# Association Rule Discovery from Collaborative Mobile Data

Jiri Hamberg

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, May 3, 2018

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Laitos — Institution — Department |
|---|---|
| Faculty of Science | Department of Computer Science |

Tekijä — Författare — Author
Jiri Hamberg

Työn nimi — Arbetets titel — Title

Association Rule Discovery from Collaborative Mobile Data

Oppiaine — Läroämne — Subject
Computer Science

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidoantal — Number of pages |
|---|---|---|
| Master's Thesis | May 3, 2018 | 67 |

Tiivistelmä — Referat — Abstract

Sophisticated mobile devices have rapidly become essential tools for various daily activities of billions of people worldwide. Subsequently, the demand for longer battery lives is constantly increasing. The Carat project is advancing the understanding of mobile energy consumption by using collaborative mobile data to estimate and model energy consumption of mobile devices.

This thesis presents a method for estimating mobile application energy consumption from mobile device system settings and context factors using association rules. These settings and factors include CPU usage, device travel distance, battery temperature, battery voltage, screen brightness, used mobile networking technology, network type, WiFi signal strength, and WiFi connection speed. The association rules are mined using Apache Spark cluster-computing framework from collaborative mobile data collected by the Carat project.

Additionally, this thesis presents a prototype of a web based API for discovering these association rules. The web service integrates Apache Spark based analysis engine with a user friendly front-end allowing an aggregated view of the dataset to be accessible without revealing data of individual participants of the Carat project.

This thesis shows that association rules can be used effectively in modelling mobile device energy consumption. Example rules are presented and the performance of the implementation is evaluated experimentally.

ACM Computing Classification System (CCS):
Information systems → Information systems applications → Data mining → Association rules
Information systems → Information systems applications → Mobile information processing systems

Avainsanat — Nyckelord — Keywords
Data Analysis, Data Mining, Mobile Devices

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

# Contents

# 1 Introduction

Sophisticated mobile devices are becoming increasingly more common worldwide. In fact, the International Telecommunication Union estimates that the number of mobile devices with Internet connections was over 3.8 billion in 2016 [1]. With this rapid growth in the number mobile devices, it is increasingly relevant to understand how these devices are used and how the usage patterns affect the energy consumption of these devices.

The Carat project [14, 15, 19, 20, 18] has collected data from over 850,000 mobile devices worldwide since its initiation in 2012. The Carat data is collected from mobile device users that have installed the Carat mobile application on their device. An analysis server collects data samples sent by the Carat mobile applications whenever the user opens the mobile application. Samples are collected by the mobile application whenever the battery level of the device changes. The data samples consist of a list of system settings and context factors, a list of currently running applications and the current level of battery life. The analysis server uses the collaborative measurements to identify energy consumption anomalies from the users' applications as well as to estimate the energy consumption of individual applications.

This thesis work develops an alternative method for estimating how the above mentioned context factors, system settings, and running mobile applications of a mobile device affect its energy consumption. This method mines the Carat dataset for association rules related to a mobile application of the user's choice. The association rules are filtered so that the consequent of the rules will be an energy consumption estimate and the antecedents of the rules will contain a precondition about the values of some context factors and system settings of a mobile device. These association rules can then be interpreted as energy consumption predictions of an application given a set of preconditions about the context factors and system settings of a mobile device.

Association rule discovery, or association analysis, is a well known data mining technique that is commonly used to discover interesting relationships in a dataset without making many assumptions about the structure of the data. Recent advances in the association rule discovery algorithms [8, 13] allow this form of analysis to be performed quickly for relatively large datasets using distributed algorithms. These features make association analysis an appealing choice of methodology for the study of mobile application data.

A secondary goal of this work is to implement a web based prototype API, which could be used to allow third parties, such as mobile application developers, to have access to the energy consumption predictions from the Carat dataset. Peltonen et al. [19] have explored different aspects of allowing third parties to have limited access to the Carat dataset. The implementation described in this thesis work attempts to conform to the constraints which Peltonen et al. describe, in order to protect the privacy of the users of the

Carat mobile application. Using association rule mining effectively hide all the details about individual Carat users, protecting their privacy, while also enabling reasonably detailed aggregated view of the dataset.

