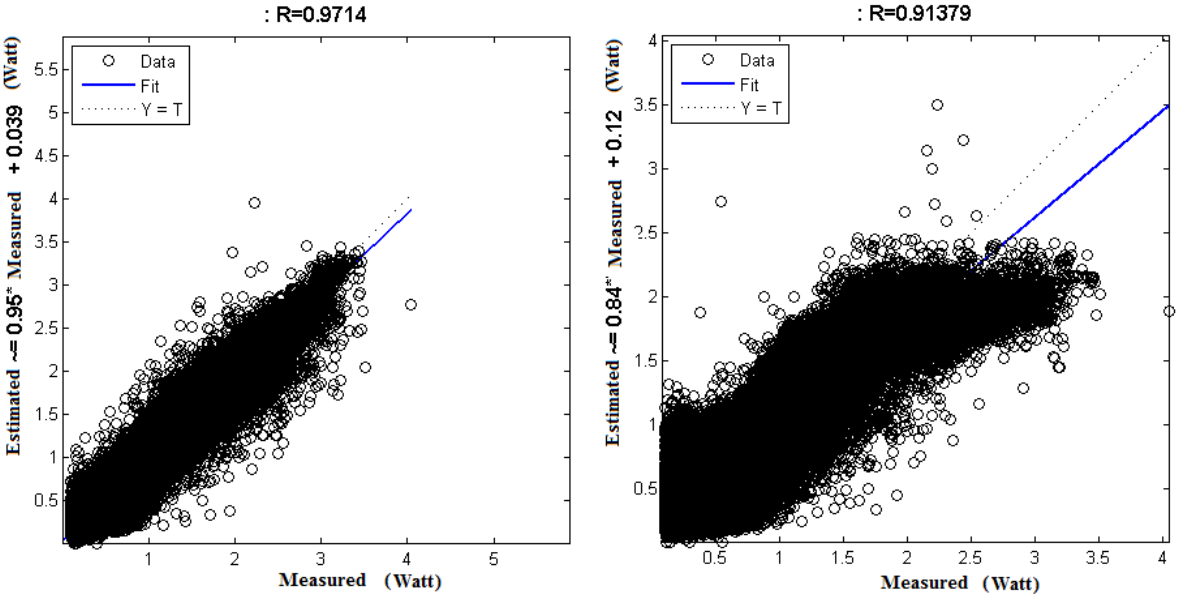
In Figure 37 we show the performance (using power samples from all users) of a 1-hidden-layer neural network (NN) power model (Figure 37.a) compared to an ordinary least squares (OLS) regression power model that we built in section 4.2.2 (Figure 37.b). It is clear that the accuracy we achieved when using the 1-hidden-layer neural network power model (R=0.96747) is higher than when using the OLS regression model (R=0.91379). The neural network power model does not exhibit any tailing effects, that is, the power model is working fine with both low power samples as well as high power samples, while the OLS regression model tends to fail for high power samples.



a) FFBP + trainbr + logsig + 85neurons **b) OLS regression model**

Figure 37: Performance comparison between single hidden layer neural network and OLS

Figure 38 shows the performance comparison between the 2-hidden-layer neural networks and OLS; here as well the NN model has better performance than the OLS model and has no tailing effect.



a) FFBP (T1=tansig, T2=logsig, m=45, n=25) **b) OLS regression model**

Figure 38: Performance comparison between 2 hidden layers neural network and OLS.

In Figure 39 we plot the CDF of the RMSE error for the regression model, 1-hidden-layer NN model, and 2-hidden-layer NN model. It is clear that NN models are

more accurate than regression models (more close the y-axis). We note that the 2-hidden-layer NN model is slightly more accurate than the 1-hidden-layer NN model.

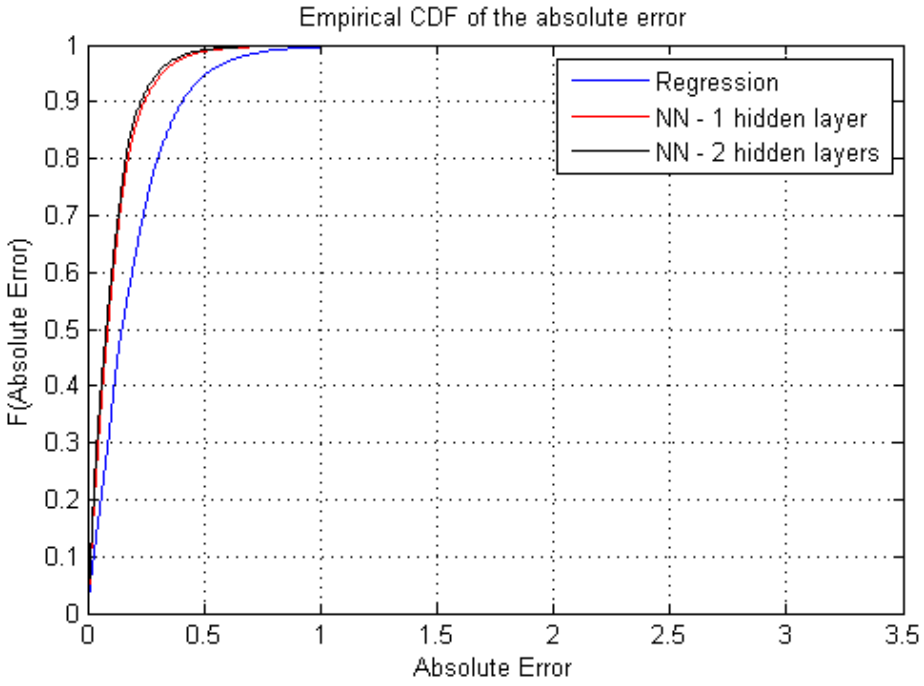
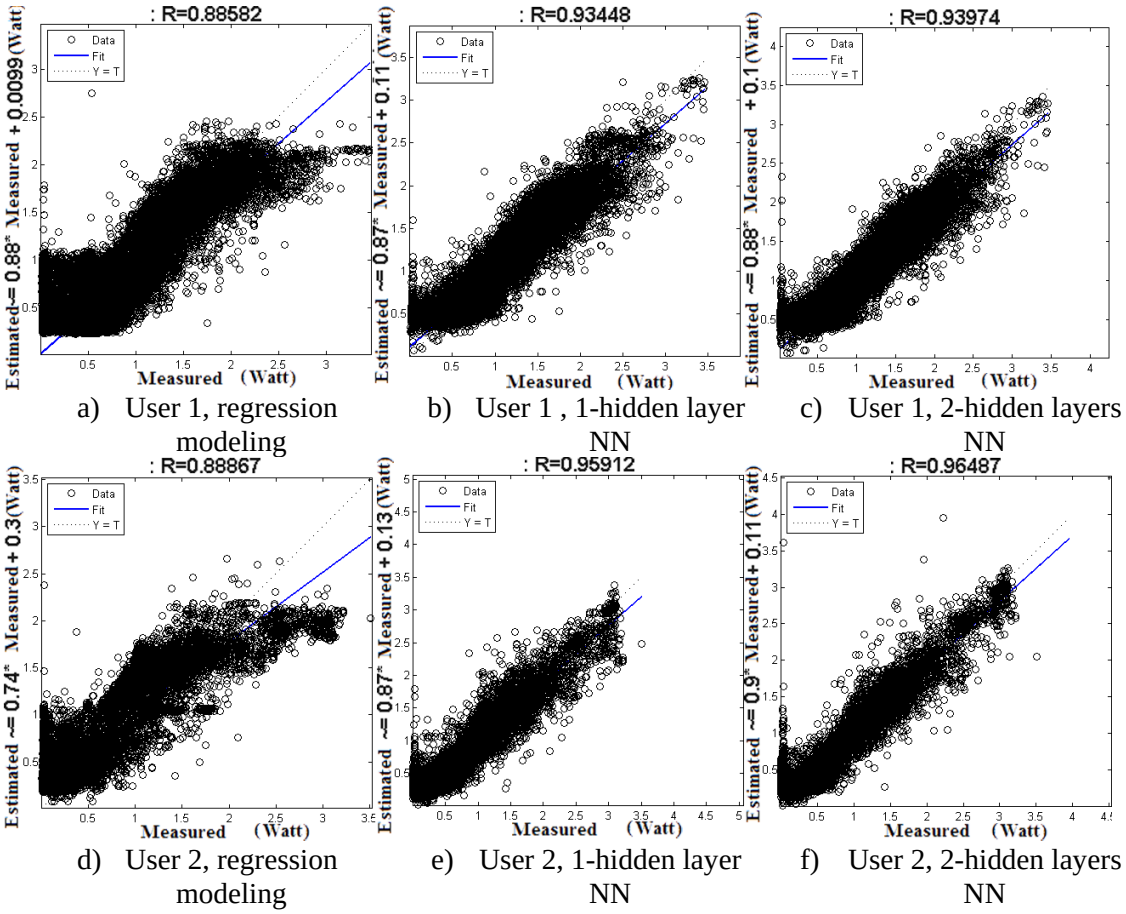


