Creating Usage Models to Identify Misbehaving Applications on Mobile Devices

by

Qiushi Jiang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

© Qiushi Jiang 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Limited battery capacity is currently a major pain point for mobile users. The problem is made worse when poorly designed applications consume a significant amount of power in the background when they are not actively used by the user. To combat this problem, we propose an automated monitoring system that can detect misbehaving applications running on mobile devices. Our system does not require any prior knowledge about the monitored applications. Instead, it collects the user's usage records and builds models to encapsulate the contexts when the user is likely to use each application. From those models, our system can identify misbehaving applications that are consuming system resources while providing no useful service to the end user. In this dissertation, we demonstrate the overall design for our system. This design allows us to collect detailed usage records while keeping our system's power consumption at a minimum. We also introduce the steps we take to construct our usage models and the rationale behind each key decisions. In the end, we evaluate the effectiveness of our system by running it on a real Android device during a two month period. From the experiment, we show the misbehaving applications identified by our system have a significant impact on the battery life, and misbehaving applications with high network usage is the main cause of fast battery drain.

Acknowledgements

I would like to thank my academic advisor Dr. Paul Ward for his continuous guidance and support during the last two years.

I would also like to thank Dr. Bernard Wong for the feedback he has provided during all the Shoshin group meetings.

Last but not least, we would like to thank my colleague Liuyang Ren for helping me testing the early versions of my automated monitoring system.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Existing Solutions 1

1.2 Thesis Contributions 3

1.3 Thesis Organization 4

2 Background 5

2.1 Defining Misbehaving Application 5

2.2 Battery Consumption of On-board Components 6

2.3 Running Background Tasks on Android 6

2.4 The App Review Process for Publishing on Google Play Store 7

2.5 Machine Learning Algorithms 8

2.5.1 K-Nearest Neighbors 9

2.5.2 Decision Tree Learning 9

2.5.3 Neural Network 10

3 Related Works 12

3.1 Existing Automated Monitoring Systems 13

3.2 Usage Modeling on Mobile Devices . . . 14
3.3 Battery Consumption of Popular Mobile Apps . . . 16
3.4 Existing Battery Saving Features on Android . . . 17

4 System Design 19
4.1 Challenges . . . 19
4.1.1 Minimizing System's Impact on Battery Life . . . 20
4.1.2 Uncontrolled Environment . . . 20
4.1.3 Inexact Time of Code Execution . . . 20
4.2 Design Principles . . . 20
4.3 System Architecture . . . 21
4.3.1 Data Gathering . . . 22
4.3.2 Data Uploading and Model Creation . . . 24
4.3.3 Detection and Outputs . . . 26

5 Model Creation 31
5.1 Data Preprocessing . . . 31
5.1.1 Quantizing Time and GPS Location . . . 31
5.1.2 Restructuring Usage Data . . . 33
5.1.3 Normalizing Numerical Features and Balancing Dataset . . . 33
5.2 Data Exploration . . . 35
5.3 Model Selection . . . 37
5.3.1 Testing Environment . . . 37
5.3.2 Detection Confidence and Accuracy . . . 38
5.3.3 Training, Validation and Testing . . . 39
5.3.4 Parameter Tuning . . . 39
5.3.5 Comparison of Results . . . 42
5.4 Size of Training Data . . . 43

6 System Evaluation **44**
6.1 Model Evaluation 44
6.2 Measuring the Negative Impact of Misbehaving Applications on Battery . . 45
6.3 A Closer Look into Some of The Misbehaving Applications 50
6.4 Our System's Battery Overhead 52

7 Conclusion and Future Works **53**
7.1 Future Works 54
7.2 Broader Implication 54

References **56**

List of Tables

2.1 Power Consumption from Different On-board Components 6

5.1 Prediction Results on the Testing Data 42

6.1 Model Performance 45

6.2 Battery Consumption of Consistently Misbehaving Applications (According to Network Usage) 50

6.3 Top Misbehaving Apps Ranked by Data Received 51

6.4 Client-side's Daily Battery Overhead 52

List of Figures

2.1 K-Nearest Neighbors Example 9
2.2 Decision Tree Example 10
2.3 Multilayer Feedforward Neural Network 11

4.1 System Design Overview 22
4.2 Raw CPUinfo Data Returned by Dumpsys 23
4.3 Raw Netstats Data Returned by Dumpsys 24
4.4 Client-server Interaction: Best-case Scenario 25
4.5 Client-server Interaction: Error Cases 27
4.6 Detection Logic Flowchart 29
4.7 Client-side Application UI 30

5.1 Incomplete Usage Record 32
5.2 Dataset Restructuring 34
5.3 App Usage Example 36
5.4 App Usage Top-down View 36
5.5 App Usage Side View 37
5.6 Model Evaluation Using Moving Windows 40
5.7 Size of Training Data vs. Detection Confidence 43

6.1 Battery Consumption Throughout One Day 46
6.2 Usage Record Counts Throughout One Day 46

6.3 Number of Misbehaving Application vs. Speed of Battery Drain 48
6.4 Number of Misbehaving Application (CPU) vs. Speed of Battery Drain . . 48
6.5 Number of Misbehaving Application (Network) vs. Speed of Battery Drain 49

Chapter 1

Introduction

Nowadays, software systems can be found in every facet of our lives, and a large number of them run on our mobile devices. We rely on them to provide us with weather reports, the best routes to our destinations, messages from our friends, and many other important pieces of information we need in our daily lives. As people delegate more responsibility to software, the software's complexities have also increased dramatically. For example, a simple Google search on an Android device could result in the browser creating more than 30 threads, and involve various onboard components such as WiFi, 3G/4G and GPS [21]. With such a high level of complexity, it is unavoidable that some software has bugs in it which leads to unexpected behaviours. Keeping those systems running smoothly requires a large number of highly trained system administrators. In order to effectively manage software, administrators not only need to have the technical know-how, but also the domain knowledge associated with each application. The problem becomes more difficult when the software itself is also constantly changing due to updates. This makes purely relying on system administrators a very costly solution for large corporations and simply not viable for small businesses or individual users. To combat this problem there has been an increasing interest in developing automated solutions for system monitoring and fault detection.

1.1 Existing Solutions

Existing automated monitoring systems can be categorized into two general types. Those that require input from the developers of the monitored software, and the those that do not. Most of the monitoring systems currently used in the industry are the first type [12].

Developers understand their own software's internal structure. With this knowledge they are able to surface the most meaningful information about the states of a software system, and they can create watcher programs in charge of monitoring that information. The watchers are simple monitoring programs that periodically query various system metrics, and flag any problems according to the rules previously defined by the developers. Such monitoring systems are generally very effective because those rules can encapsulate the unique expertise provided by the software's developers. The downside is that the watcher programs need to be custom built for each monitored software; therefore, they are very costly to deploy. This is especially challenging when the software's current operators do not have access to the original developers.

To address the problem above, researchers have proposed automated monitoring solutions that do not require inputs from software's original developers. In general, those solutions achieve automated monitoring by first creating a model that represents a software's normal operating states. Once the model is created, a monitor program can periodically compare the current state of the software against the model. Problems are reported if there are significant discrepancies between the model and the software's current state. One method for creating such a model is by looking for correlations between all the metrics collected from the running software. Previous work has shown that many long lasting correlations can be found between system metrics under normal conditions [10, 13]. A break of the correlations could indicate something is wrong within the system. If we know the component where each metric is collected from, we can also localize the problem to a few specific components within the system. Such solutions are easier to deploy than the previous methods because these solutions no longer require any involvement from the software's original developers. The operators do not need to write any customized rules to help a monitor in interpreting the system metrics. As long as the operator can provide the monitor with the system metrics under normal operational conditions, the monitoring system should be able to automatically build its model without any additional human oversight.

So far the existing solutions have shown promising results in managing various cloud services running in data centers, but there are still areas that could be improved upon. The solution proposed by Jiang [13] does not need any inputs from the software's original developers, but it does require the operators to be able to identify the normal operational states of the software. In some situations, this could be difficult for the operator. For example, on a client-side Android device, there might be poorly designed software constantly wasting precious system resources. Such software consumes battery life and cellular data while providing no meaningful benefits to the user. Since Android applications are able to run in the background, the user may not be able to detect their presence. In those kinds

of situations the previous solution would not work because the operator could not provide the system metrics under normal operational states, and the existing solution needs that training data in order to train its model. This means, if we want to create a similar automated monitoring solution for client-side mobile devices, we would have to construct the model through other means which leads to our solution.

1.2 Thesis Contributions

In this dissertation, we present an automated solution for detecting misbehaving applications running on Android devices. Comparing to the previous works, we do not require the operator of the software to provide us with any system metrics under normal conditions or any custom built watcher programs. Instead, our solution automatically studies the user's usage patterns and builds its models based on the data it collects from the user's interactions with the device. Our specific contributions can be organized into the following points:

- We introduce a solution framework for identifying misbehaving applications running on mobile devices. The solution is based on modeling the user's usage patterns and comparing the model against the applications that are found running on the device.
- We propose an effective way to model user' usage patterns using information such as location and time. We examine real-world usage data, demonstrate various data processing steps we take, and also discuss the rationale behind how we select machine learning algorithms for our use case.
- We highlight various challenges a developer could face when collecting usage data and system metrics from a mobile device. We then demonstrate the design of a robust monitoring system that creates minimum overhead, and is able to provide meaningful information even when working with an imperfect dataset.
- We demonstrate our solution's benefits to the end users by showing the negative effects misbehaving applications have on battery life.
- By studying the misbehaving applications detected during our experiment, we find misbehaving applications with high network usage consume more battery than misbehaving applications with high CPU usage. The causes for high network usage are ad pre-fetching, data harvesting, and push notification. We also find, contrary to popular belief, social network apps such as Facebook rarely upload the user's data

in the background. Instead, most of its network usage is likely spent on pre-fetching advertisements.

1.3 Thesis Organization

This dissertation is organized into the following chapters:

- **Chapter 2** provides a brief description about all the background information a reader needs in order to follow this dissertation.
- **Chapter 3** describes prior research related to user modeling, automated system monitoring and battery usage on mobile devices.
- **Chapter 4** shows the high-level architecture of our system, and some of the design principles we aim to follow throughout the development.
- **Chapter 5** explores the raw usage dataset collected during our experiment, and discusses various steps we take to construct our models.
- **Chapter 6** evaluates the effectiveness of our system. It demonstrates the accuracy of our models, and measures the negative effects that the misbehaving applications have on a device's battery life.
- **Chapter 7** discusses some potential future improvements for this work and the broader implication of similar monitoring systems.

Chapter 2

Background

In this chapter, we define the term “misbehaving application”, and explain the reason we choose this particular definition for our work. Since reducing battery consumption on mobile devices is one of the main goals for our monitoring system, a breakdown of battery consumption by mobile devices' on-board components is provided in this chapter. The monitoring system presented in this thesis is developed for Android devices. To help the readers understand the system environment, we present a brief description of various types of background processes supported by the Android operating system. We also discuss the app review process developers need to go through when publishing applications on the Google Play Store. Finally we include a brief survey of the machine learning algorithms we have experimented with for training our usage models.

2.1 Defining Misbehaving Application

In this thesis, the term “misbehaving application” is defined as follows:

- A ***misbehaving application*** is a background application that consumes a significant amount of system resources, such as battery and cellular data, while providing no useful service to the end user.

The reason we focus on system resources is because, unlike data centers with abundant computing power, client-side devices, such as smart phones, are often limited by their onboard batteries and Internet connection [9]. If we could reduce the battery and cellular data consumption without affecting users' regular usage, we would not only improve the overall user experience, but also reduce the cost of owning a smart phone.

2.2 Battery Consumption of On-board Components

Modern mobile devices contain many different on-board components, such as, screen, 3G, WiFi, GPS and many others. In order to get a general sense on how much energy each component consumes, we have compared measurements from two different groups of researchers [15, 22]. The measurements are taken from a range of devices including Google Nexus S, Hongmi and Nokia N95. The general ranges of each component's power consumption are shown in Table 2.1. Note, the measurements taken by the two groups of researchers are relatively close for most of the components with the exception of CPU. This is because the measurements from Perrucci et al. [22] are taken on Nokia N95 which is an older device with a slower CPU (332 MHz) comparing to Google Nexus S (1GHz) and Hongmi (1.5GHz).

Components	Power (mW)
CPU (full load)	612 - 1851
WiFi (transmitting)	1450 - 1471
3G (transmitting)	1338 - 1400
GPS (active)	1006 - 1293
Bluetooth (transmitting)	425 - 785
Screen (50% brightness)	414 - 462

Table 2.1: Power Consumption from Different On-board Components

2.3 Running Background Tasks on Android

As of 2017, there are about 2.7 million applications listed on the Google Play store [6]. In order to support the needs of all the application developers, through out the years Google has introduced different ways to run background tasks on the Android operating system. This makes it possible for developers to build more complex applications, but also opens the door for some applications to abuse the system and waste valuable system resources. In order to address this problem, Google has placed various restrictions to on background tasks [1]. Unfortunately, as we see in the later chapters, it is still possible for applications to abuse the system, but before we can talk about those misbehaving applications, we first need to discuss some common ways an application could run its background tasks on the Android platform:

- ***Background Service:*** Apps running on Android can start a type of application components called a background service. The background service is mainly designed to handle long-running operations that are not directly visible to the end users. On older versions of Android, an application can keep its background service alive even when the application itself is not running in the foreground. This behaviour has been modified in the more recent API level 26. Now a background service can only run for a few minutes after its parent application has left the foreground.

- ***Foreground Service:*** Similar to the background service, a foreground service can also handle long-running operations. The difference is a foreground service can be kept alive even after its parent app has been closed by the user. Unlike the background service, a foreground service is not completely unnoticeable to the user. The application's icon is displayed in the notification center as long as its foreground service is running.

- ***WorkManager Library:*** The Android platform provides a library called WorkManager. By using WorkManager, apps can register tasks which are deferrable. Once a task has been registered on the WorkManager, it would be carried out by the library even after the user has restarted the device. WorkManager also offers developer the ability to define a set of conditions when work can be triggered. Those conditions include network availability, battery status and etc. The WorkManager only runs a task when all its conditions are met.

- ***AlarmManager Library:*** The AlarmManager is used when an app needs to run tasks with precise timing. Similar to WorkManager, once a task has been registered, the app does not need to stay alive in order for the task to be triggered. Although AlarmManager is designed to run tasks at specific times, the recent versions of Android has put on more restrictions on the AlarmManager. For certain use cases, the library no longer guarantees precise execution times. The rationale behind this change is further discussed in section 3.4.

2.4 The App Review Process for Publishing on Google Play Store

Quality control has traditionally been an important step for preventing faulty and harmful products from entering the market. In this section, we introduce the app-review process

developers have to go through before their apps can be published on the Google Play Store. The followings are some of the key areas Google looks at during the review process.