Key research questions that this thesis answers are:

1. Is it possible to generate association rules efficiently and scalably from large datasets? The solution should be fast enough to be used in a real-time query API while still having reasonable predictive power?
    1.1. How do state of the art association rule generation techniques scale as the size of the dataset is increased?
    1.2. Can association rules be generated quickly enough to support a real-time energy prediction query API?
2. How to select interesting and useful association rules from the set of all generated rules?
3. How to implement a user friendly web based query engine for discovering the association rules?

This thesis consists of three main parts. Chapters one to five present the related literature and the theory on which this thesis work relies on, and explains the Carat data from which the results are derived. Chapter six presents the implementation of the analysis engine as well as the user interface. Chapters seven and eight present the results and the conclusion of this work.

# 2 Background

Mobile devices are becoming increasingly common [1] all around the world. To understand and characterize the different aspects of mobile device usage, various methods of data collection and analysis have been proposed in the literature. Despite technological advances, battery life remains a factor that severely limits the scope of mobile device usage. The literature proposes methods for both accurately modelling energy consumption and for decreasing the energy consumption of devices. Association analysis is a standard data mining technique that has been widely applied. A couple of recent examples of utilizing association analysis for different data mining tasks are shown here to give a sense of the variety of the problems for which the association analysis is applicable.

## 2.1 Mobile Data Collection and Analysis

Various approaches have been used for collecting and analysing data from mobile devices. Multiple applications have been proposed for collecting detailed logging information from a relatively small number of smart phone users [7, 6, 23]. This kind of logging data can be utilized to find device and application usage patterns, to building statistical models that predict individual user's future behaviour, and to building regression models which accurately predict individual users energy consumption.

Large scale data collection and analysis platforms have also been proposed for studying mobile data. Wagner et al. [24] describe a mobile data collection system that has been used to collect usage information from 12,500 Android mobile devices. The authors discuss challenges regarding privacy, security, transparency, and accountability of their data collection and analysis system.

Oliver et al. [16, 17] have studied diurnal mobile device usage patterns and energy consumption patterns using a dataset collected from over 15,000 BlackBerry mobile devices. The data consists of the devices backlight activity, the operating system's idle counter measurements, battery level and charging activity, device shut down events, and device type and operating system version.

Oliner et al. [14, 15] have used mobile device system settings and context factors collected from over 500,000 devices to detect energy consumption anomalies in mobile applications. The analysis system called Carat collects samples containing list of currently running applications, level of battery life and other context factors and system settings from the users that have installed the Carat mobile application. These samples are sent to an analysis server which is implemented using Spark cluster-computing framework. The applications with anomalous energy consumption are labelled as energy hogs if they consume above average amount of energy on most users' systems. If an application only consumes more than average energy under specific

circumstances, such as on a specific version of an operating system, then the anomaly is labelled as an energy bug. After analysing the samples collected from all users, the analysis server informs the Carat mobile application about discovered energy hogs and bugs. The mobile application can then notify the user about hogs or bugs affecting the user's system and even estimate the amount of battery life that could be saved by shutting off such applications.

To detect an energy hog application, the authors estimate the reference battery drain probability distribution by using data points that do not have the subject application running. They likewise estimate the subject battery drain probability distribution by using only data points that have the subject application running. If the expected value of the subject distribution is notably higher than the reference distribution's expected value, then the application is labelled as an energy hog. An application that is not an energy hog can be detected as an energy bug by estimating the subject application's energy drain probability distribution assuming some context factors such as a specific operating system version. The energy drain distribution is then estimated by using only data points that have the subject application running and meet the criteria of the context factors. If the subject distribution, given the specific context factors, has an expected value that is notably higher than the reference distribution's expected value, then the application can be labelled as an energy bug. The detected difference in the expected values of the probability distributions gives an estimate of how much energy can be saved by not using an energy hog or bug application. An application can have multiple energy bugs with different sets of context factor criteria. The authors detected a total of 233,258 instances of energy bugs and 10,110 energy hogs from a total of 102,421 applications that were present in the collected dataset.