Figure 39: Empirical CDF comparison between power models

Figure 40 shows the device-level performance plots of the power models for 2 users when using OLS (Figure 40.a and 4.0.d), 1-hidden-layer NN (Figure 40.b and Figure 40.e) and 2-hidden-layer NN (Figure 40.c and Figure 40.f). It is clear that NN models have better performance over the regression models. The performance difference between the 1-hidden-layer and 2-hidden-layer NN is negligible. Appendix A contains performance plots for all users at the device level.



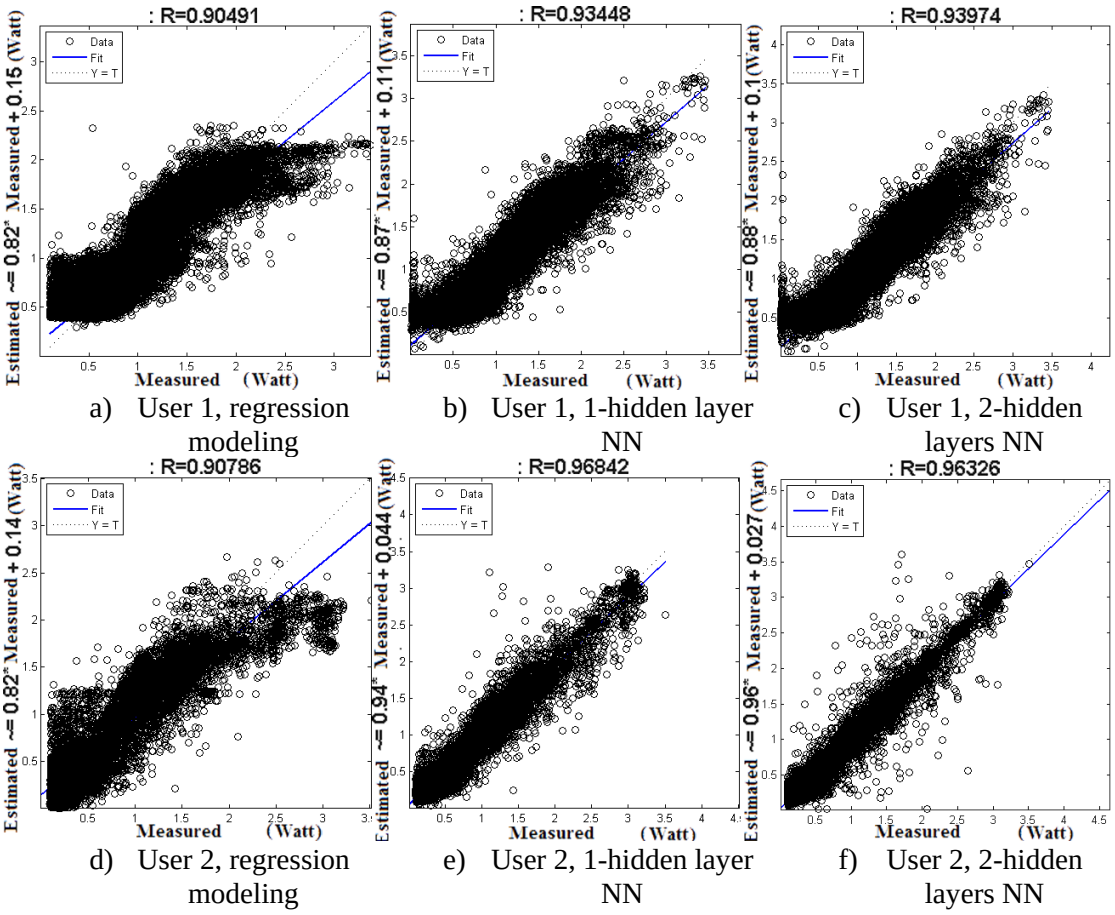
a) User 1, regression modeling
b) User 1 , 1-hidden layer NN
c) User 1, 2-hidden layers NN
d) User 2, regression modeling
e) User 2, 1-hidden layer NN
f) User 2, 2-hidden layers NN

Figure 40: Device level power modeling performance comparison for 2 users

6.2 Comparison of Results at the User Level

User-level power models are generated for each user using regression analysis in Chapter 4 (section 4.2.1) and neural networks in Chapter 5 (section 5.7). These user-level models are varied among users depending on their smartphone usage behavior.

Figure 41 is a comparative plot showing the performance of the different user-level modeling approaches. Yet again, we observe that the NN performed better, resulting in higher values of R.



a) User 1, regression modeling b) User 1, 1-hidden layer NN c) User 1, 2-hidden layers NN

d) User 2, regression modeling e) User 2, 1-hidden layer NN f) User 2, 2-hidden layers NN

Figure 41: User-level power modeling performance comparison for 2 users

Performance plots for the rest of the sample group are included in Appendix B.

6.3 Comparison of Device and User Level Models

In this section we provide an overall comparison of results obtained using the various approaches. Again the criteria will be kept as the correlation coefficient (R) and the Root Mean Square Error (RMSE). As the graphs (Figure 42 and Figure 43) indicate at the device level and for all users modeled (1 to 8), neural network-based models consistently outperformed regression-based models. Users 9 and 10 used a different smartphone model (Sony Xperia S), and hence are not included in this comparison. In the case of users 4 and 6, the NN models achieved a very high value of R and relatively low RMSE values. In the case of users 2, 5 and 8, again R was still greater than or equal to 0.92 for NN models and greater than or equal to 0.77 for regression models. User 7 is a case where the regression model failed with respect to NN models. Even though the R values for the regression model and NN model are very close, the RMSE value for the device-level regression-based model is twice that

of the NN-based model. The graphs also illustrate an exceptionally low R and a high RMSE for user 3. This user is a special case, as his profile (Appendix A) indicated an erratic usage of his smartphone and his/her usage profile is an example of an "outlier" or anomaly.