- **Store Listing:** When publishing on the Google Play Store, every developer needs to create a store listing for their app. The information on the listing page is later reviewed by reviewers from Google to ensure the app matches its description.
- **Privacy Policy:** If an application collects personal and sensitive data from users, the developer needs to provide a privacy policy along with the app's store listing. The reviewers from Google are instructed to reject any app that collects sensitive information but does not have a privacy policy.
- **Content Rating:** Developers are asked to fill a content rating questionnaire when publishing their apps. The questionnaire contains questions such as "does the app contain adult content?", "does the app have reference to tobacco?". Based on the self-reported answers, the Google Play Store automatically assigns a content rating to the uploaded app.
- **App Permissions:** After an app is uploaded, Google automatically generates a list of system permissions required by the app. This list is constructed based on the permissions declared by the app inside its manifest file. The permissions are later shown to each user for approval before the app can be installed. Along side each permission, there is also a short description explaining what kinds of actions the permission would allow the application to perform.

As we can see the review process from Google requires developers to disclose information on the sensitive data their apps may collect, the types of content they serve and the permissions their apps require [5]. What is missing from the current review process is a closer look at each app's runtime behaviour. For example, we may have two applications that both require Internet permission. One could use the permission responsibly and only requests data when needed, and the other might constantly download data in the background regardless whether the app is used by the user. As of right now, Google does not take the wasteful behaviour of the second app into account during its review process.

2.5 Machine Learning Algorithms

In order to model users' usage patterns we have experimented with machine learning algorithms including k-nearest neighbors, decision tree, and neural network. In this section, a brief introduction is provided for each of the algorithms used in this work.

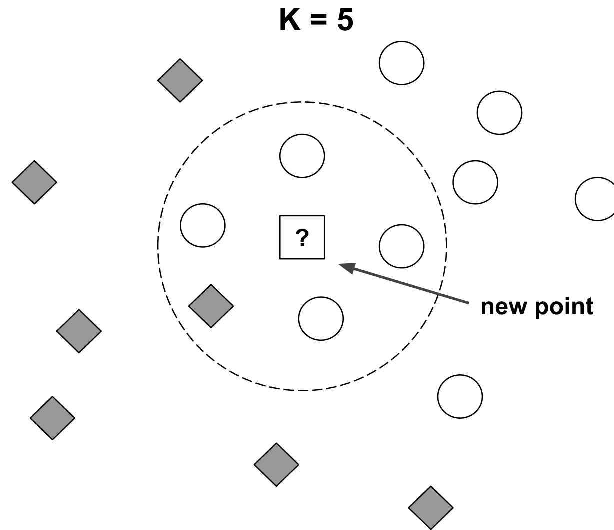


Figure 2.1: K-Nearest Neighbors Example

2.5.1 K-Nearest Neighbors

K-nearest neighbors is one of the simplest classification algorithms available. Models created by the K-nearest neighbors algorithm simply store all the instances of the training data. When a new data point is given, the model classifies the point by first finding a set of "k" neighbor points that are found nearest to the new point in the training data. A label is then assigned to the new point based on the most common label found among its nearest neighbors [8]. For example, in Figure 2.1 the new point would be classified as "circle" since the majority of its nearest neighbors are circles. In order to compare the distance between sample points, the nearest neighbor algorithm requires a distance function. In most cases, the simple Euclidean distance is used as the distance function, but there can also be other distance functions such as Manhattan distance, Chi-Square distance, Cosine distance, and Minkowski distance. Despite its simplicity, k-nearest neighbors can often achieve impressive results in many areas when it is supplied with an appropriate distance function [26].

2.5.2 Decision Tree Learning

Decision tree learning is another machine learning algorithm commonly used for solving classification problems. The algorithm aims to break complex problems into a set of simple

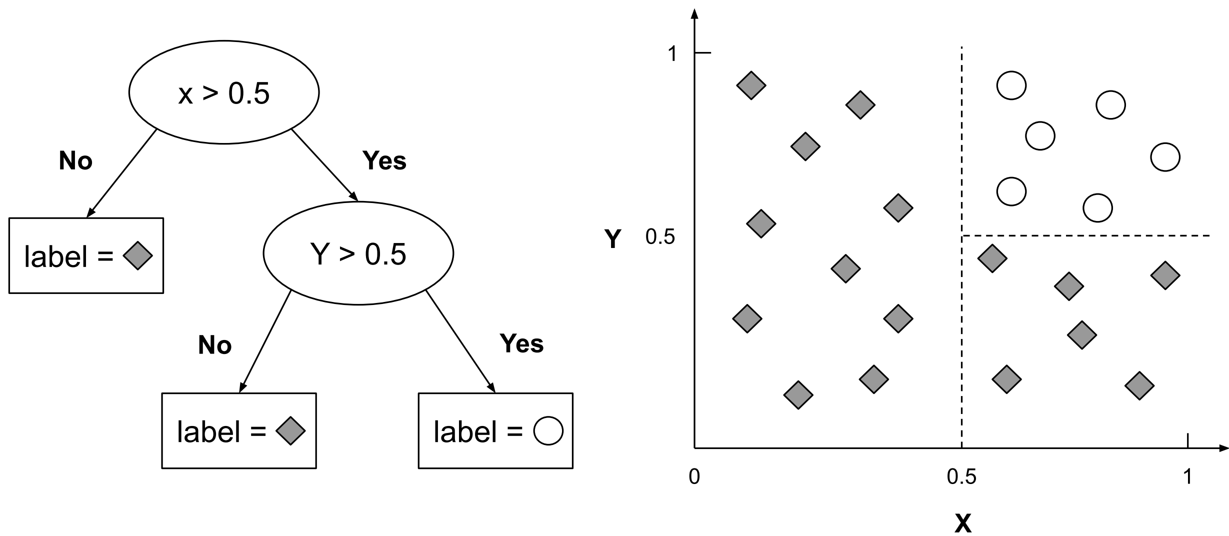


Figure 2.2: Decision Tree Example

questions represented using decision trees, and hoping by answering those simple questions we can arrive at a desirable answer [24]. When constructing decision trees, the goal is to ask the question that could achieve the cleanest split in the dataset. The quality of a split is evaluated based on the impurity of the resulting subdatasets. By finding the best split at each node, the tree can gradually reduce the impurity left at its leaf nodes, and eventually be able to assign a label to each leaf node with high confidence. An example of this process is shown in Figure 2.2. The decision tree shown in the figure contains two questions. The first question splits the dataset into two subsets. One subset with x value less or equal to 0.5, and another subset with x value greater than 0.5. After the first split, the subset on the left has no impurity left; therefore, no additional question is needed for this subset. On the other hand, the subset on the right still has impurity left; therefore, an additional question is asked by the decision tree. This time the remaining dataset is split based the Y value of each point. After this split, the resulting subsets are both free of impurity, and the construction of the decision tree is complete.

2.5.3 Neural Network

Neural network is a powerful machine learning algorithm that is capable of solving complex classification problems. It takes inspiration from nature, and aims to mimic the way the human brain works. The network is constructed using simple computing cells called

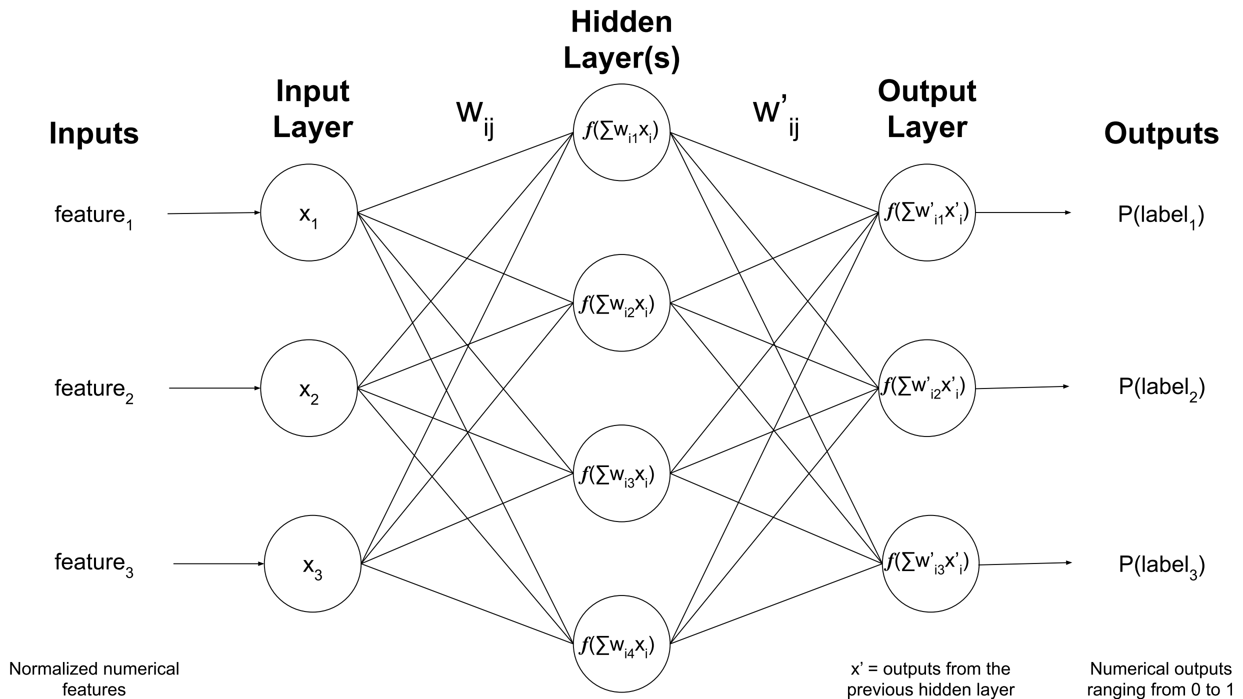


Figure 2.3: Multilayer Feedforward Neural Network

"neurons". By connecting those neurons in different ways, the network is capable of storing knowledge which can later be used to solve problems [11]. There are several types of neural networks, but the multilayer feedforward network is the most widely used model among them all [16]. The multilayer feedforward network consists of several layers of neurons, as illustrated in Figure 2.3. Those layers can be separated into three types: input layer, hidden layers and output layer. As their names suggest, the input and output layers hold the values for the network's inputs and outputs. The hidden layers sit between the input and output layers. The values inside each hidden layer are calculated based on the values found in the proceeding layer. By doing so, neural network can extract higher level concepts from the inputs it is given, and as the number of hidden layer increases, the network is capable of modeling even more complex concepts based on the values from the previous hidden layers. This behaviour has proven to be very useful in the area of image recognition [14, 20]. When analyzing digital images, the value of each individual pixel holds very little information, but as the number of hidden layers increases, the neural network starts being able to capture dots, lines, shapes and eventually everyday objects.

Chapter 3

Related Works

When surveying related works, we find most of the previous monitoring solutions are designed for monitoring server-side applications. Those solutions have varying degrees of autonomy. Some can function with little involvement from human operators, and others may require custom "watcher" programs written specifically for each monitored software. The solutions with higher level of autonomy generally rely on studying relationships found between system metrics under normal operating conditions. By analyzing past data, the monitor can build models to capture long-lasting relationships exist between the various system metrics generated by the monitored software. Those models are later used to identify faulty software states through spotting anomalies in the system metrics.

On the client-side, many efforts have been made in user behavior modeling, but so far, those models are mainly designed for purposes such as predicting which apps users are likely to use next for reducing loading time, and predicting apps installation for marketing purposes. The potential of utilizing usage models in developing automated monitoring system has not yet been widely studied. In the area of mobile battery, researchers have looked into the battery consumption of popular mobile applications, and discovered a few major causes of fast battery drain. Based on the findings from those research, developers at Google have gradually introduced several battery saving features into the Android operating system, but in order to avoid breaking existing Android applications, those changes so far have been fairly conservative, and still leave room for improvement.

3.1 Existing Automated Monitoring Systems

Automated monitoring systems can be found in most of the modern data centers. For example, Google uses a cluster management system named Borg to manage virtually all its cluster workloads [25]. For each task running under Borg, there is a built-in HTTP server that publishes performance metrics about the health of the associated task. By reading those performance metrics, Borg is able to identify whether a task has stopped running, and if it needs to be restarted. Since Borg does not know exactly what each task is designed to do, it could not detect any service-level objective (SLO) violation. To detect SLO violations, the owner of each task has to implement a monitoring system outside of Borg's framework.

Microsoft has a similar system named Autopilot that manages its own data centers [12]. In the area of fault detection, Autopilot uses an existing concept called "watchdog". Watchdogs are programs capable of interpreting system metrics from the machines running under Autopilot. There can be more than one watchdog program for each machine, and if any of them reports an error status, the machine would be considered as running in error. Similar to Borg, Autopilot supplies a set of default watchdog programs which can perform basic non-service-specific detections, but what sets it apart is the ability to incorporate additional service-specific watchdog programs. Instead of requiring developers to create customized monitoring solutions outside of Autopilot's framework, developers can create their own watchdogs managed by the Autopilot, and take advantage of the additional features provided by the framework, such as logging and auto-restart.

Borg and Autopilot are two examples of the automated monitoring solutions currently used in the industry. Both Borg and Autopilot have been managing production level software for many years and have become crucial parts of Google's and Microsoft's respective infrastructures. Unfortunately, they both share the same limitation that is requiring developers to write customized monitoring program in order to perform service-level fault detection. This is less of a problem for Google and Microsoft because most of their applications are developed in-house, and they have the resources and technical know-how to create customized monitors for each of their applications, but this may not be the case for other smaller companies or individual operators.

To address the limitation found in Borg and Autopilot's monitoring systems, Jiang et al. [13] have proposed an automated monitoring framework that requires minimum knowledge about the monitored software. Their work is based on the observation that long-lasting relationships can be found between various system metrics while software is operating under normal conditions. Instead of relying on custom-built "watcher" programs,

their monitoring system collects the system metrics produced by the monitored software, and acquires its application-specific knowledge by identifying the long-lasting relationships between those metrics. In order to capture both linear and non-linear relationships, Jiang et al. have utilized linear regression models along with information-theoretic models. The monitoring system is first supplied with system metrics collected under normal operating conditions. By using past system metrics as the training dataset, the system is able to create a set of models, and flag potential errors in the system by comparing the models with the most up to date system metrics. A potential problem is flagged when the current system metrics diverge from the existing models. Since each metric can often be linked to a set of underlying system components, Jiang's solution can not only perform error detection, but also provide useful information for locating the root problem. During their experiments, the solution has shown promising results on monitoring complex server-side applications. That said, there is still a major limitation preventing this solution from working on certain environments. The limitation is the solution's dependency on the previous system metrics collected during normal operating conditions. In order to collect those system metrics, the operators need to have to capability to manually identify whether the software is operating under normal conditions. In some environments, the operators may not have the capability to do so.