## 2.2 Battery Life of Mobile Devices

As mobile devices become increasingly essential for our every day lives, the demand for longer battery lives is constantly increasing. Despite the impact that mobile device battery life has on every day lives of hundreds of millions of people worldwide, the factors which affect a mobile device's battery life have not been studied extensively.

Mobile networking is an active area of research where mobile device energy consumption has an import role. New mobile networking technologies are being developed constantly and providing faster, more reliable and more energy efficient mobile networking solutions for customers is a profitable business for the internet service providers of the world. Consequently, it is no surprise that funding and research efforts have gravitated towards the field.

As a concrete example of such research, the impact of using 2G and 3G networks for the mobile phone battery life has been compared by Perrucci et

al. [21]. The authors used a Nokia N95 phone to test the relative battery consumption of various tasks comparing the results of using GMS, a 2G mobile networking technology and UMTS, a 3G mobile networking technology. The tasks for which the battery consumption was measured included sending 50 SMS messages of 100 bytes, downloading 100 megabytes of data, and performing a 5 hour voice call.

The conclusion of the study is that different networking technologies are energy efficient in different tasks. While the UMTS network is significantly more energy-efficient for downloading data, the GMS is more energy-efficient when sending SMS messages or performing voice calls. The authors argue that this information could be used to minimize energy consumption of mobile phones when multiple networking technologies are available.

Another perspective from which the mobile device energy consumption has been studied in recent years, is offloading or remote execution of programs. The idea of offloading is simple: since CPU intensive tasks tend to consume a lot of energy, computing tasks can be executed remotely in a dedicated server or cloud environment. However, transferring data to and from the offloading platform also consumes energy and imposes other constraints such as delay, connectivity, availability, security concerns and potential costs of using such a platform.

One approach to decreasing mobile device energy consumption using computation offloading is proposed by Qian and Andresen [22]. In this work, the authors propose a programming model and a runtime environment called Jade for creating processes that are offloading aware. Programs written using their programming model will be subject to custom task scheduling. The scheduler communicates with available computing servers exchanging information about their available computing resources such as the number of CPUs and the amount of RAM to make decisions on where certain tasks should be offloaded to. Upon first execution of each task that uses Jade, the task will be profiled in order to find out if the program is suitable for offloading in future executions. The profiler collects statistics such as runtime, energy consumption and size of the task. Whenever a remotable task, that has been profiled, is executed, the Jade environment performs an optimization step where it tries to figure out the optimal way to execute the remotable task. The optimizer estimates the energy consumption of the task for each available offloading server as well as the mobile device itself and chooses the host that is estimated to be the most energy efficient for the execution of task.

The authors wrote two applications to evaluate their systems performance. The first program performed facial recognition on photos on the phone, chosen by the user. The other program simulated a navigation application by performing Dijkstra's shortest path finding algorithm on a graph. Both applications were run both locally and with offloading enabled. When using offloading, the programs' energy usage was reduced by 34% and by 39%

respectively. The execution time of the programs was also reduced by 37% and 45% respectively.

The usage patterns of mobile devices and applications have also been the subject of various studies and play a role in the energy consumption of the mobile devices. Ferreira et al. [7] have shown that mobile applications are frequently used in short bursts of activity they call micro-usage. This micro-usage is especially prevalent in social applications and applications that provide users with notifications. The authors suggest that the operating system of a mobile device could possibly optimize resource allocation by identifying applications that are often subject to micro-usage.

Falaki et al. [6] have studied and characterized the usage patterns of 255 mobile phone users. The authors have discovered a large diversity in the average mobile phone usage statistics such as the amount of network traffic, total energy consumption, number of installed applications, and the diurnal user activity. The authors suggest that energy consumption modelling can be enhanced by incorporating personalized usage statistics into the models. They also demonstrate this idea by implementing a energy drain prediction model that accurately predicts user's energy consumption based on the user's past usage patterns and recent activity.

Regression models have been used to model energy consumption of Android mobile devices based on context factors and system settings by Shye et al. [23]. The authors show that the CPU usage and screen brightness are the two dominant energy consumption factors in their data which they collect using a custom Android logging application. They also use the data to characterize the users' workloads and develop an application which gradually lowers an Android phone's screen brightness and CPU usage to decrease energy consumption while attempting to keep the changes small enough to be hard for the user to notice. The authors show that the power saving application is able to save up to ten percent of total battery life. For the study, the application was tested by 20 users, 15 of which said they would use the optimizations that the application provides.