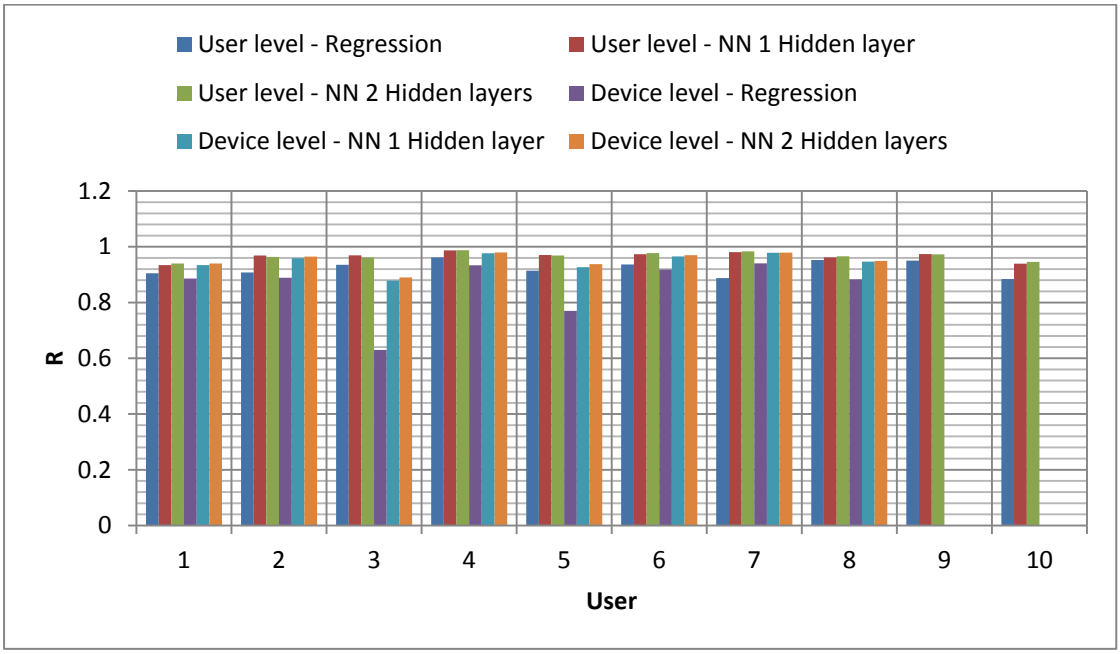


Figure 42: Comparison between regression, 1-hidden-layer NN and 2-hidden-layer NNin terms of R

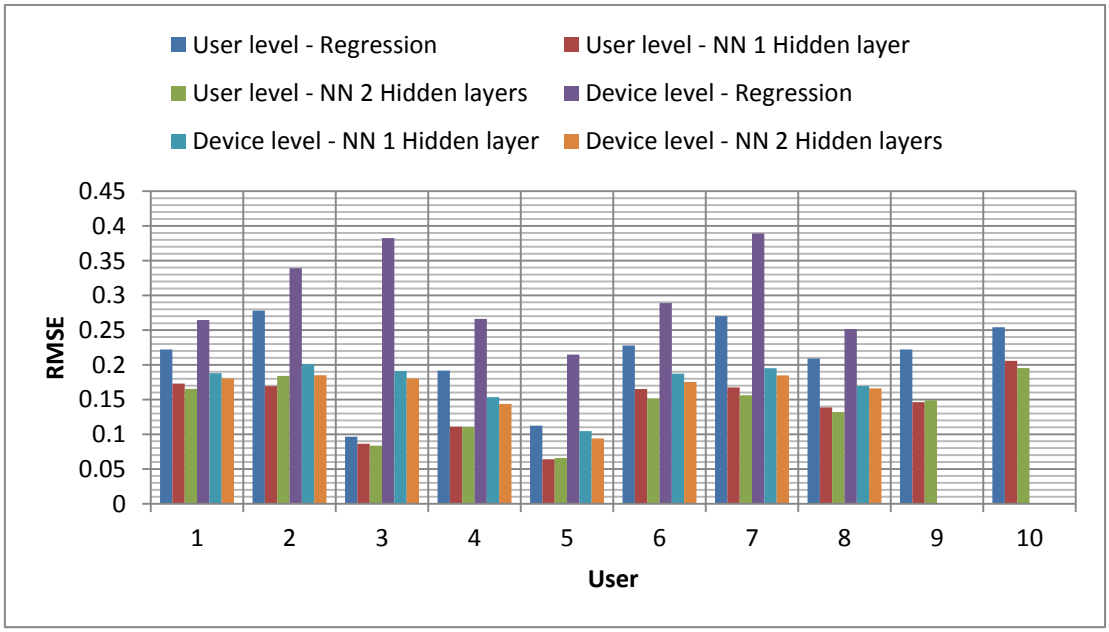


Figure 43: Comparison between regression, 1-hidden-layer NN and 2-hidden-layer NN in terms of RMSE

At the user level, as well, the outperformance of the NN compared to regression is evident. However, still at the user level, regression-based results (R and RMSE) were acceptable when compared to the performance of the regression-based approach at the device level.

It is clear that the performances of user-level models are more accurate (or have the same accuracy) than device-level models for all approaches (see Figure 42 and Figure 43). However, device-level is still accurate enough in providing a general power model for a smartphone.

Chapter 7
Conclusion and future work

As smartphones become more and more advanced, their computing capabilities have almost reached those of PCs. However, battery technology is slowly advancing and can't satisfy the power requirements of modern smartphones. This problem is further aggravated by the fact that the limited size and weight of the smartphones does not allow for the utilization of larger batteries.

The only hope for elongating the battery life in smartphones is to improve the energy efficiency of the smartphones' components, which is highly dependent on the smartphone's application computing requirements. The end user of the smartphone determines which applications to run, thus which components will be used, and hence the overall power consumption of the smartphone. The actual workload of the smartphone is the end user; understanding the end user's usage behavior and modeling the corresponding power consumption would be the corner stone for future power-efficient designs of both components and applications.

In this work we developed an application to log smartphone usage patterns and assist in computing the power consumption. We used correlation analysis to select a subset of predictors that could be utilized to adequately model power using regression-based techniques. Furthermore, we also studied the feasibility of using neural network techniques in generating reliable power models.

Regression-based models are very fast to construct; however, they do not perform as well when compared to NN-based power models. NN-based power models require more construction time (training time) than regression-based power models. From the results, we can conclude that the 2-hidden-layer NN power models are the most accurate models, but they require more training time. It also observed that 1-hidden-layer NN models have accuracy that is comparable to the 2-hidden-layer NN models with less training time.

User-level power models are built based on the usage patterns for each user while device-level power models are built based on the usage patterns of all users of a smartphone model. User-level power models perform better than device-level models. Since only the user data is used to construct and test the model, this data is more representative of the user's behavior than general data. However, device-level power models are still useful in providing an insight into the possible consumption characteristics of the device.

Device-level power models can be used with smartphone emulators [**36**] to estimate the power consumption of either single applications or sets of application with different usage scenarios. This can help software developers to increase the energy efficiency of their applications with different usage scenarios and patterns. Regression-based device level models can provide a detailed breakdown of the power consumption of an application.