3.2 Usage Modeling on Mobile Devices

One of such environments is the mobile platform. There are many differences between client-side environment and server-side environment as Rudafashani et al. have pointed out in their research [23]. For example, difference in processing power and memory capacities, difference in number of services running in the environment, lack of replication on client-side, etc. Among them, the two following differences are the most relevant to our work:

- Unlike data centers where a machine is often tasked with serving one specific application, for mobile devices, a smartphone on average has more than 80 apps installed [4]. Many of those apps have complex behaviours which can utilize various onboard components.
- The operators of the server-side machines are professionally trained system administrators, but for mobile devices, the operators are ordinary end users with varying levels of expertise.

Due to the high complexity from severing more than one application, and the operators' lack of experience, it can be very difficult for the average smartphone users to tell whether a

device is operating normally at a particular time; therefore, any solution requiring previous system metrics under normal operating conditions would not work in this case. This means, we would have to construct our models through other ways. One potential solution is to study the users' usage patterns in real-time and derive the models based on the usage records. Some previous works have already been done in the area of modeling people's usage patterns on mobile devices, and we believe similar models could also used on building automated monitoring systems.

Yan et al. have designed and built a system called Falcon to perform app-specific pre-launching on Windows phones [27]. They achieve this by modeling the contexts when each app is likely to be used by a user. The contexts are described by features such as location, time-of-day and the trigger app of the current usage session. From real world usage records, they have observed users' tendency toward structured behaviours. For example, they notice people have high game usage at home, high browser and calendar usage at work, and high social-network usage at shopping centers. They also notice users' behaviours may change significantly over longer time frames. This is especially common for game and entertainment apps. For those apps, users tend to use them intensely during a month-long period, then stop using them afterward. By capturing those behaviour patterns and prelaunching apps accordingly, Yan et al. manage to reduce the average launch time of mobile apps by 50% while incurring only a 2% battery overhead.

Similar to Falcon, Parate et al. also build a prediction system for prelaunching mobile applications [19]. Instead of relying on the contextual data such as location and time-of-day, Parate proposes an approach utilizing both Prediction by Partial Match (PPM) model and Time Till Usage (TTU) temporal model. Previously PPM has been widely used in the field of text compression [7]. Parate adapts the PPM method to app prediction by treating each app as a "character" and a user's usage history as a sequence of characters. Similar to predicting the next character when given an incomplete word, Parate uses the usage history to predict the apps which are likely to be used next. Along with the PPM, the TTU temporal model is used to answer when the next app is likely to be launched. This is done by studying the historical usage data to find the distribution of time spent between the launch of each application. Combining both models, Parate's approach has achieved an above 80% prediction accuracy while only requiring 7 to 10 days of training data.

Unlike works that are done by Yan and Parate where the models rely on features collected locally on each device, Pan et al. have developed a model for predicting mobile application installation using social networks [18]. Their model is based on the assumption that social networks correlate with individual behaviours. In their work, they use a composite social network which is a mixture of four separate networks constructed us-

ing various types of data. The networks include call log network, Bluetooth proximity network, friendship network (based on Facebook friends), and affiliation network (based on GPS location). In the network, each user is represented as a node, and relationships between users are represented as weighted edges. By examining the applications installed on each user's neighbor nodes, the model aims to predict the applications the user is likely to install in the future. This information can later be used by App market makers such as the iPhone AppStore to perform targeted advertising. From their experiment, they find the model can achieve a mean precision (# true adopters among total predicted adopters / total predicted adopter) of 0.31, and a F-score of 0.43.

3.3 Battery Consumption of Popular Mobile Apps

Beside usage modeling, we have also looked into work around energy accounting on smartphones. One of our goals is to identify applications which might be wasting energy doing non-essential work; therefore, it is important for us to understand how popular mobile applications consume battery.

In this area, Pathak et al. from the Purdue University have designed a fine-grained energy profiler for mobile apps called Eprof [21]. The profiler tracks the energy consumption of each hardware component and maps them to various software entities of each application. Depending on the user's needs, the software entities can range from processes to threads, to subroutines, and all the way down to individual system calls. By using Eprof, the researchers are able to study the energy consumption of popular apps such as Angry Birds, Facebook and Google Chrome. The followings are some of their observations:

- A major portion of the battery is spent on I/O. This includes components such as WiFi, 3G and GPS.
- I/O components exhibit tail-energy behaviour. It means once a component enters a high power state, the component would remain in this state for a period of time even after the task has already been completed. The behaviour is observed on components such as WiFi and 3G. This is because every time after a packet is sent, the antenna would remain active for a period of time in order to capture any response packets.
- For popular apps such as Angry Birds, CNN and Facebook, 65% - 75% of the energy is spent on third-party advertisement and analytic modules.

These observations shed light on how battery is consumed on smartphones, and some of those findings have already helped Google improving their Android operating system.

3.4 Existing Battery Saving Features on Android

As one of the major developers of Android, Google has been gradually adding API policies and monitoring features to improve battery life for Android devices. The approaches they have taken so far can be summarized into the following three groups:

- ***Limit System Wake Time:*** Since Android 6.0, the Android operating system has become more aggressive in restricting background processes. Android gives each application the ability to create additional threads while running in the foreground, but once the application leaves the foreground, it only has a few minutes before the operating system stops all its threads completely. Beside shutting down unessential background threads, the operating system also puts restrictions on long lasting background processes. On the newer versions of Android, applications with repeating background tasks can no longer set an exact time interval for the repetition. By making this change, the operating system now has the flexibility to group multiple repeating tasks together to run one after another. This change allows tasks to share the overhead of waking up the CPU, and further minimizes the time when the CPU is awake.

- ***More Transparent Battery Accounting:*** Despite all the restrictions Android has placed on background processes, if an application really needs to keep the device awake, it can still do so by acquiring the device's wake lock. Since there are legitimate use cases built on this feature, Android could not just take it away without breaking existing applications. In order to prevent applications from abusing the wake lock, Android's development team takes the duration that an application holds the wake lock into account when assigning battery consumption to each application [2]. The power consumption assigned to the usage of wake lock is estimated by multiplying the wake time with device's average power consumption when CPU is idle [3]. This way if an application has been hogging the wake lock, it would show up with a high battery usage in the device's settings page, and this make users more likely to uninstall the app if they think the app has not been doing useful work. Note, with the current battery accounting logic, it is possible to double count the power consumption caused by wake lock. This happens when more than one application are holding the wake lock at the same time. The reason Android allows this to happen is likely because the main purpose of the built-in battery accounting system is not to provide the most accurate measurements, instead it is meant to create an incentive for developers to take battery consumption into consideration when designing applications.

- ***Adaptive Battery:*** In Android 9.0, Google has also started experimenting with the idea of limiting background processes based on the users' usage patterns. This adaptive battery feature promises that it would learn "how you use apps over time", and "limit battery for infrequently used apps". At the time of writing this dissertation, Google has not published any detailed documentation on how exactly their system works. From our limited testing, we found the adaptive battery system so far only limits apps which have not been used by the user for a long period of time, roughly around a month. For any other apps, the adaptive battery feature currently has no effect. During our experiments, we find that most misbehaving applications are used at least once by the user in the past two weeks. In this case, the current adaptive battery system would only limit a very small percentage of all the misbehaving applications.

Chapter 4

System Design

The goal of our system is to identify misbehaving applications on mobile devices. From a high level, this is achieved by comparing the apps found running on a device against the apps that are actually used by the user. An application would be flagged as misbehaving if it is consuming significant amount of system resources while providing no meaningful service to the end user. In order to detect such applications, there are three majors steps we needs to take. First, the system needs to collect information about a user's usage behaviour. This includes information such as the location and time when each application has been used in the past. After obtaining this information, the next step is to build a usage model based on the data collected previously. The usage model would be viewed as the "correct" state of a device since it represents what the user actually uses. The last step is to actively monitor all the processes running on the device. The system compares the list of current running process with the user's usage model and marks any application operating outside of the model as misbehaving. Overall, the detection process is fairly straight forward on paper, but there are still a few challenges we need to address when implementing it on real devices. The following sections discuss some of the challenges we have faced during the development, and introduce our designs for over coming those challenges.

4.1 Challenges

In order to create a viable solution for the real world, there are three major challenges we need to address.

4.1.1 Minimizing System's Impact on Battery Life

Ideally, our monitor could benefit from having real-time access to data such as system metrics and user's usage records, but in order to do so, the monitor would need to constantly poll records from the under-laying operating system. When running on a mobile device powered by battery, any type of aggressive polling could have significant impact on the device's battery life. One of the benefits users get from identifying misbehaving applications is the potential of reducing battery consumption, but if the monitoring software ends up costing more battery than it could save, the value of using our monitoring system would be greatly diminished.

4.1.2 Uncontrolled Environment

Unlike past solutions[12, 13, 25], our system does not run on controlled environments managed by IT professionals such as data centers. Instead, it runs on mobile devices operated directly by the end users. At anytime, the software may be turned off by the user, or the entire device may shut down due to low battery. On top of that, we also do not control when a device is charged, and whether the device has access to WiFi, both of which we would depend on during the model training process.

4.1.3 Inexact Time of Code Execution

In order to maximize battery life on mobile devices, developers of the Android operating system have placed several restrictions on applications' ability to run background tasks. One of them is removing the ability for applications to schedule periodic tasks with exact time intervals. Previous research has shown that there is a significant upfront energy cost when starting onboard components like WiFi and 3G/4G. It is beneficial to group multiple applications together to run at the same time in order to share this fixed overhead. As for monitoring system like ours, this means we would not be able to spread our code execution evenly throughout the time.

4.2 Design Principles

To address the challenges mentioned above, we have devised the followings design principles which we aim to follow throughout our system:

- **Gracefully Handle Datasets with Missing Data:** Our client-side program is built as a long-lasting Android service, but this does not mean the monitor is safe from unexpected shutdowns. Users always have the choice to turn off services manually, and on some Android distributions, the operating system may also silently turn off background services when resources are scarce. This means our system should be able to work with datasets with gaps of missing data, and whenever a gap appears in the dataset, the system needs to be able to downgrade its detection accuracy gracefully while maintaining most of the monitor's functionalities.

- **Minimize Usage of Expensive I/O Components:** As previous works have shown, I/O components are major contributors to fast battery drain [21]. So it is important that our system minimizes its usage of I/O components such as WiFi and GPS.

- **Able to Work with Various Polling Frequency:** As Google gradually improves the Android Platform API, various limits have been placed on how frequent apps are allowed to run their background tasks. Several major changes have already been made in the past few years, and there is no guaranty what kind of changes Google would implement in the future. In order to maximize the deployability of our system, we should not have any hard requirements on the frequency of our data points.

- **Maintain Loose Coupling Between Client And Server:** There is no guaranty the device would be connected to the Internet throughout the day. Sometimes it is possible the device might be running for multiple days without Internet connection. The client should make no assumptions on when it would be able to connect to the server, and it should remain functional to the best of its ability when there is no Internet connection.

4.3 System Architecture

As shown in Figure 4.1, our monitoring system can be split into the client-side section and the server-side section. The client-side runs on the users' Android devices and is responsible for gathering data, uploading raw usage records, identifying misbehaving applications, and displaying outputs to the users. The server-side runs on a Ubuntu server and is responsible for training models, and storing raw data for later analysis. In the rest of this chapter, a brief description is provided for each of those tasks.

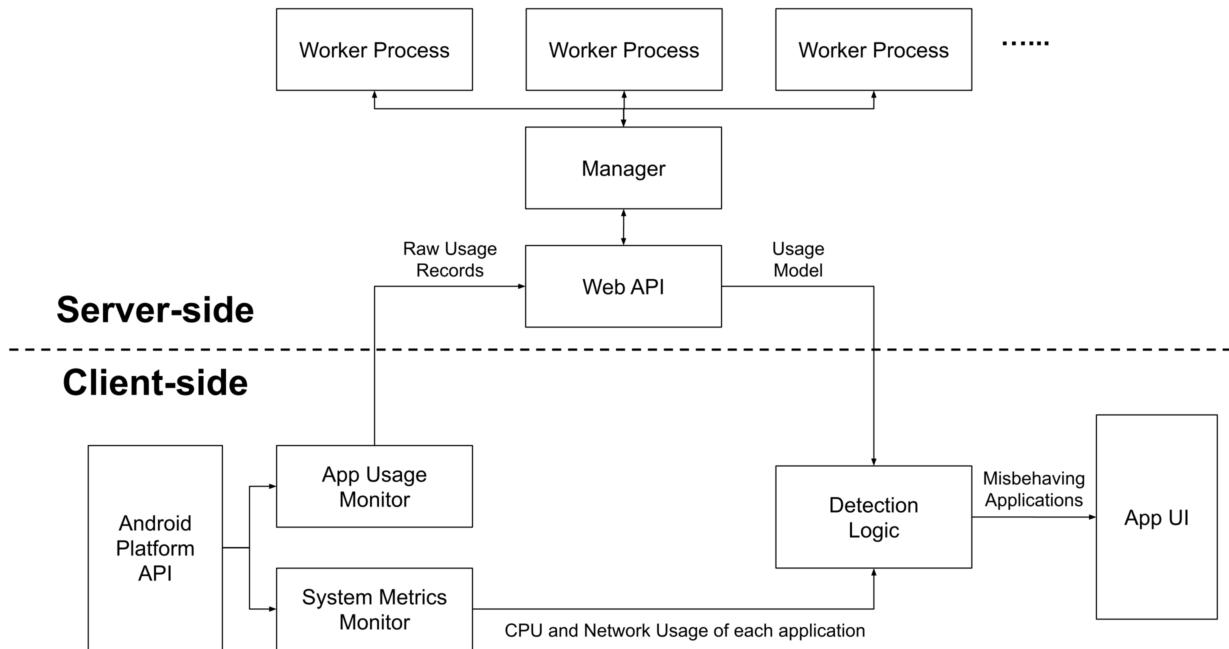


Figure 4.1: System Design Overview

4.3.1 Data Gathering

The data gathering task is jointly handled by two components running on the client-side. The two components are **app usage monitor** and **system metrics monitor**.