Peltonen et al. [20, 18] have described a way to model a mobile device's battery life as a function of various context factors and system settings including type, speed and activity of the network that the device was connected to, screen brightness, CPU usage, battery health, voltage and temperature, and the movement of the device. Using conditional mutual information, the model estimates the impact of these factors and variables on the energy consumption of the mobile device. Using this information, the authors construct a decision tree based recommendation system called Constella. The Constella application provides the user of a mobile device with recommended actions to increase remaining battery life. The system compares the current values of the context factors and system variables of the device with the model described above, discovering which changes are estimated by the model to save the most battery life. An example of such recommendation could be

"Change from mobile to WiFi network. Expected improvement 33m 33s ± 57s".

### 2.3 Association Analysis

Association analysis is a data mining method developed to find common patterns from large databases. The method was famously conceived to find common patterns in shopping cart content databases in order for the supermarkets to optimize the layout of their stores [2]. Since then, the method has been applied to a wide variety of data mining problems.

Karabatak and Cevdet Ince [11] have used association rule generation and a neural network to train an expert system for detecting breast cancer. The input data of the expert system consists of nine variables describing the clump and a cell specimen of the suspected cancer tissue. The authors use the Apriori algorithm [3] to discover association rules between the input variables. The discovered rules are then used for feature extraction and dimension reduction of the input data. Extracted features are then used as the training data for a multilayer perceptron which is used to classify the tumors to either malignant or benign class. After 3-fold cross validation on a database of 699 records, one combination of association rule assisted feature extraction and neural network achieved a correct classification rate of 97.4% whereas a network which used the original nine variables as input only achieved a correct classification rate of 95.2%.

In another study, Karabatak et al. [12] have used association analysis to classify textures. In this study, images of textures are transformed using a wavelet transformation. Each pixel is then mapped to range 0 to 2 based on brightness of the transformed pixels. Items for the association rule generation are then generated using a 3 x 3 sliding window, essentially concatenating the nine pixels in the sliding window. Apriori algorithm [3] is used to generate frequent item sets from the items. The frequent item sets and their related support values are then used to characterize the texture from which the transaction database is generated. In order to classify an unseen texture image, the authors generate the frequent item sets and related support values and use a shortest distance classifier to label the new image to one of the texture classes. Their training and testing data consists of 500 texture images of size 128 x 128. The images are gray scale representations of 10 classes of textures, such as bark, plastic bubbles and brick wall. In the test scenario, the classifier had a success rate of 97%.

# 3 Carat Data

The Carat data consists of samples containing mobile device system settings and context factors, current battery level, the list of currently running mobile applications and a user specific identification token unique to each Carat application installation. The Carat mobile application periodically collects these samples, typically when the device's battery level changes [15]. The application sends all collected samples to the server whenever the user opens it or another sample is taken while the application is open.

Since we are interested in the effects that the mobile device system settings and context factors have on the energy consumption rate of the device, the samples need to be examined as a time series to estimate the energy drain rate over time. This has been done by grouping all samples according to their user identification token. The grouped tokens are then sorted according to the time that the sample was taken. These sorted samples are then paired up so that the first sample and the second sample make up pair number 1, the second and the third sample make up sample pair number 2 and so forth. These sample pairs are used as the basis of this analysis. The energy rate of a sample pair is calculated as the difference of the samples' battery levels divided by the difference of their time stamps. The set of running applications for a sample pair is decided to be the union of both samples' running applications. For all other system settings and context factors, the more recent sample of the pair is used to determine the context factors and system settings of the sample pair.

Since the Carat data comes from a large number of unsupervised clients, there is no guarantee for the integrity of the data. A faulty device or a hostile client may produce erroneous or tampered data. It is therefore essential to apply proper pre-processing to the data in order to minimize the effect that invalid data points have on further analysis.