A core requirement of effective and efficient management of energy is a good understanding of where and how the energy is used [**10**], that is, which components consume the most power and how. Some energy management techniques rely on instruction-level power models to predict the power consumption of a device [**37**], while others depend on performance counters to model power and energy management [**38**]. Smartphone applications are interactive and hence the usage behavior is the actual workload of the smartphone, not only the energy cost of the instructions. Any future smartphone energy management technique should take into account usage behavior-based power modeling to estimate and predict power consumption and hence power-aware scheduling of threads and processes.

For future work and to improve the outcome of this thesis we make the following recommendations:

- Capture information about the power state of the system (sleep, active etc.), as this can allow us to build power models based on the various states of the mobile.
- Capture information about memory usage (using Android wake locks).

- Study the user profile patterns for a larger community of users. This will require developing a generic logger that can run on almost all Android-based mobiles.
- Last but not least, researchers can attempt to develop application-level models to further understand the power consumption problem.

Bibliography

[1] Google. (2013, April) Our Mobile Planet. [Online]. http://www.thinkwithgoogle.com/mobileplanet/en/

[2] Hossein Falaki, Ratul Mahajan, and Deborah Estrin, "SystemSens: a tool for monitoring usage in smartphone research deployments," in *Proceedings of the sixth international workshop on MobiArch*, New York, NY, USA, 2011, pp. 25-30.

[3] S. Keshav, Y. Chawathe, M. Chen, Y. a Zhang, and A. Wolman, "Cell phones as a research platform," *MobiSys Panel*, 2007.

[4] Earl Oliver, "A survey of platforms for mobile networks research," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 12, pp. 56-63, 2008.

[5] Hossein Falaki et al., "Diversity in smartphone usage," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, pp. 179-194.

[6] Qiang Xu et al., "Identifying diverse usage behaviors of smartphone apps," in *Proceedings of the ACM SIGCOMM conference on Internet measurement conference*, 2011, pp. 329-344.

[7] Andrew Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260-269, 1967.

[8] Claude Berrou, Alain Glavieux, and Punya Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *IEEE International Conference on Communications, ICC 93. Geneva. Technical Program, Conference Record*, Geneva, 1993, pp. 1064-1070.

[9] Andrew Rice and Simon Hay, "Measuring mobile phone energy consumption for 802.11 wireless networking," *Pervasive and Mobile Computing*, vol. 6, pp. 593-606, 2010.

[10] Aaron Carroll and Gernot Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the USENIX conference on USENIX annual technical conference*, 2010, pp. 21-21.

[11] GP Perrucci, FHP Fitzek, and J Widmer, "Survey on energy consumption entities on the smartphone platform," in *IEEE 73rd Vehicular Technology Conference (VTC Spring)*, 2011, pp. 1-6.

[12] Maruti Gupta, Ali Taha Koc, and Rath Vannithamby, "Analyzing mobile applications and power consumption on smartphone over LTE network," in *International Conference*

on Energy Aware Computing (ICEAC), 2011, pp. 1-4.

[13] Abdulhakim Abogharaf, Rajesh Palit, Kshirasagar Naik, and Ajit Singh, "A methodology for energy performance testing of smartphone applications," in *7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 110-116.

[14] Rajesh Palit, Ajit Singh, and Kshirasagar Naik, "Modeling the energy cost of applications on portable wireless devices," in *Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems*, 2008, pp. 346-353.

[15] Alex Shye, Benjamin Scholbrock, and Gokhan Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 168-178.

[16] Earl Oliver and Srinivasan Keshav, "Data driven smartphone energy level prediction," University of Waterloo Technical Report, 2010.

[17] Lide Zhang et al., "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*, 2010, pp. 105-114.

[18] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 153-168.

[19] Mian Dong and Lin Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 335-348.

[20] Jaymin Lee, Hyunwoo Joe, and Hyungshin Kim, "Smart phone power model generation using use pattern analysis," in *IEEE International Conference on Consumer Electronics (ICCE)*, 2012, pp. 412-413.

[21] (2013, March) Logging with Log4J in Android. [Online]. https://code.google.com/p/android-logging-log4j/

[22] TI. (2013, March) BQ27520-g3 Fuel Gauge With Integrated LDO. [Online]. http://www.ti.com/product/bq27520-g3

[23] Douglas C Montgomery and George C Runger, *Applied statistics and probability for engineers*.: Wiley, 2010.

[24] Kun-Lin Hsieh and Yen-Sheng Lu, "Model construction and parameter effect for TFT-LCD process based on yield analysis by using ANNs and stepwise regression," *Expert Systems with Applications*, vol. 34, pp. 717-724, 2008.

[25] Yu Xiao et al., "A system-level model for runtime power estimation on mobile devices," in *Green Computing and Communications (GreenCom), IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 27-34.

[26] Sumit Ahuja, Deepak A Mathaikutty, Avinash Lakshminarayana, and Sandeep Shukla, "Accurate power estimation of hardware co-processors using system level simulation," in *System-on-Chip (SOC) Conference. IEEE International*, 2009, pp. 399-402.

[27] Bruno Leonardi and Venkataramana Ajjarapu, "Development of Multilinear Regression Models for Online Voltage Stability Margin Estimation," *IEEE Transactions on Power Systems*, vol. 26, pp. 374-383, 2011.

[28] Rahul Murmuria, Jeffrey Medsger, Angelos Stavrou, and Jeffrey M Voas, "Mobile Application and Device Power Usage Measurements," in *IEEE Sixth International Conference on Software Security and Reliability (SERE)*, 2012, pp. 147-156.

[29] Patricia and Pedrycz, Witold Melin, *Bio-inspired hybrid intelligent systems for image analysis and pattern recognition*.: Springer, 2009.

[30] Warren S McCulloch and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

[31] Martin T Hagan, Howard B Demuth, and Mark H Beale, *Neural network design*.: Pws Pub. Boston London, 1996.

[32] A Suissa, O Romain, J Denoulet, K Hachicha, and P Garda, "Empirical method based on neural networks for analog power modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 839-844, 2010.

[33] Dipankar Dhabak and Soumya Pandit, "An Artificial Neural Network-Based Approach for Performance Modeling of Nano-Scale CMOS Inverter," in *Institute of Engineering and Management Conference*, 2011, pp. 165-170.

[34] (2013, March) Neural network toolbox documentation. [Online]. http://www.mathworks.com/help/nnet/index.html

[35] Matlab. (2013, March) Feed forward Neural Networks. [Online]. http://www.mathworks.com/help/nnet/ref/feedforwardnet.html

[36] Google. (2013, May) Android Emulator. [Online]. http://developer.android.com/tools/help/emulator.html

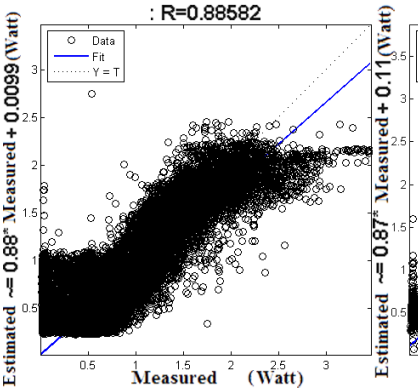
[37] Mike Tien-Chien Lee, Vivek and Malik, Sharad Tiwari, and Masahiro Fujita, "Power analysis and low-power scheduling techniques for embedded DSP software," in *Proceedings of the 8th international symposium on System synthesis*, 1995, pp. 110-115.