The **app usage monitor** is responsible for collecting data that describes the context when a user is likely to use each app. Every time the user turns on the screen, the monitor starts to periodically log the Android Application ID of the app currently in focus along with the GPS location, and the current time. An example of the raw usage record is shown below:

```
time, appID, latitude, longitude
2018/11/8-14:16:45, com.android.chrome, 43.471412, -80.563666
2018/11/8-14:18:13, com.oneplus.deskclock, 43.471412, -80.563666
2018/11/8-14:20:00, com.facebook.katana, 43.471412, -80.563666
2018/11/8-14:20:15, com.android.chrome, 43.471412, -80.563666
2018/11/8-14:20:30, com.facebook.katana, 43.471412, -80.563666
```

In order to minimize our system's energy overhead, special care is taken when programming

```
Load: 0.2 / 0.51 / 0.62
CPU usage from 131338ms to 90589ms ago (2019-05-15 14:55:37.295 to 2019-05-15 14:56:18.044):
  18% 906/system_server: 8.7% user + 9.6% kernel / faults: 4065 minor
  1.8% 1913/com.android.systemui: 1.4% user + 0.4% kernel / faults: 568 minor
  1.6% 532/ueventd: 1.3% user + 0.3% kernel
  1.6% 8405/kworker/u16:9: 0% user + 1.6% kernel
  1.5% 9550/com.android.chrome:sandboxed_process0: 1.2% user + 0.3% kernel / faults: 57 minor
  0.9% 765/surfaceflinger: 0.6% user + 0.3% kernel / faults: 43 minor
  0.7% 96/kcompactd0: 0% user + 0.7% kernel
  0.7% 29081/com.google.android.gms: 0.5% user + 0.1% kernel / faults: 6008 minor
  0.7% 26495/com.google.android.gms.persistent: 0.4% user + 0.2% kernel / faults: 556 minor
```

Figure 4.2: Raw CPUinfo Data Returned by Dumpsys

the usage monitor. First, the usage monitor listens for the "screen on/off" events broadcast by the Android operating system. It would start periodically checking the onfocused app only after it has received the "screen on" event, and stops once the "screen off" event is triggered. This means the usage monitor would not be running when the phone is idle, and this helps to maximize the CPU's sleep time. Second, the monitor heavily utilizes "passive location provider". Instead of actively requesting fresh locations from the GPS hardware, the monitor tries to use the last known location previously requested by other apps running on the same device. The monitor only requests fresh location when the there is no last known location (when the device was recently started) or when the last known location is too old. In the current version, the monitor is programmed to request a new location if the previous one was collected more than 30 minutes ago. From our experience, just by relying on the locations requested by first party apps from Google alone, we are able to maintain relatively fresh GPS locations, and the monitor rarely needs to request locations by itself.

The **system metrics monitor** is responsible for finding out what applications are actively running on a device. In this work, we use CPU usage and network usage as the two main metrics for identifying active applications. Both of those metrics are collected through the *dumpsys* tool provided by the Android platform. The CPU usage is found under the *cpuinfo* section of the *dumpsys*'s outputs, and the network usage is found under the *netstats* section. Examples of the raw data are shown in Figure 4.2 and Figure 4.3.

From Figure 4.2 we can see the CPU usage is fairly straight forward to interpret. Each process has a percentage along with an application ID. We can simply group each line by the application ID, and sum up the percentage values to get the total CPU usage for each application. For the network usage, the raw records are a little more complicated. The operating system identifies each application by its Unix UID (user identifier), and aggregates each application's network usage into two hour windows. For example, if an

```
ident=[{type=MOBILE, subType=COMBINED, subscriberId=302220..., metered=true, defaultNetwork=true}] uid=10266 set=FOREGROUND tag=0x600
  NetworkStatsHistory: bucketDuration=7200
    st=1557432000 rb=11727 rp=15 tb=1986 tp=13 op=0
    st=1557453600 rb=51331 rp=48 tb=4920 tp=36 op=0
    st=1557496800 rb=3705 rp=9 tb=1587 tp=11 op=0
    st=1557532800 rb=37224 rp=40 tb=5265 tp=38 op=0
    st=1557540000 rb=15736 rp=17 tb=2175 tp=15 op=0
    st=1557612000 rb=19095 rp=20 tb=1975 tp=17 op=0
    st=1557784800 rb=14720 rp=24 tb=3930 tp=24 op=0
    st=1557856800 rb=12835 rp=16 tb=2171 tp=19 op=0
    st=1557864000 rb=14460 rp=24 tb=3793 tp=28 op=0
    st=1557871200 rb=2069 rp=6 tb=932 tp=8 op=0
    st=1557878400 rb=1686 rp=4 tb=759 tp=5 op=0
ident=[{type=MOBILE, subType=COMBINED, subscriberId=302220..., metered=true, defaultNetwork=true}] uid=10272 set=DEFAULT tag=0x20003300
  NetworkStatsHistory: bucketDuration=7200
    st=1557669600 rb=7269 rp=24 tb=7215 tp=22 op=0
    st=1557849600 rb=9034 rp=32 tb=10649 tp=30 op=0
    st=1557864000 rb=12076 rp=38 tb=11283 tp=37 op=0
```

Figure 4.3: Raw Netstats Data Returned by Dumpsys

application has accessed the network between 4pm and 6pm on May 14 2019, the usage record would show up under the row with the timestamp *"st=1557864000"*. Here the value *"1557864000"* represents the epoch time for 4pm May 14 2019. The value for "rb", "rp", "tb", and "tp" represent "bytes received", "packet received", "bytes sent" and "packet sent". The operating system only shows the *netstats* data aggregated using two hour windows, if we want to poll the network usage at another frequency we would need to take two snapshots at different time and calculate the difference on our own.

Unlike the app usage monitor that runs only when the user is actively using the device, the system metrics monitor has to collect data regardless of users' activities in order to find misbehaving applications. This creates additional challenges in minimizing the system's energy overhead. If the metrics monitor wakes up the device too often, the monitor itself might start turning into a misbehaving application. To prevent this from happening, we decide to give up the ability of setting exact execution time for our periodic tasks. By doing so, we can provide the operating system with more flexibility when scheduling background processes. The API we call is the *"setInexactRepeating"* method found under Android's AlarmManager. With this API we can still request a general time interval, but the exact time of code execution would be decided by the operating system. With inexact repetition, the operating system has the ability to combine as many periodic tasks as possible to run one after another. This allows multiple background tasks to share the cost of waking up the CPU, and reduces the overall battery consumption.

4.3.2 Data Uploading and Model Creation

Moving model creation to the server-side is one of the most important decisions we have made early on in the development process. By delegating this responsibility to the server,

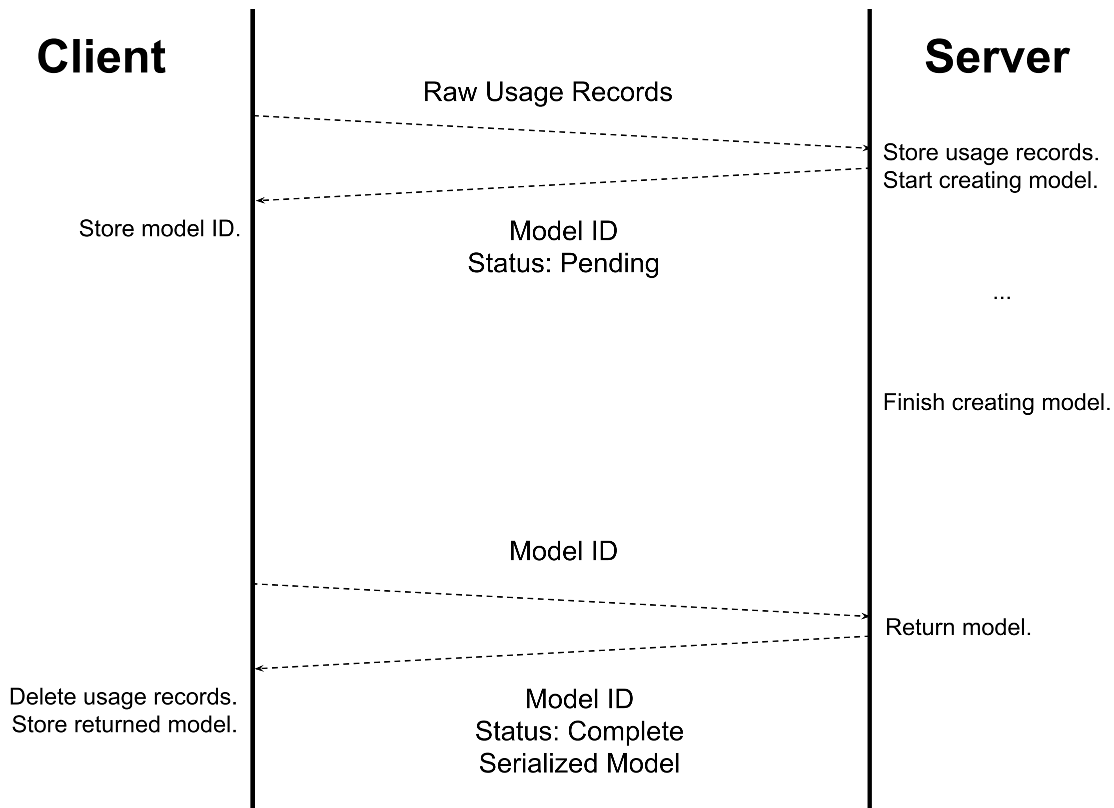


Figure 4.4: Client-server Interaction: Best-case Scenario

our model creation is no longer limited by the client-side hardware. This also gives us the flexibility to try different model parameters without having to update the client program on users' devices. Unfortunately, there are some drawbacks associated with this design. Our system now requires an Internet connection in order to generate and update its models. Along with this dependency are various potential issues which might occur during network calls. All of those network issues would need to be handled gracefully. Also depending on the size of the dataset, the the server may take more than a couple seconds to create all the models. This means, due to the one minute timeout limit commonly used by proxy servers along the network, we could no longer reliably finish the data uploading and model creation all within the same network request. To solve this, the server has to take an asynchronous approach which adds further complexity to the interaction between our clients and the server.

The code for uploading usage data shares the same periodic task with the system metrics

monitor. By doing so, we eliminate the need of creating another repeating background process. Every time when the metrics monitor finishes collecting system metrics, it would check to see if there are any raw usage records from the previous days. If there are existing records, and the device is also currently connected to WiFi and battery, the client would then precede to upload those records to the server. Figure 4.4 illustrates the best-case scenario of this interaction between the client and the server. After receiving the raw usage records, the server creates an entry for the new model inside its SQL database, and marks the model status as "pending". It then passes the usage records to the worker manager, and delegates the model creation to a separate worker process. A more detailed discussion on how we create our model can be found in Chapter 5. Once the job has been successfully scheduled, the server terminates the request by returning the model ID along with a "pending" status to the client. The client takes the model ID, and stores it on the device. The next time when the periodic task is triggered, the client retrieves the completed model by sending the model ID back to the server. After successfully receiving the models, the client deletes any usage records which have already been uploaded. This concludes the best-case scenario of all possible interactions between a client and the server. When operating in the real world, different problems could arise during this interaction. In general, the client does not delete any usage records unless a valid model has been returned. During server downtime, the client would keep the usage records stored on its local storage, and wait for the server to come back. The Figure 4.5 illustrates how our system would handle some of the potential problems.

4.3.3 Detection and Outputs

Once a client receives its models from the server, it is ready to begin detecting misbehaving applications. Note, the detection is only carried out when the device is running on battery. We do not flag any misbehaving applications when the device is charging. An overview of the detection process is shown in Figure 4.6. First, the client needs to find out all the active applications which are current running or have been running recently on the device. This is done by looking at the CPU and network usage data collected through the system metrics monitor. Next, the client inputs the current contextual data into its models, and gets a list of applications which are likely to be used by the user under the current context. Any active application that is not likely to be used under the current context is a potential misbehaving application. That said, there is still one more check before the client can make the final decision. It is possible an application is recently installed, and the application has never been recorded in the model. The client is programmed to skip any applications current running on foreground. This helps to reduce the chance of a newly

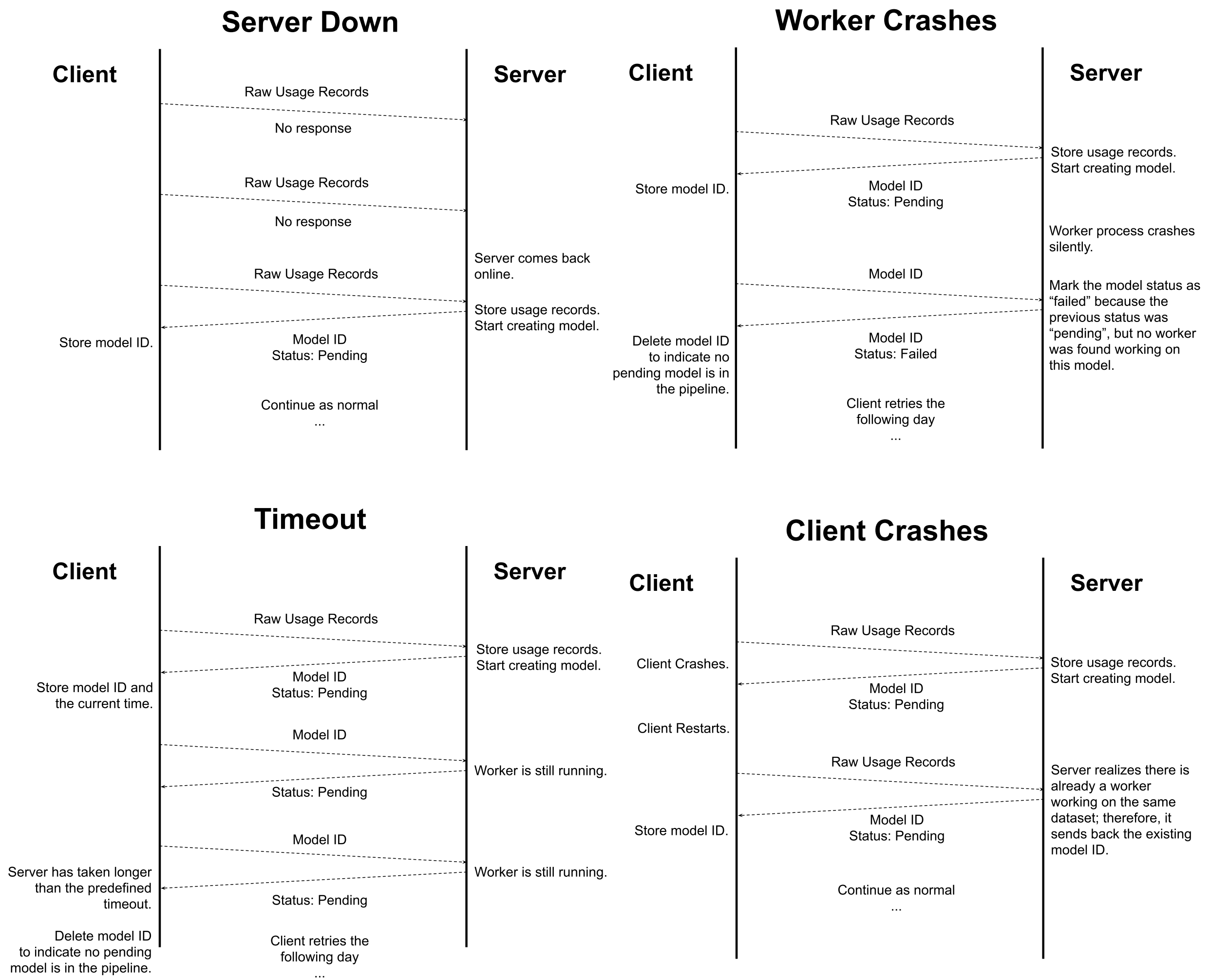


Figure 4.5: Client-server Interaction: Error Cases

installed application being flagged as misbehaving.