Let us give a brief description of each of the system settings and context factors that were used as part of the analysis. Association analysis requires discrete data, as explained in Chapter 4. We therefore describe the way each of these variables is discretized. The following presentation of the data is based on a subset of the Carat data consisting of samples that were collected between 26.8.2016 and 3.10.2016 that had Facebook mobile application running. For increased uniformity and simplicity, the data set only contains samples from clients running Android operating system.

## 3.1 Energy Rate

Energy rate of a mobile device is the velocity at which the mobile devices battery is discharging. The unit of the energy rate is percentage per second. This means that an energy rate of $0.05\frac{\%}{s}$ would drain the whole battery in just $\frac{100}{0.05/s} = 2000s \approx 33minutes$. Any data points where the energy rate

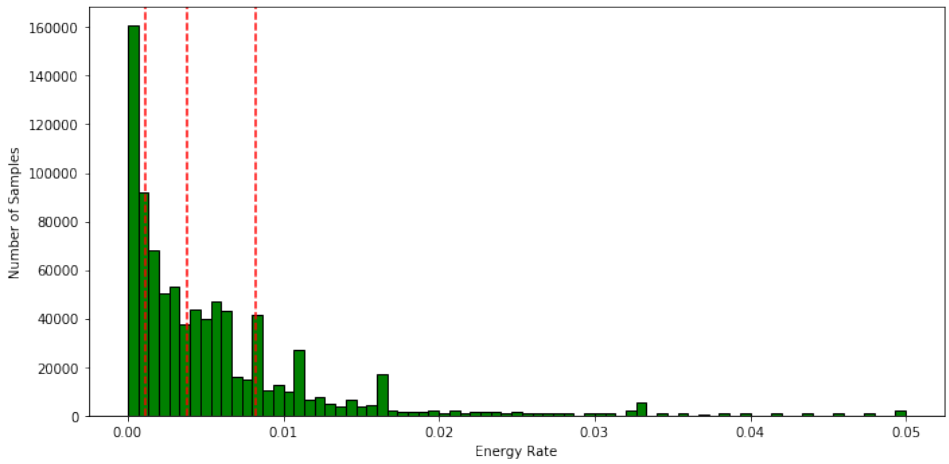



Figure 1: Histogram of energy rates with 75 bins in green. The dotted red lines show the boundaries of the four equal mass bins.

was negative, meaning the device's battery was recharging, were filtered out.

Figure 1 shows a histogram of the energy rate distribution. The distribution appears to be a rough approximation of an exponential distribution. The distribution does not seem have an evident categorical division. Thus, the data was discretized by dividing it to four bins of equal mass. The discretization boundary points along the energy rate -axis were 0.0011, 0.0038 and 0.0083.

### 3.2 CPU Usage Level

CPU usage level is the fraction of time that the central processing unit(s) of the mobile device were busy when the sample was collected. The CPU usage level is in a unit of percentages of the maximum level. All CPU usage levels that were below zero or greater than 1 were discarded as faulty data.

Figure 2 shows a histogram of the CPU usage level distribution. The distribution very roughly approximates the uniform distribution except for 100% and 0% CPU usage levels, which are quite overrepresented. The data was discretized into four bins of equal frequency. The discretization boundary points along the CPU usage -axis were 0.39, 0.63 and 0.85.

High CPU usage rate has been shown to increase the level mobile device energy consumption [23, 20], and has been identified as one of the most significant variables in predicting a device's energy consumption. One would therefore assume the CPU utilization level to appear as an antecedent for rules which predict high energy consumption.