[38] Karan Singh, Major Bhadauria, and Sally A McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture*

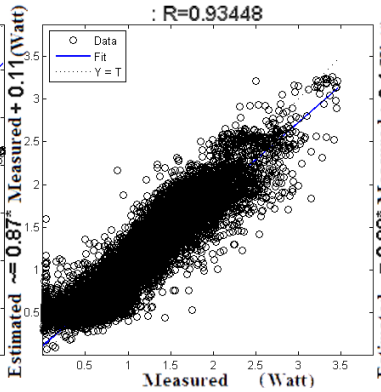
News, vol. 37, pp. 46-55, 2009.

Appendices

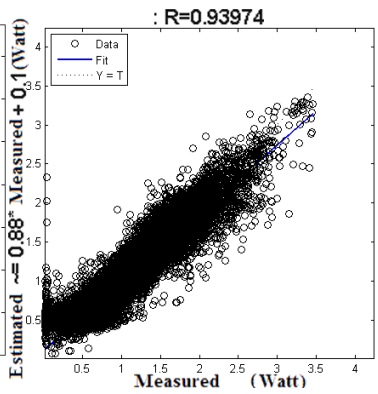
Appendix-A: Device level power modeling performance comparison



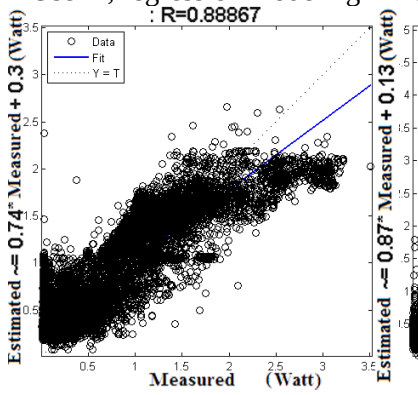
User 1, regression modeling



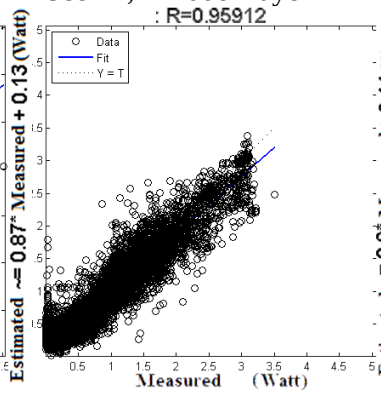
User 1 , 1 hidden layer NN



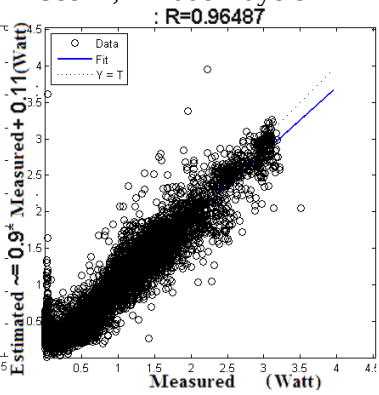
User 1, 2 hidden layers NN



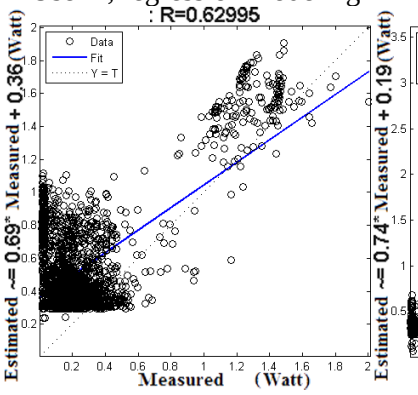
User 2, regression modeling



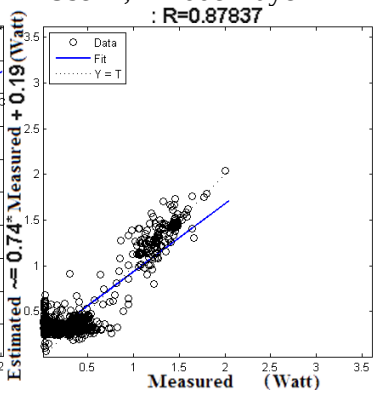
User 2, 1 hidden layer NN



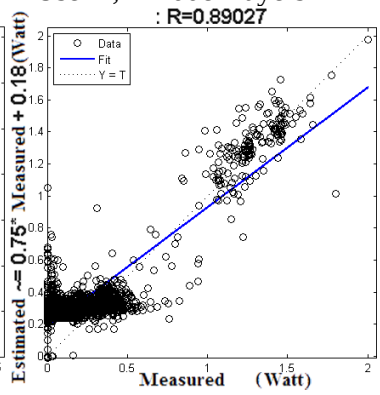
User 2, 2 hidden layers NN



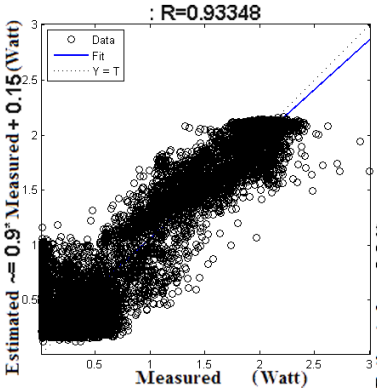
User 3, regression modeling



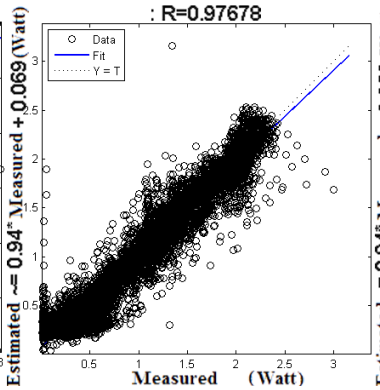
User 3, 1 hidden layer NN



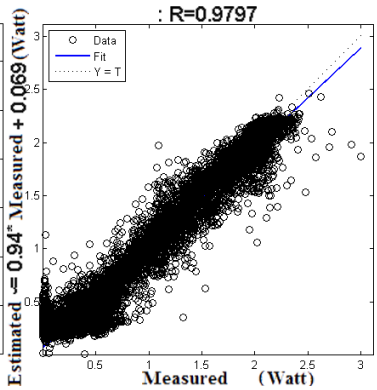
User 3, 2 hidden layers NN



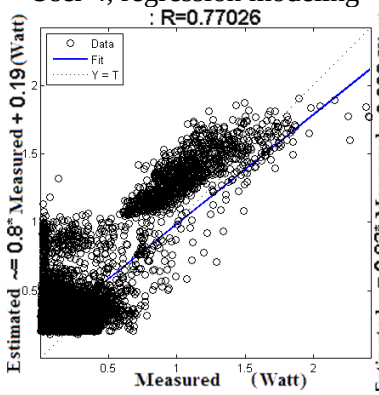