After misbehaving applications have been identified, the last step is to show this information to our end users. The screenshots of our client-side app is shown in Figure 4.7. Inside the UI, we list all the misbehaving applications that have been detected by our system. For each application, we include information such as application ID, the first and last time the app is flagged as misbehaving, and the total number of times an app has been flagged. Note, It is worth emphasizing that not all misbehaving apps are equal. Some apps misbehave more frequently than others. By sorting the list in descending order of the "flag count", we are able to surface the most problematic apps to our users. This also allows us to filter out apps that have not misbehaved frequently enough to cause significant problems.

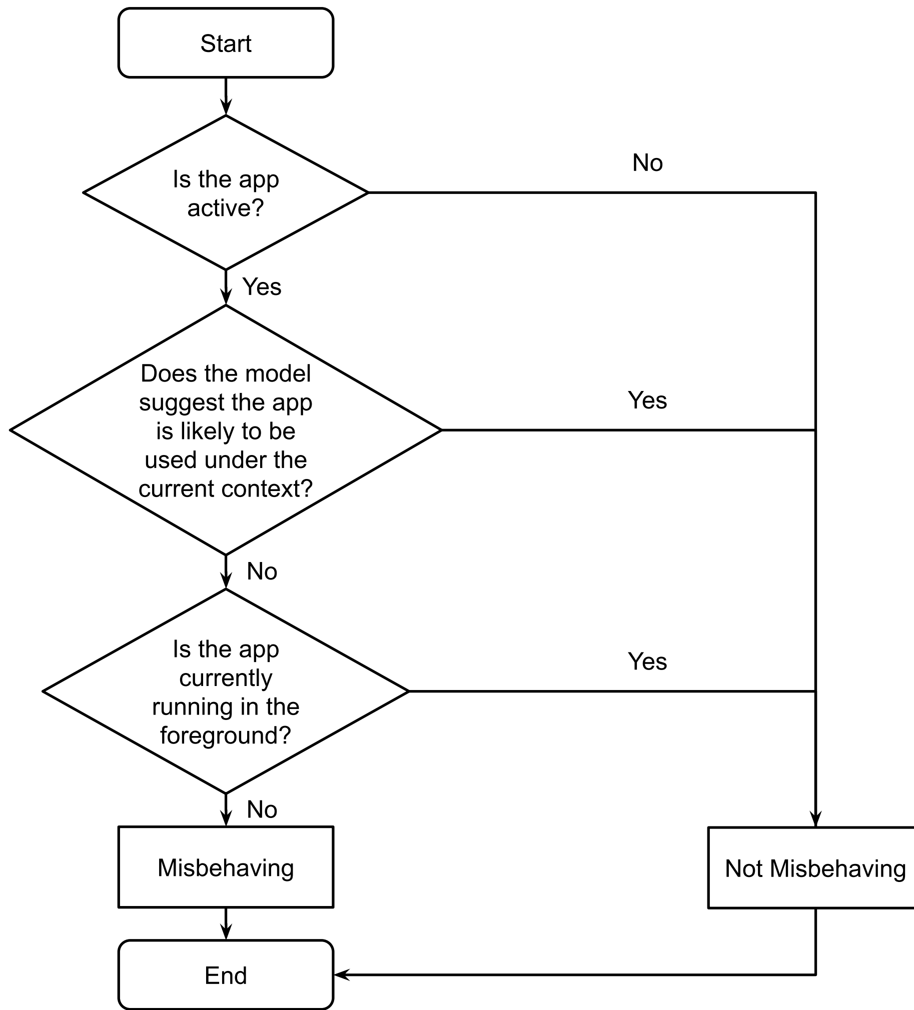
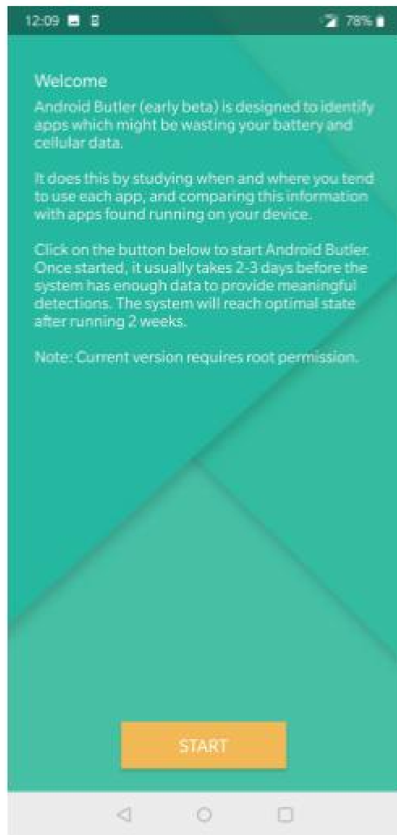


Figure 4.6: Detection Logic Flowchart



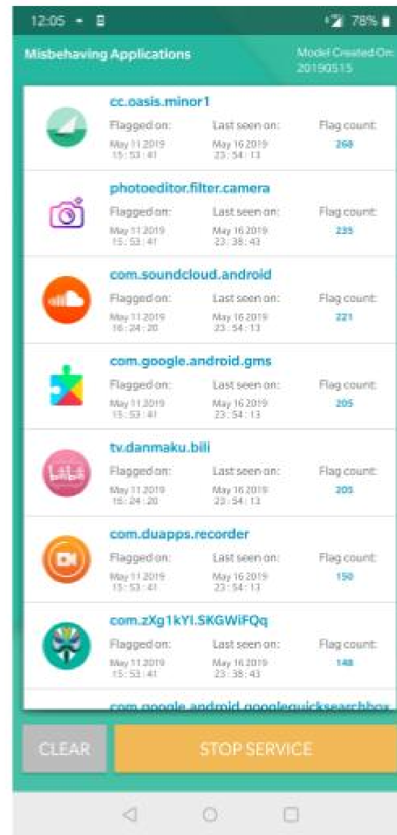


Figure 4.7: Client-side Application UI

Chapter 5

Model Creation

In the previous chapter, we present the system architecture of our monitoring solution. We show how our client collects various types of data, and how it detects misbehaving application using the usage models generated by the server, but till this point, the usage models have largely remained as a set of black boxes. In this chapter, we provide a high level description on how our usage models are built. We first introduce the steps we take to clean and reformat the raw usage data collected by our client. We then explore the data by visualizing them on plots. From those plots we aim to gain a deeper understanding about the underlying patterns. In the end, we experiment with different machine learning algorithms, and select the best one for our use case. The decision is made based on the testing results along with the intuition we have previously gained from the data visualization.

5.1 Data Preprocessing

Data collected from the real world is often messy and incomplete. Once the raw datasets arrive at our server, we first need to clean and reformat them before any models could be trained.

5.1.1 Quantizing Time and GPS Location

As mentioned in section 4.3.1, our usage monitor runs periodically to record the application that are currently onfocus. In order to minimize our system's power consumption, we try to keep the frequency for our periodic polling relatively low (around twice per minute

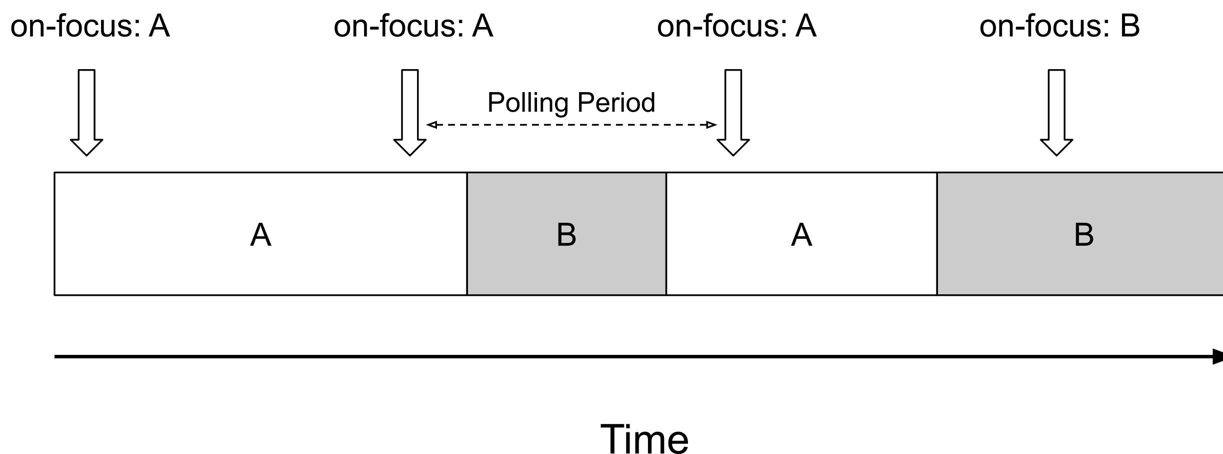


Figure 5.1: Incomplete Usage Record

when the screen is on). This means if the time of an application running in the foreground is less than the polling period, it is possible that our system could miss the application and end up with an incomplete usage record as illustrated by Figure 5.1. To alleviate this problem, we decide to quantize the timestamp into larger time windows. For example, let's say we have a usage record with an application running at 3 different times. Those times are 10:00:30 am, 10:01:00 am and 10:01:30 am. If we decide to quantize the timestamp using a 5 minutes window, we would end up combining all those three records into one single record with the timestamp equals to 10:00:00 am. This new timestamp indicates this particular application is found running between 10:00:00 am and 10:05:00 am. With this larger time window, a record is created regardless how many times the app is spotted during this 5 minutes period. Using this approach, our monitor can now afford to miss a few short usage sessions. As long an application is spotted once in the larger time window, the application would show up in the final record. The downside is that our timestamps would be less precious, but this is not a problem for our use case since we are modeling real humans and not machines that follow the clock perfectly.

Beside the timestamp of each usage record, another important component of our raw dataset is the GPS location. One common issue when dealing with GPS is a problem called GPS drift. This happens when the deceive is stationary, but the GPS location shows the device has moved due to fluctuation in the measurements. As mentioned in section 4.3.1, our location data is collected through Android's passive GPS provider. This means, for most of the time, there is no GPS drift issue since our client are reading fixed value cached previously by other applications. The problem arrives when the device is

stationary, but the cached GPS value is updated by another application. This results in a sudden change in the measurement, and it could be misinterpreted by our model as meaningful information when in reality it is just random noise. To address this problem, we take a similar approach as the quantization we have done on the timestamps. Instead of keeping all the digits returned by the GPS component, we would slightly reduce the precision in our longitude and latitude measurements. This reduces the resolution of our data, but it helps to filter out noise caused by GPS drift.

5.1.2 Restructuring Usage Data

The goal for our models is to answer the question "given a set of contextual data, is an application likely to be used by the user under this context". One way to achieve this is by structuring the question as a classification problem. For each set of contextual data, an application could be labeled either as "used" or "not used". Our models then try to encapsulate the context that have been associated with each label in the past, and make predictions about the label based on the current contextual data.

Before any model could be built, we first need to restructure our datasets for training. An overview of this process is shown in Figure 5.2. For each application, a new dataset is generated. Inside those datasets, each row represents a time period of 5 minutes (in the actual system, a period of 30 minutes is used). Beside the timestamp, we also have the GPS location of the device and a label indicating whether the application is used during this time period. Each dataset is later used for training models specifically for the application it is associated with.

5.1.3 Normalizing Numerical Features and Balancing Dataset

Measurements such as longitude and latitude have very different ranges of values comparing to time-of-day; therefore, it is important for us to normalize them into a similar range before feeding them to the machine learning algorithms. For GPS data, we choose to use normalization by Z-score. This is because normalization by Z-score is less sensitivity to extreme values. For most of the cases, a user's location is concentrated within one city, but "extreme" values in longitude and latitude may occur when the user occasionally travels to another city. In those situations, normalization by Z-score could gracefully handle those extreme values without making the rest of the data points undifferentiable.

For time-of-day, we do not have the problem of extreme values since our data ranges from all possible values between 00:00:00 and 23:59:59. In this case, we could simply use

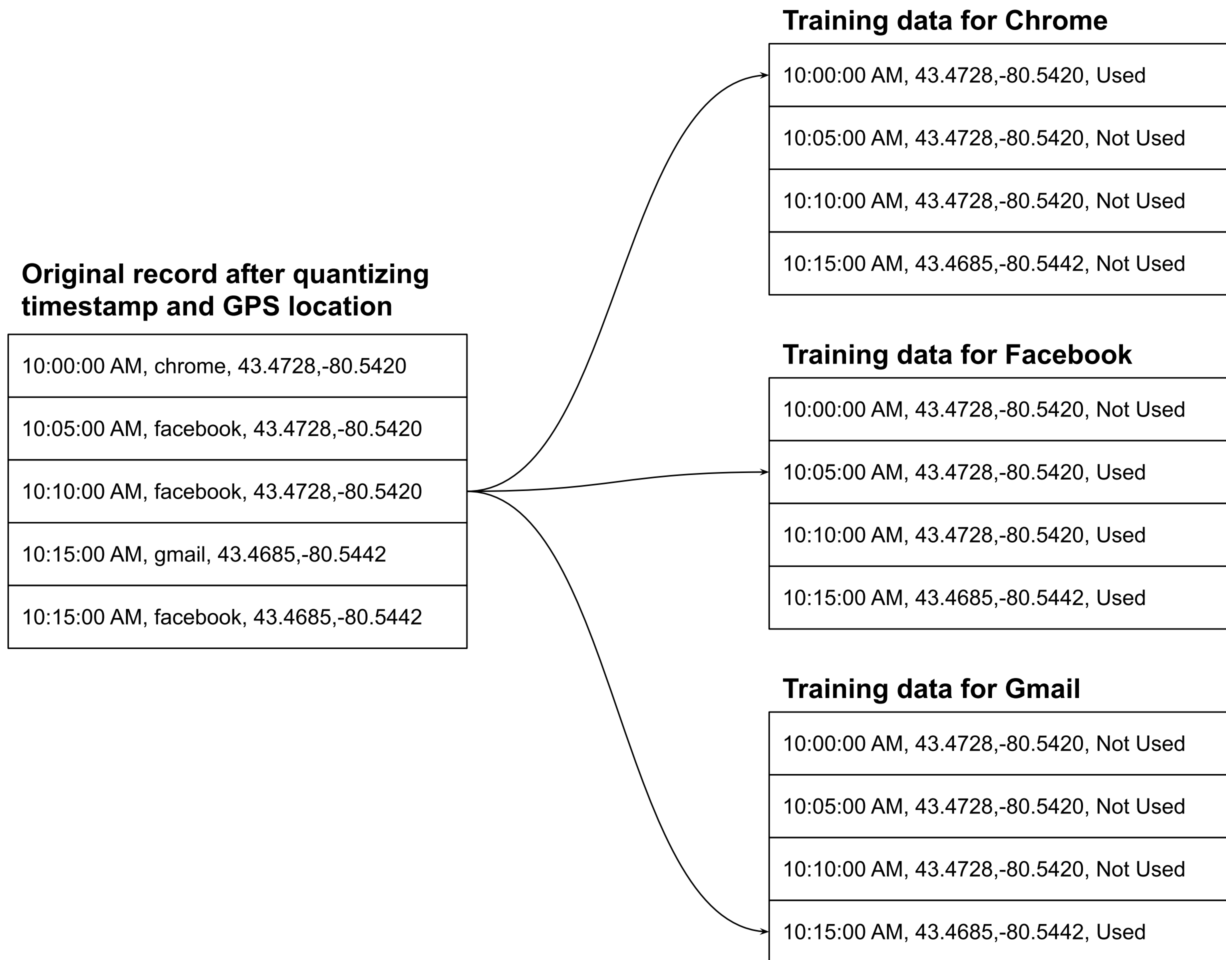


Figure 5.2: Dataset Restructuring

the min-max normalization to map each timestamp to a number between 0 and 1. 0 here represents the time 00:00:00 and 1 represents the time 23:59:59.