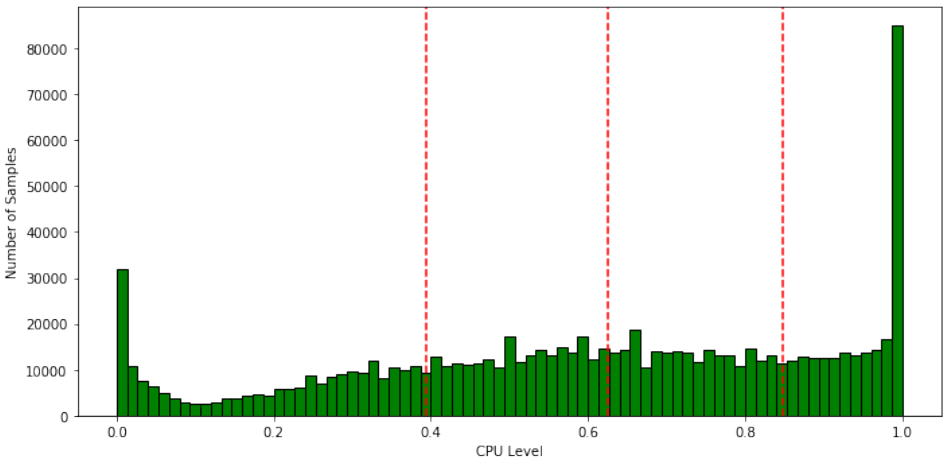



Figure 2: Histogram of CPU usage levels with 75 bins in green. The dotted red lines show the boundaries of the four equal mass bins.

### 3.3 Travel Distance

Travel distance is the distance in meters, that the mobile device moved between the two samples. Moving mobile phone users have been shown to consume less energy on average compared to stationary ones [18]. This could be due to stationary users being more likely to actively use their devices.

Figure 3 shows a histogram of the travel distance distribution. Most of the mass of the distribution is concentrated in the close proximity of zero with other values having very low frequencies. The travel distance was discretized to two categories: *moving*, if the travelled distance was greater than 100 meters, and *static* otherwise.

### 3.4 Battery Temperature

Battery temperature is the measured mobile device battery temperature in degrees Celsius. Temperatures less than five degrees were discarded as it is very rare for a battery temperature to be that low even in subzero climates. Likewise battery temperatures of over 100 degrees were discarded, as healthy devices very rarely reach such high battery temperatures.

Figure 4 shows a histogram of the battery temperature distribution. The distribution approximates a normal distribution with some skewedness. Notably, there is a small cluster of measurements near zero degrees Celsius. This is most likely due to mobile devices systematically reporting a value of zero if the measurement data is not available. The data was discretized into four bins of equal frequency. The discretization boundary points along the battery temperature -axis were 27, 31 and 34.