User 4, regression modeling



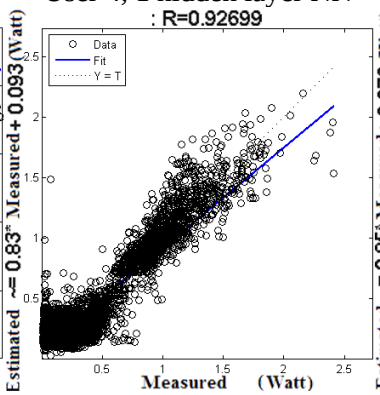
User 4, 1 hidden layer NN



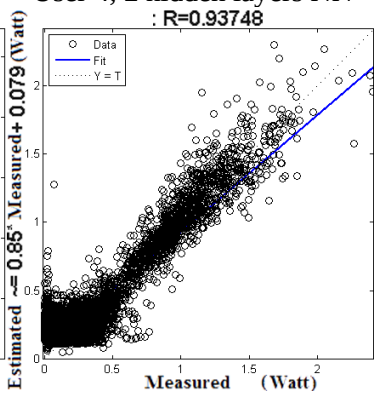
User 4, 2 hidden layers NN



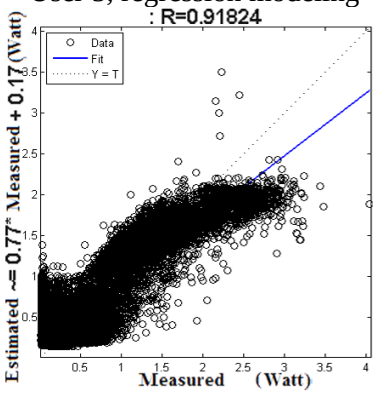
User 5, regression modeling



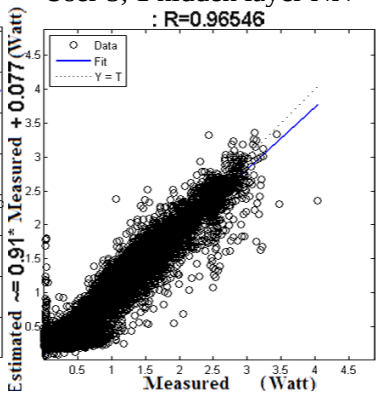
User 5, 1 hidden layer NN



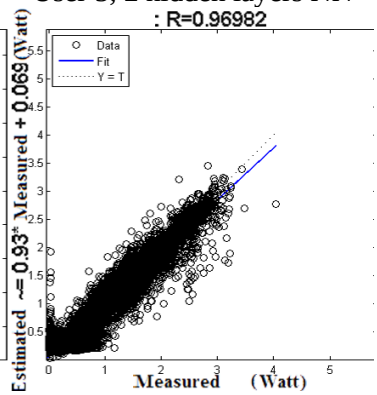
User 5, 2 hidden layers NN



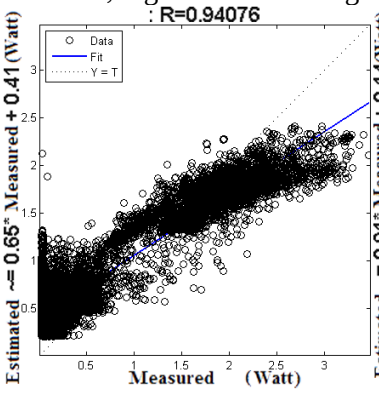
User 6, regression modeling



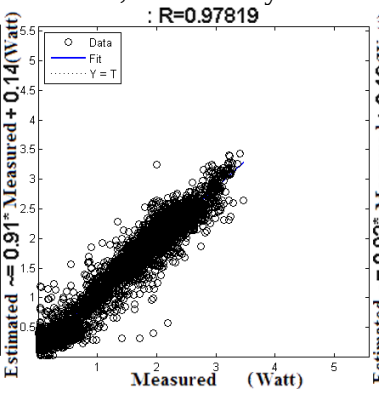
User 6, 1 hidden layer NN



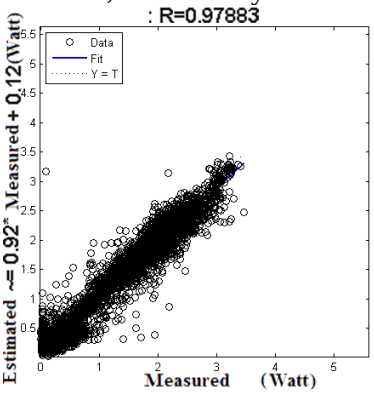
User 6, 2 hidden layers NN



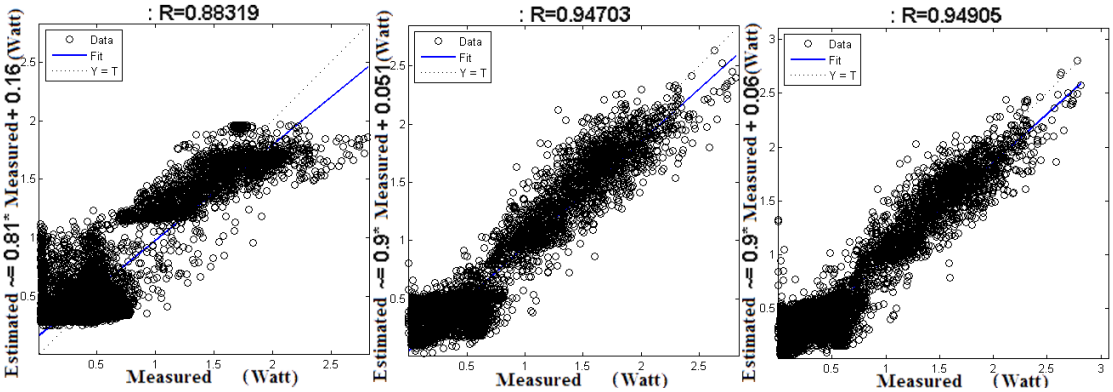
User 7, regression modeling



User 7, 1 hidden layer NN

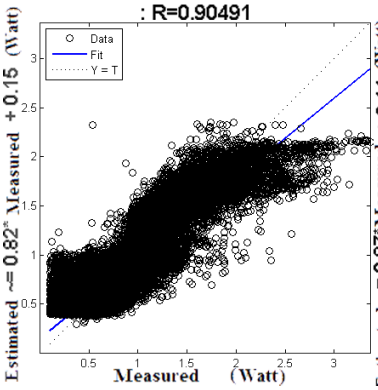


User 7, 2 hidden layers NN

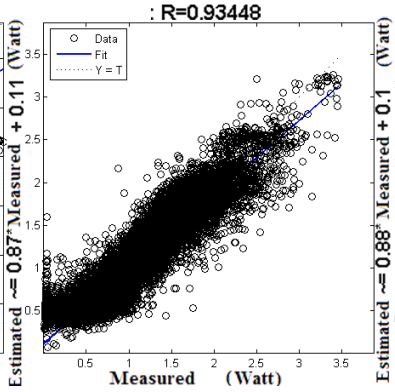


User 8, regression modeling User 8, 1 hidden layer NN User 8, 2 hidden layers NN

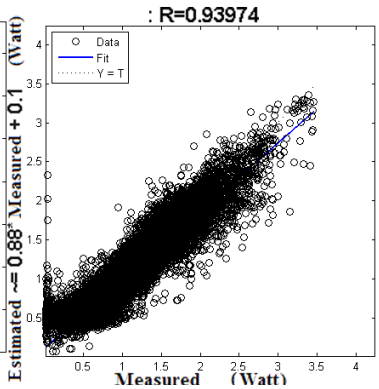
Appendix-B: User-level power modeling performance comparison