Another problem we need to address is the unbalanced dataset. If an application is only used within one particular hour each day, then most of this application's data points would have the label marked as "not used". Models trained on this kind of datasets could achieve a high accuracy by simply guessing the application is never used, and those models are not going to provide us with any useful information on users' behaviours. To prevent this from happening, we need to balance our datasets before training. This is achieved by duplicating the data points with label marked as "used" until there are equal numbers of data points for both labels. By doing so, we are essentially telling the machine learning algorithm to pay more attention to the times when the application is used by the user.

5.2 Data Exploration

For this work, we take the Exploratory Data Analysis (EDA) approach when studying our datasets. This approach requires us not making any assumptions about the underlying model in the beginning; instead, we should first study the data by visualizing them on plots. From those plots we aim to gain a deeper understanding about the underlying patterns, this would later help us to make more educated choices when constructing our models [17]. After restructuring the original usage records, we now have one dataset for each application. Inside each dataset, there are three features and one label. The three features are time-of-day, latitude and longitude. The label is a boolean value that indicates whether the application is used within each time window. We could graph all the data points when an application is used by the user in a three-dimensional graph, and we would end up with a graph similar to Figure 5.3.

Before plotting the graphs, we have originally expected the data points to form spheres around certain points in the three-dimensional feature space, but we soon realize that is not the case. People in general do not walk in circle around a particular point while using their phones; instead, people tend to sit or lay down at a fixed location. This behaviour results in thin cuboids on the three-dimensional plot. You can see the patterns more clearly in the top-down view shown in Figure 5.4 and the side view shown in Figure 5.5.

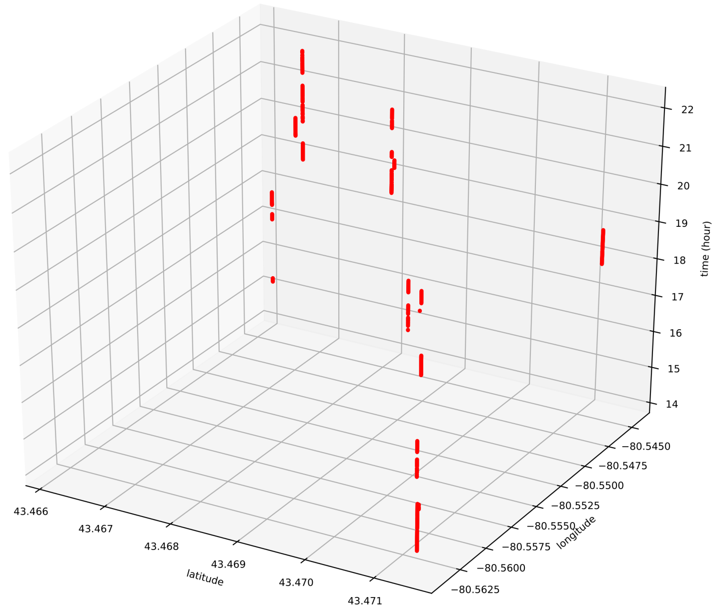


Figure 5.3: App Usage Example

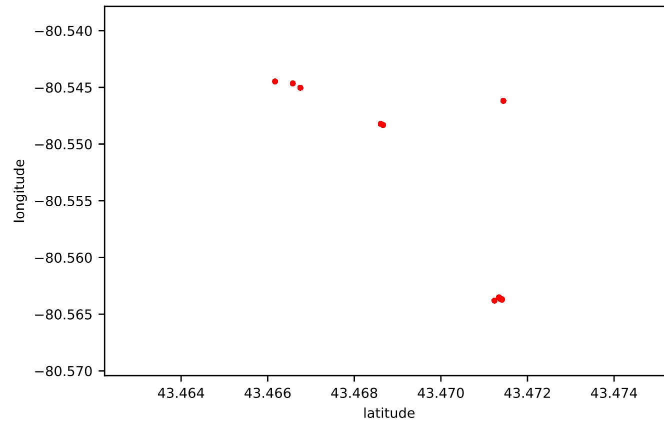


Figure 5.4: App Usage Top-down View

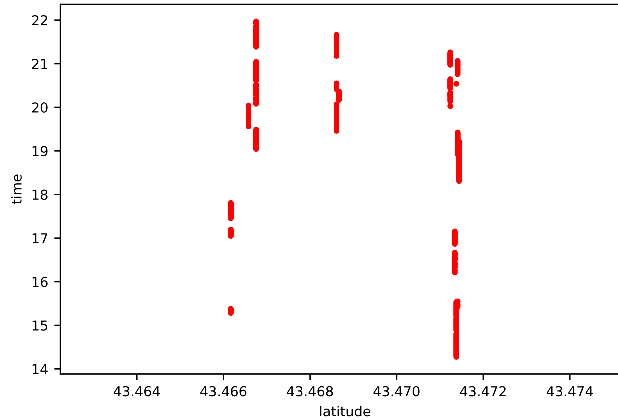


Figure 5.5: App Usage Side View

5.3 Model Selection

In this section, we want to answer two important questions. First, which machine learning algorithm we should use for our use case. The algorithms we have tested include k-nearest neighbors, decision tree, and neural network. Second, we also want to know how much data we need for training our models, but before we can answer those two questions, we first need to talk about the testing environment we use for carrying out our experiment.

5.3.1 Testing Environment

The device we use for our experiment is an Oneplus 6 running on Android version 9.0. The applications installed on the device include the top 10 most downloaded apps from various popular categories on the Google Play store. The categories include social, game, entertainment, photography, shopping, communication, etc. Combining with the pre-installed apps such as Google Chrome and YouTube, there are in total around 150 apps installed on the device during our experiment.

The experiment is divided into two phases. The first phase lasts one month, during this time our client collects usage data and system metrics as described in section 4.3.1, but the server does not create any model during this phase, and no detection is made by the client. The second phase last two months, during the second phase all the system's features are turned on, and client starts to detect misbehaving applications. The data

collected during the first phase is used for exploring different models and their parameters, the results from the first phase are discussed in the current chapter. Once we know how to build our models, the system's overall performance is evaluated during the second phase, the results from the second phase are discussed in chapter 6.

5.3.2 Detection Confidence and Accuracy

Our datasets contain two types of records, the usage records and the system metric records. The usage records describe which applications the user has interacted with during each time window, and the system metric records describe which applications are found running during each time window along with their CPU and network usage. In this work, both usage records and system metrics records are used when evaluating our models. First we train our models based on the usage records. Once we have all the models, we then iterate through each application found in the system metric records, and ask our model the question "is this application likely to be used by the user under the given context?". If the model indicates the application is not likely to be used under the given context, the application would be marked as misbehaving. After iterating through all the system metrics records, we then compare the misbehaving applications our models have identified with the actually usage records. If the application is indeed not used by the user within the next hour then the detection is considered correct. If it is used by the user, then the detection is considered incorrect.

The two main metrics we use to measure our models are detection confidence and accuracy. The two metrics are defined as the followings:

$$Detection\ Confidence = \frac{TP}{TP + FP} \times 100\% \tag{5.1}$$

$$Accuracy = \frac{TP + FP}{TP + FP + TN + FN} \times 100\% \tag{5.2}$$

Here TP, FP, TN, FN are defined as:

- TP: Number of misbehaving applications correctly identified.
- FP: Number of well behaving applications falsely identified as misbehaving.
- TN: Number of well behaving applications correctly identified.

- FN: Number of misbehaving applications falsely identified as well behaving.

Among those two metrics we prioritize detection confidence over accuracy. This is because a monitoring system that is capable of detecting a subset of all existing problems is still valuable to the end users as long as the system has high confidence in its detections. On the other hand, a system that tries to detect all the problems but can not be certain in any of its results is less meaningful to the end users. This is because the users would not be able to carry out any actionable plan with high confidence.

5.3.3 Training, Validation and Testing

In order to evaluate different models, we split the one month data collected during the first phase of the experiment into three subsets. The first 14 days are only used for training, the next 5 days are mostly used for validation, and the last 11 days are mostly used for testing. The training set as the name suggests is for training our models, the validation set is for experimenting with different model parameters, and the testing set is for evaluating the final models. Usually after a model is created, it is evaluated against the entire testing set. In this work, we decide to take a slightly different approach involving "moving windows". Instead of using the last 11 days as one single testing set, we would break them into 11 smaller testing sets each containing 1 day worth of data. For each small testing set, a model is created based on the usage data collected from the previous 14 days. The overall performance of our models and their parameters are evaluated by calculating the average results from all the smaller testing sets. The Figure 5.6 is a graphical illustration of this process, as you can see the same process is also applied when testing against the validation set.

There are two benefits with this approach. One, the moving window approach is what our monitor system uses when running on real devices. By evaluating our models this way, we get a better picture on how the system would operate in the real world. Two, occasionally a model might explain a testing set perfectly just by random chance. With multiple pairs of training and testing sets, we can get a more reliable picture about the true performance of our models.

5.3.4 Parameter Tuning

K-nearest neighbors, decision tree, and neural network are the three machine learning algorithms we have experimented with before constructing our models. Each of those

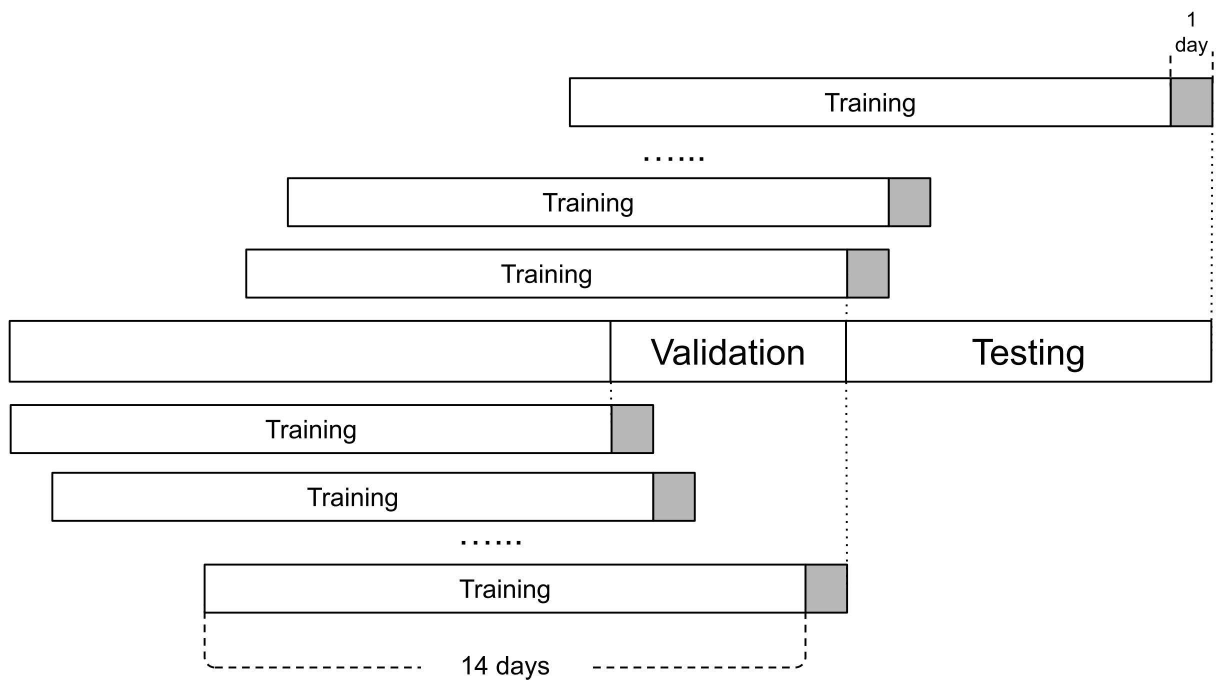


Figure 5.6: Model Evaluation Using Moving Windows

algorithms has a few important parameters need to be configured according to the characteristics of the underlying dataset. By testing on the validation sets, we are able to a find a set of parameters that works for our use case.

For the k-nearest neighbors algorithm, the most important parameter is the "number of neighbors". This parameter decides how many nearest data points the algorithm should take into consideration when predicting the label for a new data point. The value for "number of neighbors" should be set depending on the total number of data points and how densely they are concentrated. When the value is set too low, a prediction's outcome is highly influenced by a few points near the new data point. This could result in inaccurate predictions due to outliers in the dataset. On the other hand, when the value is set too high, the algorithm might end up including points further away which are not related to the point that we want to predict. For our dataset, the density of data points are relatively low. This is due to the quantization we have done on our usage data described in section 5.1.1. After comparing results on the validation set, we find setting the "number of neighbors" to 5 works best for our dataset. That said, the models are currently trained on 14 days of usage data. If the training set is increased in future, we should also consider increasing the "number of neighbors" since there would be more data points in each group.

For the decision tree learning algorithm, the parameter with the most impact is the maximum depth of the tree. With each additional level of depth, the max number of partitions within a feature space doubles. A higher maximum depth allows the algorithm to achieve more fine tuned splits between each class, but it also increases the risk of overfitting. During our testing, we find the accuracy of our model gradually increases as we increase the maximum depth from 1 to 3. The accuracy starts to become inconsistent once the value goes beyond 3.