High battery temperature has been shown to cause increased battery

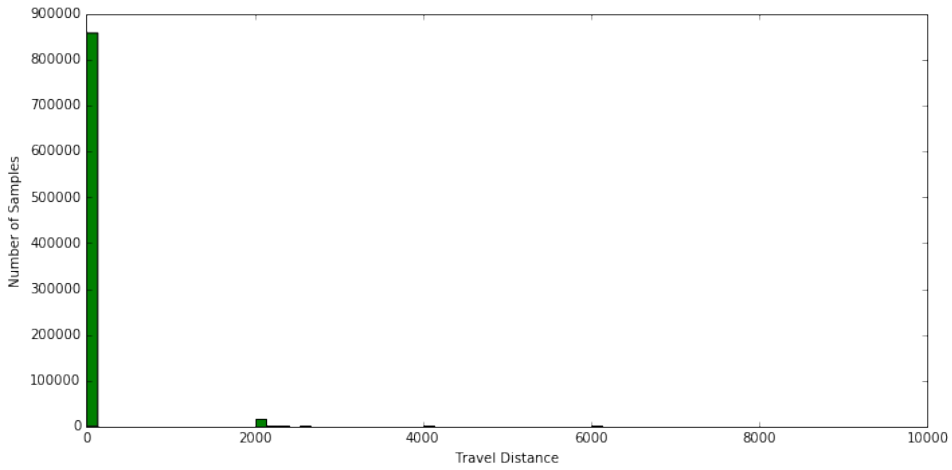



Figure 3: Histogram of travel distance with 75 bins

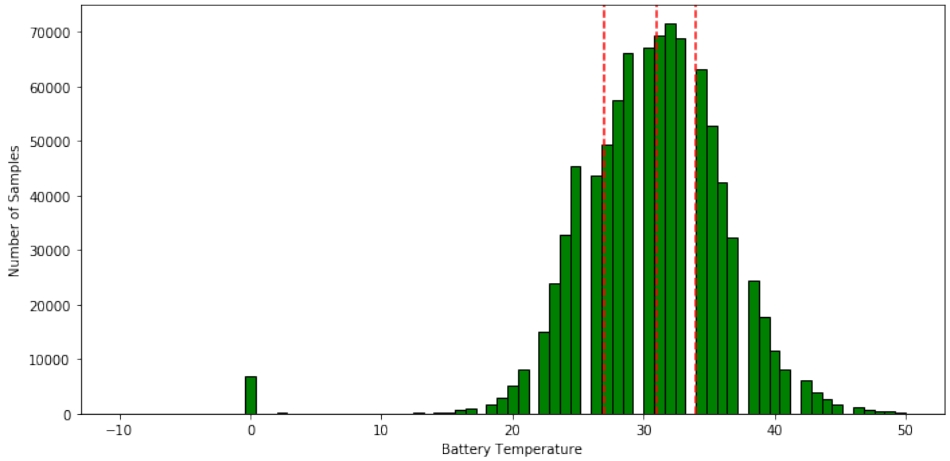



Figure 4: Histogram of battery temperatures with 75 bins in green. The dotted red lines show the boundaries of the four equal mass bins.

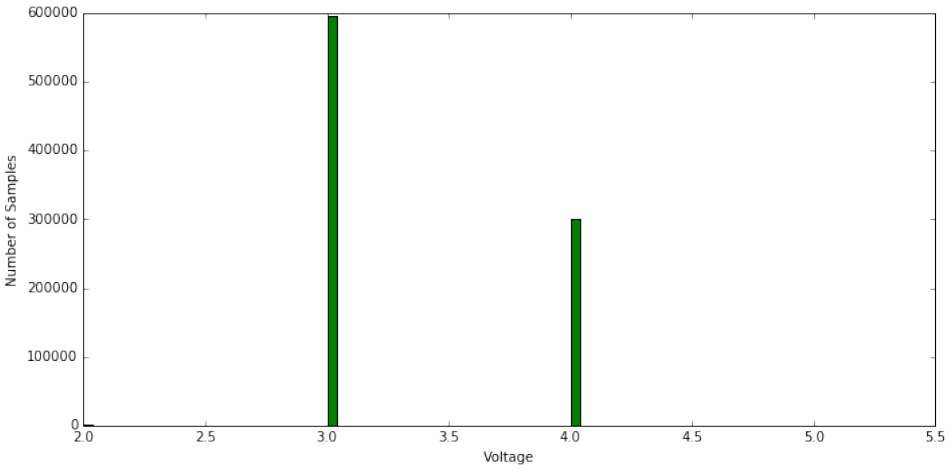



Figure 5: Histogram of battery voltages with 75 bins in green.

consumption [18]. The increase in battery temperature cannot always be explained by CPU usage alone, and other factors such as the ambient temperature can affect the battery temperature. It is to be expected that high battery temperature appears as an antecedent of many rules predicting very high energy consumption.

### 3.5 Battery Voltage

Battery voltage is the electric potential difference generated by the battery in units of volts. Different devices may carry batteries with different voltages. A malfunctioning battery that is near end of its lifetime may give lower than usual voltage readings.

Figure 5 shows a histogram of battery voltage distribution from the Carat data. The voltages are clustered almost discretely around values of 2, 3 and 4 volts. The voltages were accordingly divided to three bins to reflect this clustering. The discrete voltage data is due to a bug in the Carat data collection software which rounds the readings to integer values. This bug affects the data set used in this thesis work, but has since been fixed. By using a more recent dataset, one would be able to use continuous battery voltage readings in the analysis, which would likely increase the predictive power and accuracy of the generated association rules.