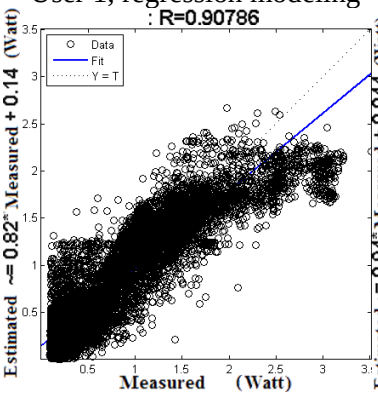
User 1, regression modeling



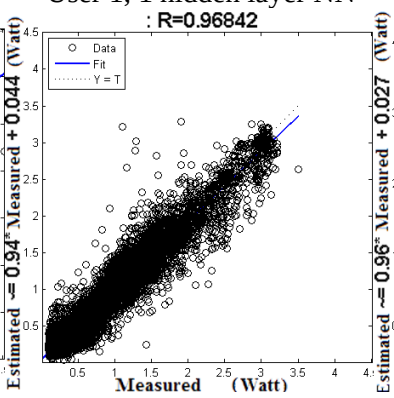
User 1, 1 hidden layer NN



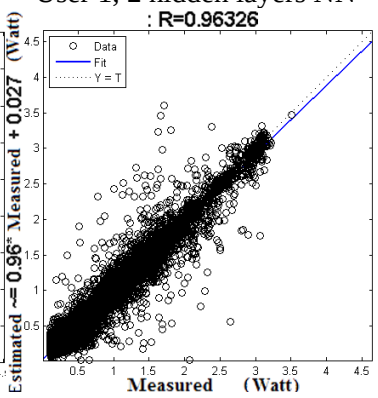
User 1, 2 hidden layers NN



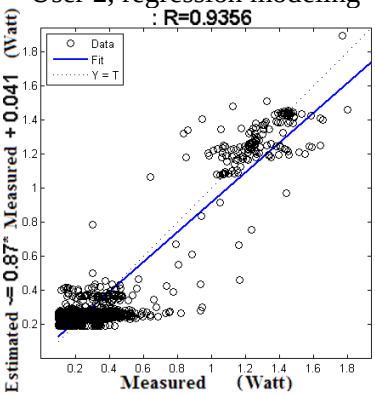
User 2, regression modeling



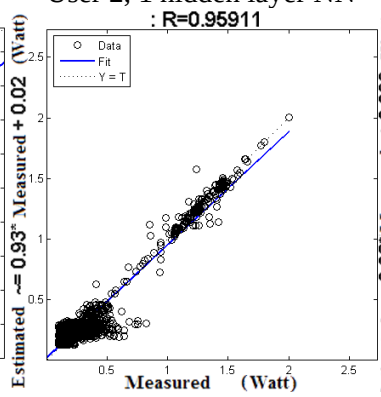
User 2, 1 hidden layer NN



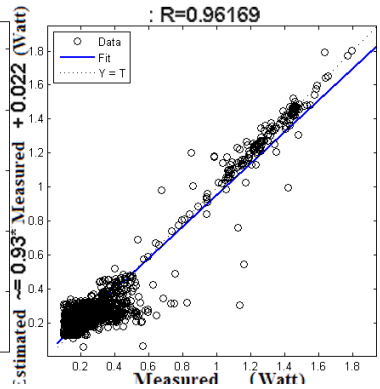
User 2, 2 hidden layers NN



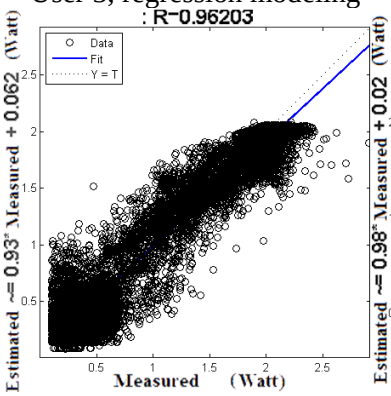
User 3, regression modeling



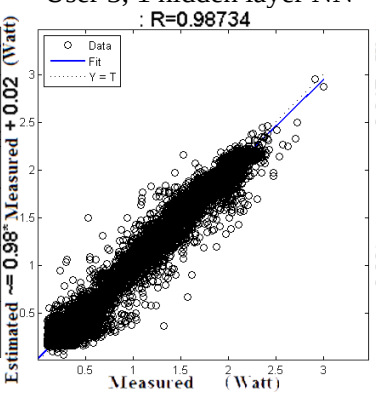
User 3, 1 hidden layer NN



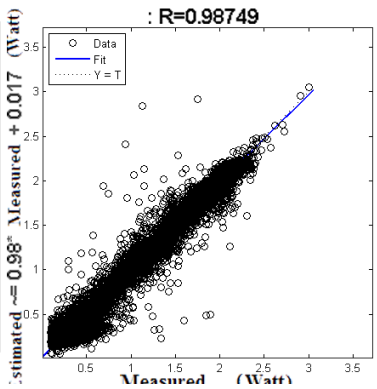
User 3, 2 hidden layers NN



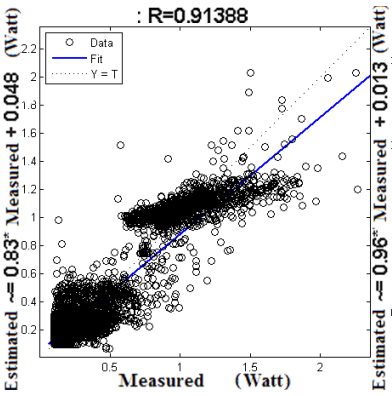
User 4, regression modeling



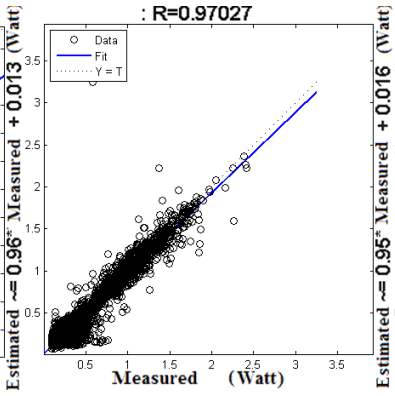
User 4, 1 hidden layer NN



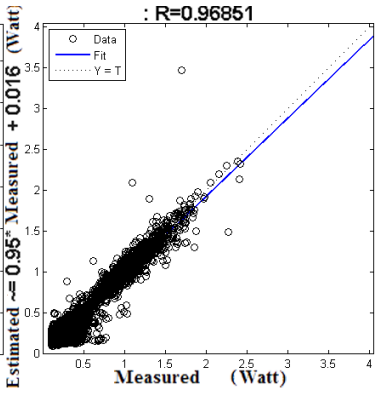
User 4, 2 hidden layers NN



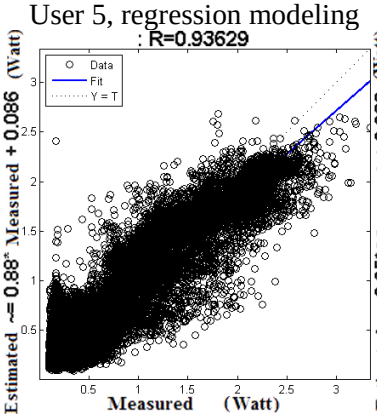
User 5, regression modeling



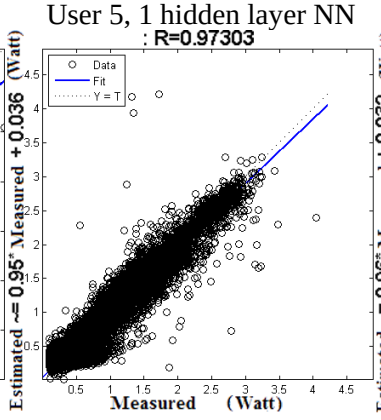
User 5, 1 hidden layer NN



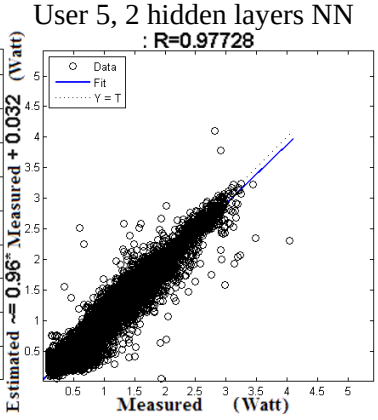
User 5, 2 hidden layers NN



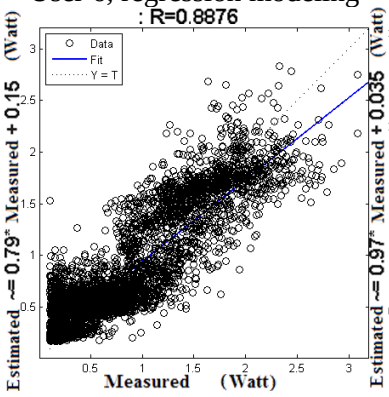
User 6, regression modeling



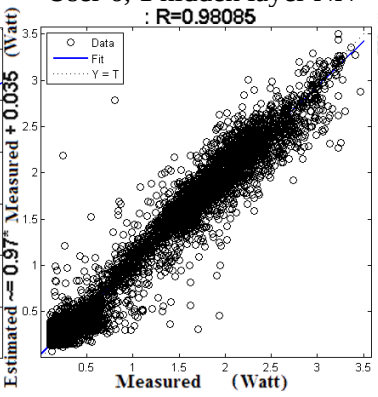
User 6, 1 hidden layer NN



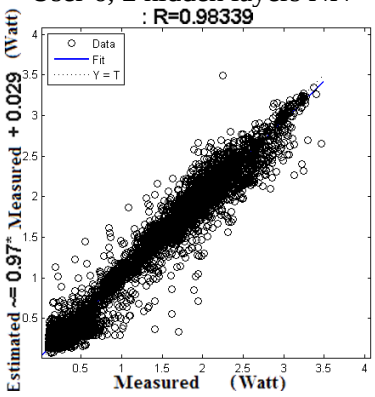