For the neural network algorithm, we need to decide on the high level structure for the network. This includes the number of layers and the number of neurons inside each layer. The configuration for the input and output layers are predetermined based on the dimension of the dataset, and the type of problem the network is meant to solve. In our case, the input layer contains 3 neurons since we have three features in our dataset, and the output layer contains 1 neuron since we are dealing with binary classification. The real question is how to configure the hidden layers. We start with 1 hidden layer. As we increase the number of neurons inside the layer from 1 to 10, we find the accuracy peaks when the hidden layer has 3 neurons, and the accuracy starts to drop once the number of neurons goes above 3. After we have decided to use 3 neurons for the first layer, we then proceed to add a second hidden layer. For the second layer, we start with 2 neurons. The reason we don't consider 1 neuron this time is because with only 1 neuron in the second hidden layer, the network would end up having a single connection between the second

hidden layer and the output layer. This setup has the same effect as connecting the first hidden layer directly to the output layer rendering the second hidden layer useless. In the end, we find having 2 neurons in the second hidden layer achieves the best result. After two hidden layers, additional layers start to negatively affect the models' accuracy; therefore, the final network is structured with two hidden layers with 3 neurons in the first layer and 2 neurons in the second layer.

5.3.5 Comparison of Results

With all the important parameters determined, we then move on to testing our models using the testing set. The result of each algorithm is shown in Table 5.1. From the table we can see the decision tree algorithm has out performed k-nearest neighbors and neural network in both detection confidence and overall accuracy. This result can be explained by the intuition we have gained from data visualization discussed in section 5.2. If we plot our dataset we would find the points with label equals "used" generally form thin cuboids in the three-dimensional feature space. From this intuition, we can rule out any algorithm that looks for concentrated "bubbles" in the feature space such as the k-nearest neighbors algorithm. Instead we think decision tree would be a good fit for this particular use case. This is because every time when the tree creates a new branch, it would slice the feature space in half. After several levels of branching, our model would end up creating one or more cuboids in the three-dimensional feature space, and those cuboids can comfortably encapsulate the usage patterns we have observed so far from the plots shown in Figure 5.3, 5.4 and 5.5. Note, the highest detection confidence achieved by our model is around 90%. It is possible for a normal behaving applications to be falsely flagged as misbehaving from time to time. As described in section 4.3.3, our UI ranks and filters its outputs based on the number of times an application has been flagged as misbehaving; therefore, unless an application has been constantly flagged by our system, it would not show up on the list of misbehaving applications.

Learning Algorithm	Detection Confidence (mean)	Detection Confidence (SE)	Accuracy (mean)	Accuracy (SE)
K-NN	84.74%	1.67%	72.59%	1.70%
Decision Tree	90.79%	0.45%	74.22%	2.79%
Neural Network	85.62%	1.43%	71.74%	2.84%

Table 5.1: Prediction Results on the Testing Data

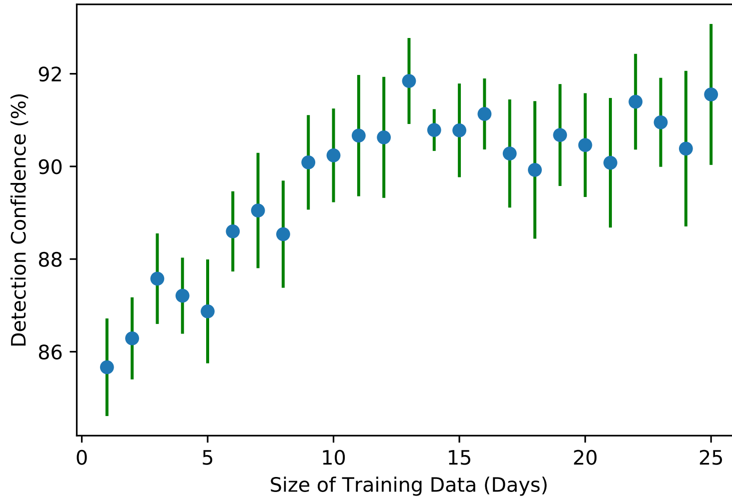


Figure 5.7: Size of Training Data vs. Detection Confidence

5.4 Size of Training Data

Another question we want to answer is "how much data do we need for training our models?". To answer this question we have experimented with various sizes of training data, and the results are shown in Figure 5.7. From the figure we could see the detection confidence of our decision tree models start to plateau after the size of training data reaches around 10 days; therefore, we decide to use 14 days as the size of our training data. The reason we use 14 days instead of 10 days is because 14 days contain two full weeks of data. This way, each day within a week would be represented equally. If we use another size that is not divisible by 7, we could end up with biased models and unfairly emphasize patterns from a few particular days in a week.

Chapter 6

System Evaluation

In this chapter, we evaluate the performance of our overall system. There are three main questions we would like to answer. One, can our models correctly identify misbehaving applications? Two, are the misbehaving applications detected by our system actually harmful to users' experience? Three, how much power does our system consume?

6.1 Model Evaluation

After running the system for two months, our models' detection confidence and overall accuracy are measured and shown in Table 6.1. In this section the performance of our system is compared against a "straw man" system. The "straw man" here represents a hypothetical system that simply marks every running program as misbehaving. From the table we can see our system out performs the "straw man" system on detection confidence, but under performs on overall accuracy. This is expected because for our use case, we prioritize detection confidence over the standard measurement for accuracy. It is more important to have a monitoring system that can detect some problems with high confidence than a system that tries to find all the problem, but not certain in any of its detections. Another reason the "straw man" system is able to achieve an artificially high performance in this case is because in order to study the negative impacts of misbehaving applications on battery life, we have intentionally loaded our device with a large number of potential misbehaving applications. This results in a high percentage of misbehaving applications among all the applications found running on the device. For the average users, the accuracy from the "straw man" system is likely to be much lower.

Model	Detection Confidence (mean)	Detection Confidence (SE)	Accuracy (mean)	Accuracy (SE)
Decision Tree	91.06%	1.07%	75.42%	1.4%
Straw Man	80.74%	0.70%	80.74%	0.70%

Table 6.1: Model Performance

6.2 Measuring the Negative Impact of Misbehaving Applications on Battery

Battery life is one of the most important resources for mobile devices. In this section, we study the negative impacts misbehaving applications could have on an device's battery life. In order to do so, we join the detection records, usage records and system metrics previously collected by our client into a single dataset. The resulting dataset provides us with an overview of the device's states at various times. Each row inside the dataset represents a 15 minutes time window. Inside each window, there are information including usage records, device's charging status, the amount of battery consumed in the current window measured in milliamp hour (mAh) and the misbehaving applications identified by our system along with their CPU and network usage.

Before we can start measuring the impacts from misbehaving applications, there are two things we need to do. First, we need to filter out the time windows when the device is connected to a power source. This leaves us with records collected when the device is only powered by battery. Second, we need to filter out all the time windows when the device is actively used by the user. This is because unlike background processes which slowly drain a device's battery over a long period of time, active foreground applications tend to consume large amount of battery during short bursts, and the consumption often varies greatly depending on many different factors which can not be easily observed. This is demonstrated in Figure 6.1 and Figure 6.2. As we can see, the speed of battery drain varies greatly when the number of usage records is greater than 0. If we leave the time windows with active usage records in the dataset, we would end up with a large amount of noise which can not be explained by the data we currently have; therefore, it is necessary to drop those data points if our goal is to measure the battery drain purely from background processes.

As described in the section 4.3.3, our monitor takes both CPU usage and network usage into account when detecting misbehaving applications. An application is flagged as misbehaving if the application is found using either CPU or network while the model thinks

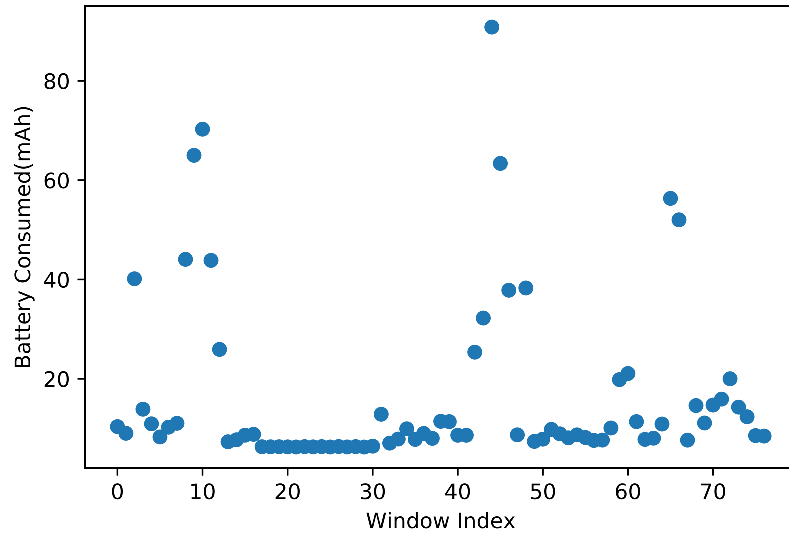


Figure 6.1: Battery Consumption Throughout One Day

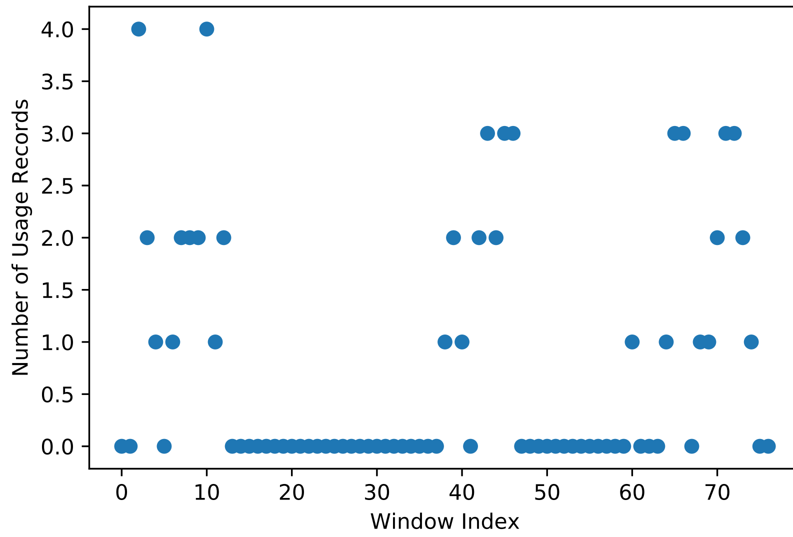


Figure 6.2: Usage Record Counts Throughout One Day

it should not be running. In order to study the impact of those misbehaving applications have on the battery life, we first group the time windows based on the number of misbehaving applications they have. We then calculate the average speed of battery drain for each group. The results are shown in Figure 6.3 where the X is the number of misbehaving applications, and Y is the average speed of battery drain. From this graph we can see, the speed of battery drain gradually increases as the number of misbehaving applications goes up. Eventually when the number of misbehaving applications reaches 10, the speed of battery drain plateaus around 50 mAh per hour. That said,there is one data point reaches above 60 mAh per hour when the number of misbehaving application is at 15. This is likely caused by a few extreme values in the dataset. It is very rare for the device to have 15 or 16 misbehaving applications running at the same time; therefore, the last few data points on the graph could be easily influenced by outliers which is also why they have longer error bars. The plateauing of battery consumption could be explained by resource sharing. For example, when an application requests to transmit a packet over the Internet, there is an battery overhead for starting the WiFi module associated with the request. If another application wants to send a packet 10 minutes after the previous request, the WiFi module would need to be turned on once against, and this would also create additional battery overhead, but as the number of applications that use WiFi increases, eventually the requests would come in frequently enough, and the WiFi module would be powered on the entire time. At that point, the battery consumption would start to plateau which is what we are seeing in Figure 6.3.

Another interesting observation we have found is that most of the battery drain on mobile device is caused by network usage, and not CPU usage. Every time our monitor flags a misbehaving application, it also records the CPU and network usage of the application for future analysis. With this data, we can separate the applications flagged due to CPU usage from the ones flagged due to network usage. If we plot their impacts on battery separately we get Figure 6.4 and Figure 6.5. From those two figures we can see when flagging application based on only CPU usage, the number of misbehaving applications has very little effect on the speed of battery drain. On the other hand, the misbehaving applications flagged due to network usage have a much more visible effect on the speed of battery drain. This observation verifies the finding from previous research done by Pathak et al. In their research, they have shown that a majority of a mobile device's battery is spent on I/O components such as WiFi and 3G which is also what we have observed in our data [21].

Since we have shown that most of the battery drain is caused by network usage, from this point on we will mainly focus on the misbehaving applications flagged based on their network usage. In Figure 6.5, we can see the speed of battery drain is 32.66 mAh/h

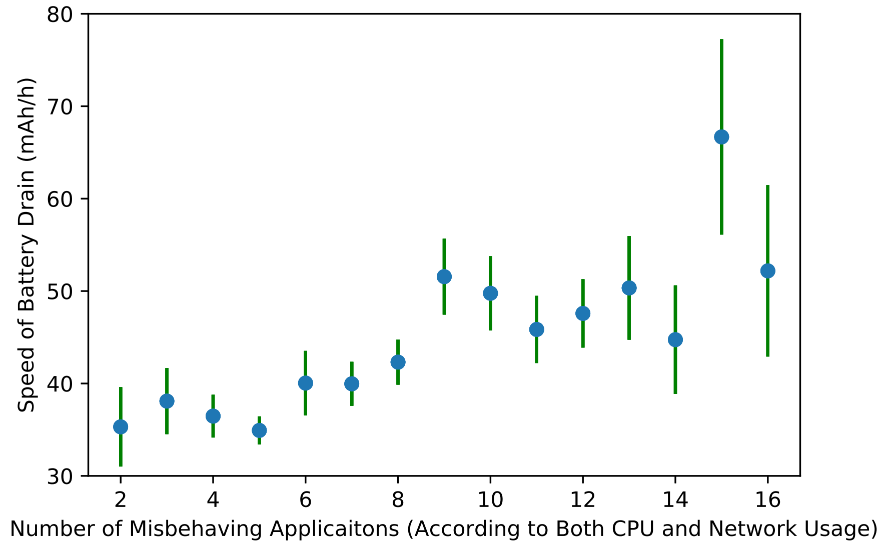


Figure 6.3: Number of Misbehaving Application vs. Speed of Battery Drain

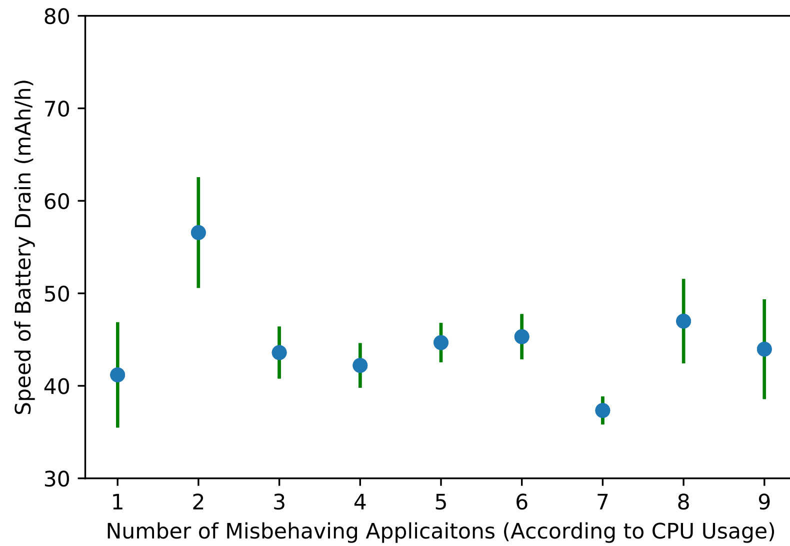


Figure 6.4: Number of Misbehaving Application (CPU) vs. Speed of Battery Drain

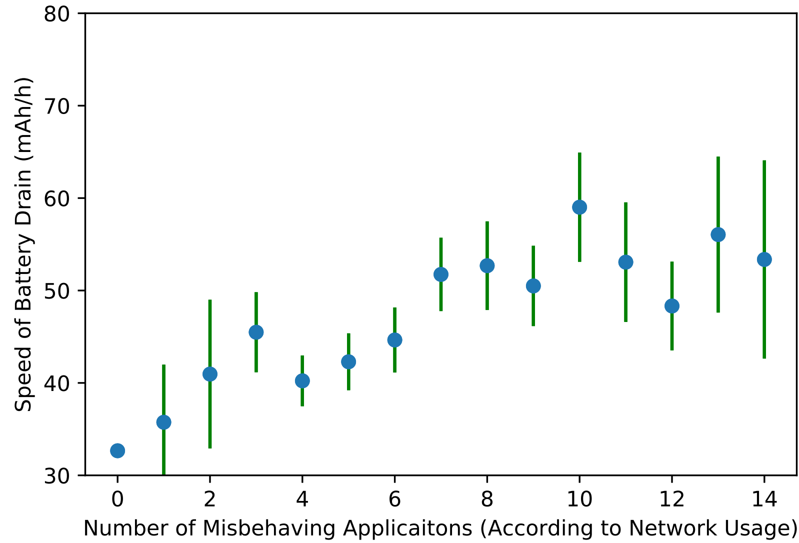


Figure 6.5: Number of Misbehaving Application (Network) vs. Speed of Battery Drain