User 6, 2 hidden layers NN



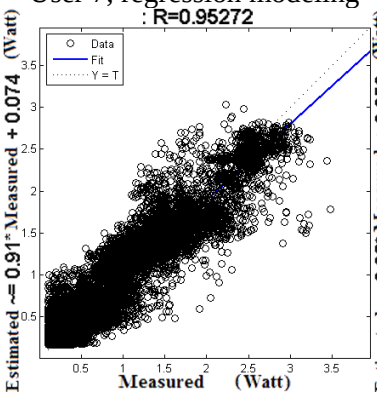
User 7, regression modeling



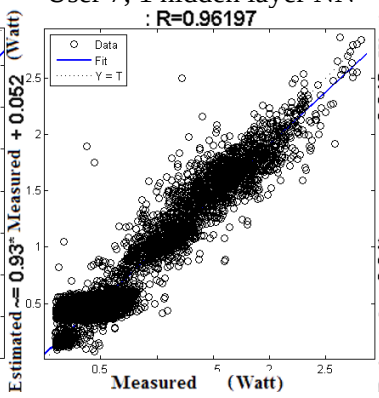
User 7, 1 hidden layer NN



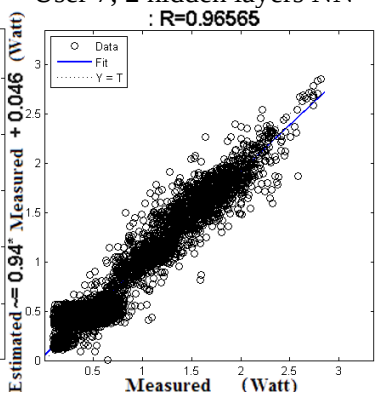
User 7, 2 hidden layers NN



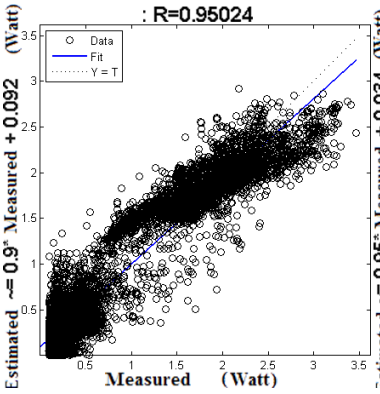
User 8, regression modeling



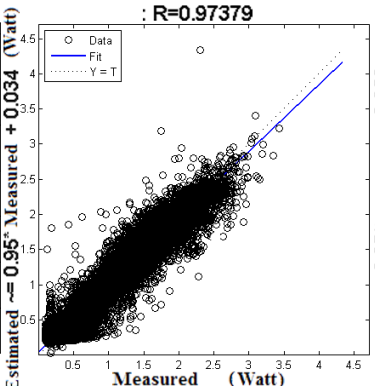
User 8, 1 hidden layer NN



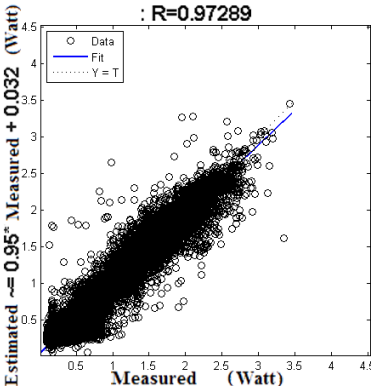
User 8, 2 hidden layers NN



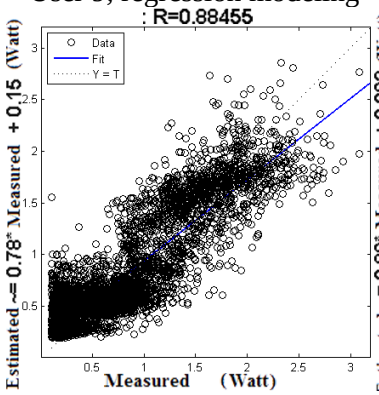
User 9, regression modeling



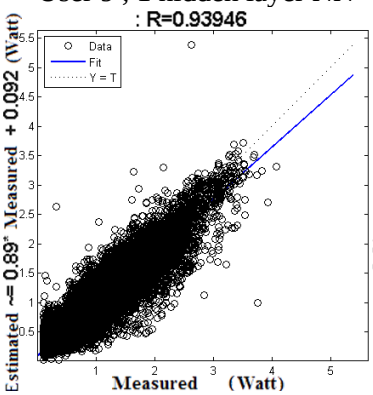
User 9 , 1 hidden layer NN



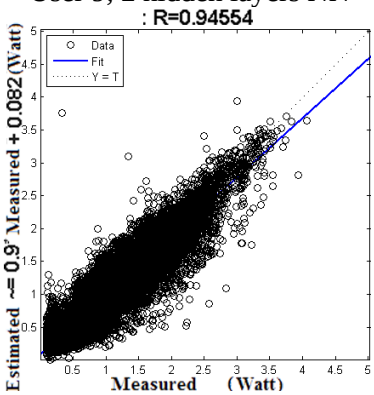
User 9, 2 hidden layers NN



User 10, regression modeling



User 10 , 1 hidden layer NN



User 10, 2 hidden layers NN

Vita

Sameer Allan Alawnah was born on December 17, 1985, in Nablus, Palestine. After completing his high school education in 2004, he joined the Computer Engineering program at An-Najah National University in Palestine from which he graduated in 2009. Mr. Sameer received a graduate teaching assistantship to join the master's program in Computer Engineering at the American University of Sharjah. He was awarded the Master of Science degree in Computer Engineering in 2013.