when there is no misbehaving application. As the number of misbehaving applications increases from 0 to 11, the speed of battery drain increases by 2.01 mAh/h on average for each additional misbehaving application found running on the device. The speed of battery drain plateaus around 50 mAh/h when the number of misbehaving application goes beyond 11. This happens when the device is heavily infested with misbehaving applications, but for average users their devices will likely to have less misbehaving application running than these extreme cases. The device we use in our experiment has a 3000 mAh battery. From the information we have so far we can estimate the percentage of battery each consistently misbehaving application consumes during a period of 24 hours, and we can also find out the impact of those misbehaving applications have on the device's total stand-by time. The results are shown in Table 6.2. From the table we can see, each consistently misbehaving application consumes 1.61% of battery each day and on average reduces the device's stand-by time by 2 hours.

Number of Consistently Misbehaving Applications	Additional Percentage of Battery Consumed Each Day	Stand-by Time
0	0%	91.86h
1	1.61%	89.13h
2	3.22%	86.55h
3	4.82%	84.13h
4	6.43%	89.13h
5	8.04%	81.83h
6	9.65%	79.66h
7	11.26%	77.59h
8	12.86%	75.64h
9	14.47%	73.78h
10	16.08%	72.01h
11	17.69%	68.71h

Table 6.2: Battery Consumption of Consistently Misbehaving Applications (According to Network Usage)

6.3 A Closer Look into Some of The Misbehaving Applications

Within our dataset, we have collected network usage from each application through out the entire experiment. This data allows us to take a closer look into some of the misbehaving applications found on the device. In Table 6.3, we have the top 15 misbehaving applications ranked according to their network usage during one month period. The network usage shown in the table are measured while the phone is idle and running on battery. By comparing each application's network usage against the features it provides, we have come up with the following likely causes of network usage:

- ***Ads Pre-fetching:*** Pre-fetching contents for advertisements is a technique commonly used by app developers. By periodically downloading the newest ads, developers can minimize the load time of each in-app advertisement while keeping the content relatively fresh. Apps likely to have this feature are Facebook, Bilibili, WeChat, Weibo, Selfie Camera and SoundCloud. All of those apps have received more data than others. They all have in-app advertisements, and many of those

APP ID	Data Received	Data Sent	Receive Count	Send Count	Description
com.facebook.katana	24.4 MB	0 MB	136	0	Facebook
tv.danmaku.bili	11.0 MB	0 MB	464	2	Bilibili (video site)
com.tencent.mm	10.5 MB	0.6 MB	446	30	WeChat
com.mcdonalds.superapp	7.5 MB	0.1 MB	44	2	Mcdonalds' rewards app
com.weico.international	4.6 MB	0.1 MB	94	2	Weibo
photoeditor.filter.camera	3.8 MB	0.1 MB	242	4	Selfie camera
com.duapps.recorder	2.6 MB	0.9 MB	166	118	Screen recorder
com.soundcloud.android	2.4 MB	0.1 MB	170	4	SoundCloud
com.facebook.orca	2.3 MB	0.1 MB	98	1	Facebook Messenger
net.oneplus.weather	1.8 MB	0 MB	126	0	Weather app
com.epicgames.portal	0.4 MB	0.1 MB	36	24	Fortnite (game)
com.whatsapp	0.3 MB	0.1 MB	66	22	Whatsapp
com.xiaoji.emulator	0.2 MB	0.3 MB	180	76	Game console emulator
com.netflix.mediaclient	0.2 MB	0 MB	26	0	Netflix
com.tencent.ig	0.1 MB	0.1 MB	16	12	PUBG (game)

Table 6.3: Top Misbehaving Apps Ranked by Data Received

advertisements are able to be displayed on the application's splash screen without any obvious loading time.

- ***Data Harvesting:*** Data harvesting is also a common practice found in many mobile applications. One interesting observation is that contrary to popular belief, social network apps such as Facebook and Weibo have rarely been spotted uploading data while running in the background. Note, the network usage shown in Table 6.3 only contains the network usage collected while the phone is running on battery, but we find this behaviour also holds true when the device is connected to power. This observation seems odd at first glance since both companies' business models are built on targeted advertising which requires large amounts of user data, but then we realize there is probably no need for social networks like Facebook to actively harvest data from users' devices. Just by having all the network requests logged on the server-side, Facebook already has plenty data about a user's interests and behaviours; therefore, it might actually be beneficial for Facebook to lower its battery footprint so the user can stay in their app longer. On the other hand, the apps which have been found uploading data in the background are mainly games and utility tools such as Fortnite, PUBG and the DU Screen Recorder. This is probably because unlike social network

apps, games and utility tools have fewer channels for them to gather information about a user; therefore, it is in their best interests to collect as much information as possible from the user's physical device.

- ***Notifications:*** Push notifications can also cause high network usage. Those notifications include messages from messaging apps such as WeChat, Facebook Messenger and Whatsapp. They also include promotional offers from apps such as the Mcdonalds's reward app and the weather notifications from the weather app.

Another observation from Table 6.3 is that the total amount of network data consumed by background processes is relatively low. Originally we have suspected that misbehaving applications might have significant impact on uses' monthly data quota, but we have found this is not the case. There might be frequent network requests initiated by the misbehaving applications, but the size of each request is often very small, and around 80% of those data is transmitted on WiFi and does not consume data quota. Instead, the negative impacts from those network requests are mostly observed on the battery life.

6.4 Our System's Battery Overhead

One of our design goals is to keep the monitoring system's battery overhead at minimum. To achieve this, a lot of attention has been spent on finding the most optimal ways of collecting usage data and system metrics. In order to measure our system's battery overhead, we have collected the battery statistics returned by the Android's *dumpsys* command, and the outputs from the *dumpsys* command are analyzed using Google's Battery Historian. The results are shown in Table 6.4. As we can see, our app only consumes around 0.38% of the total battery each day.

CPU user time per day	1m 16s 935ms
CPU system time per day	0m 20s 490ms
Total CPU time per day	1m 37s 425ms
Power use per day	0.38%

Table 6.4: Client-side's Daily Battery Overhead

Chapter 7

Conclusion and Future Works

In this work, we propose an automated monitoring system that can detect misbehaving applications running on mobile devices. Unlike previous automated monitoring systems, our solution requires minimum inputs from the end user. Instead, it collects the user's usage records in the background and build models to encapsulate the contexts when the user is likely to use each application. Using the models it creates, our system can identify misbehaving applications that are consuming system resources while providing no meaningful benefit to the end user.

During our development, we find there are several challenges when it comes to creating monitoring systems for mobile devices. If not handled with care, the monitor could either provide us with incomplete information making it difficult to later build models, or the monitor might require a large battery overhead making it hard to justify for the its benefits. In this work, we introduce a practical design for addressing those problems. Our monitor can collect detailed information for generating models while following all the best practices of energy saving recommended by the Android operating system.

For usage modeling, we first study our dataset through data visualization. From data visualization, we manage to gain intuitions about the underlying usage patterns. We then experiment with several machine learning algorithms, and measure their performance for encapsulating the underlying patterns in our data. By combining intuitions and testing results, we arrive at the conclusion that the decision tree learning algorithm is a good fit for our usage case.

In order to evaluate our solution, the monitoring system is tested for a period of two months. From the data collected during the two months period, we find there is a strong positive relationship between the number of misbehaving applications identified by our

system and the speed of battery drain. We also find the misbehaving applications with high network usage are the main causes of fast battery drain, and the high network usage is most likely due to aggressive ads pre-fetching, data harvesting and push notifications.

7.1 Future Works

One potential improvement is incorporating additional features beside time and location in the usage models. For example, we can collect data from a device's accelerometer and gyroscope. By combining those data, we could extract more meaningful higher level features such as "whether the user is at home" or "whether the user is currently traveling on a vehicle". Using those higher level features we would be able to model more complicated contexts and identify usage patterns which are not observable otherwise.

Currently our system only flags misbehaving applications in its UI, and it is up to the user to uninstall them. Another direction for future works is to experiment with actively limiting misbehaving applications' resource consumption. This can be done by terminating misbehaving applications gracefully or putting them into a low priority state with less access to system resources such as CPU and network modules.

Addressing the privacy concerns could also be a future improvement. Right now our client-side application relies on the external server to train its usage models. The current setup gives us the flexibility to experiment with different model parameters without having to change any client-side code, but once we have found the optimal parameter values for our models, we could start delegating the training process to the client-side. The training could be done in the background when the device is charging, and this would solve most of the privacy concerns since all sensitive data would be kept locally on the device.

7.2 Broader Implication

In recent years, more and more software developers have shifted their monetization methods from direct user payments to advertising. This shift has allowed developers to create products which were not financially viable in the past. From the users' perspective, people are willing to sit through the advertisements as long as they feel the software is providing enough value to justify for its cost. If a software is not providing enough value or being too aggressive in showing advertisements, the user can always make the decision to uninstall the software. This arrangement between developers and users has been working relatively

well so far, but there is one problem with this business model. That is there are other costs beside the time spent on viewing advertisements, and those costs are often hidden from the user. For example, as we have shown in this work, there is a significant cost in battery life associated with software using ad pre-fetching. Since those costs are hidden from the user, it can be very difficult for users to properly evaluate the costs and benefits of each software. By building automated monitoring solutions, we can shine a light on those hidden costs, and we believe solutions like our can help to create a more transparent and healthier marketplace for mobile applications.

References

[1] Android documentation: Guide to background processing. http://web.archive.org/web/20190506011631/https://developer.android.com/guide/background. Accessed: 2019-05-06.

[2] Android source code for aggregating an application's battery consumption. https://web.archive.org/web/20190705000235/https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/com/android/internal/os/BatterySipper.java. Accessed: 2019-07-04.

[3] Android source code for estimating power consumption due to wake lock. https://web.archive.org/web/20190705001227/https://github.com/aosp-mirror/platform_frameworks_base/blob/d59921149bb5948ffbcb9a9e832e9ac1538e05a0/core/java/com/android/internal/os/WakelockPowerCalculator.java. Accessed: 2019-07-04.

[4] App annie 2017 retrospective report. https://www.appannie.com/en/insights/market-data/app-annie-2017-retrospective/. Accessed: 2019-05-11.

[5] Google play store: Launch checklist. https://web.archive.org/web/20190406010226/https://developer.android.com/distribute/best-practices/launch/launch-checklist. Accessed: 2019-04-06.

[6] Number of android applications. https://web.archive.org/web/20170210051327/https://www.appbrain.com/stats/number-of-android-apps. Accessed: 2019-05-02.

[7] John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

[8] Thomas M Cover, Peter E Hart, et al. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[9] G Deepak and BS Pradeep. Challenging issues and limitations of mobile computing. *International Journal of Computer Technology & Applications*, 3(1):177–181, 2012.

[10] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 259–268. IEEE, 2006.

[11] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[12] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[13] Miao Jiang. Modeling management metrics for monitoring software systems. 2011.

[14] Yann Le Cun, Ofer Matan, Bernhard Boser, John S Denker, Don Henderson, Richard E Howard, Wayne Hubbard, LD Jackel, and Henry S Baird. Handwritten zip code recognition with multilayer networks. In *Proc. 10th International Conference on Pattern Recognition*, volume 2, pages 35–40, 1990.

[15] Xiangyu Li, Xiao Zhang, Kongyang Chen, and Shengzhong Feng. Measurement and analysis of energy consumption on android smartphones. In *2014 4th IEEE International Conference on Information Science and Technology*, pages 242–245. IEEE, 2014.

[16] James L McClelland, David E Rumelhart, PDP Research Group, et al. Parallel distributed processing. *Explorations in the Microstructure of Cognition*, 2:216–271, 1986.

[17] Mary Natrella. Nist/sematech e-handbook of statistical methods. 2010.

[18] Wei Pan, Nadav Aharony, and Alex Pentland. Composite social network for predicting mobile apps installation. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[19] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 275–284. ACM, 2013.

[20] R Parisi, ED Di Claudio, G Lucarelli, and G Orlandi. Car plate recognition by neural networks and image processing. In *ISCAS'98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No. 98CH36187)*, volume 3, pages 195–198. IEEE, 1998.

[21] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[22] Gian Paolo Perrucci, Frank HP Fitzek, and Jörg Widmer. Survey on energy consumption entities on the smartphone platform. In *2011 IEEE 73rd vehicular technology conference (VTC Spring)*, pages 1–6. IEEE, 2011.

[23] Masoomeh Rudafshani and Paul AS Ward. Towards dependable clients: Improving the reliability and availability of the browsers. In *Proceedings of the 9th Middleware Doctoral Symposium of the 13th ACM/IFIP/USENIX International Middleware Conference*, page 6. ACM, 2012.

[24] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[25] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[26] Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in neural information processing systems*, pages 1473–1480, 2006.

[27] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 113–126. ACM, 2012.