### 3.6 Screen Brightness

Screen brightness is system setting that takes integer values between -1 and 255. Higher values correspond to higher screen brightness. The value negative one has a special meaning indicating automatic screen brightness,

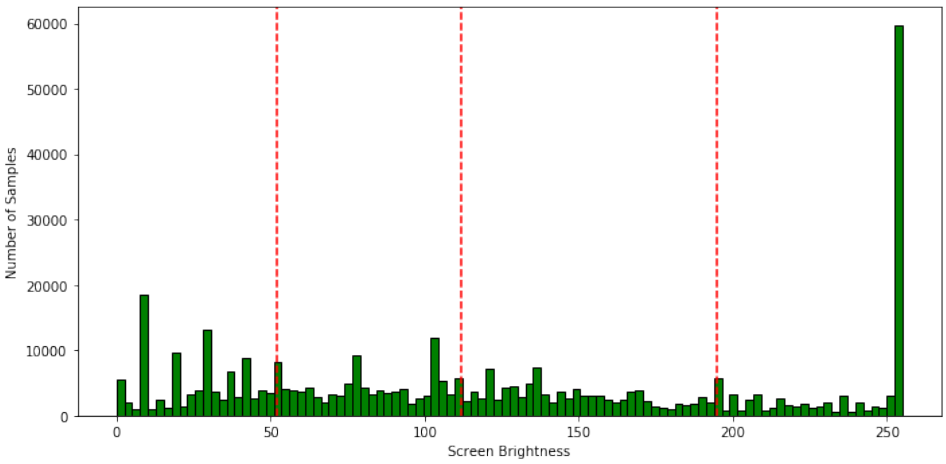



Figure 6: Histogram of screen brightness divided to 100 bins in green. The dotted red lines show to boundary points of the four equal mass bins.

where the screen's brightness is adjusted according to changing illumination of the environment [20].

Figure 6 shows a histogram of the screen brightness values, where the values of -1 have been removed. The brightness values seem to roughly follow a uniform distribution, although very high screen brightness settings seem to be over represented. Approximately half of the samples had their brightness value at -1, indicating automatic brightness setting. Since the automatic setting is categorically different from all other brightness values, the brightness attribute was discretized in the following way: the value -1 formed it's own bin labelled as "auto", while the other numerical values were divided to four bins of equal mass. The boundary points of the equal mass bins along the screen brightness -axis were 52, 112 and 195.

Increase in screen brightness has been shown to cause increased energy consumption in mobile devices [23, 20] while low and automatically adjusted screen brightness values lead to decrease in energy consumption. It is expected that this phenomena shows up in the association rules as well.

### 3.7 Mobile Network Technology

The mobile network technology is a system property that reports the name of the mobile technology that mobile device is using for its mobile data communication. Common values include LTE, HSPA, EDGE and UTMS.

Figure 7 shows a histogram of all mobile network types in the Carat data. Numeric values were mapped to mobile network type names according to Android developer reference manual [1]. Ambiguous values such as "unknown",

[1] https://developer.android.com/reference/android/telephony/TelephonyManager.html

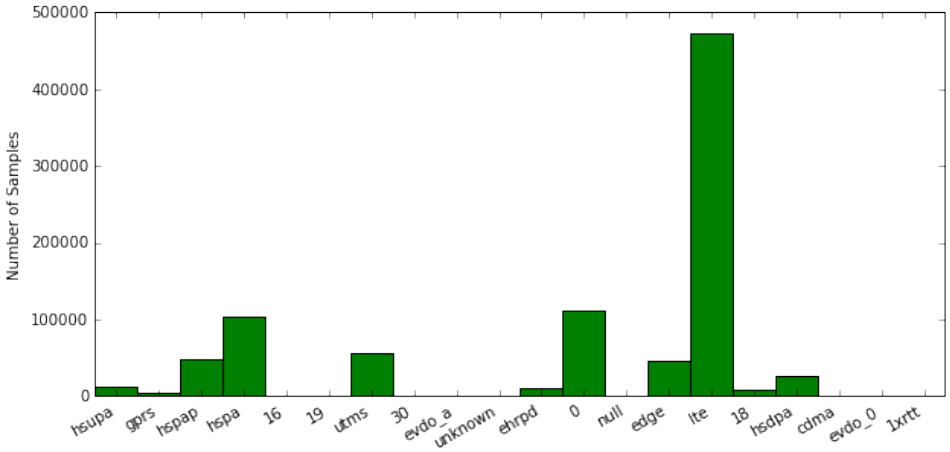



Figure 7: Histogram of mobile network technologies in Carat data.

"null" as well as any numerical value not listed in the Android developer reference manual, were combined to a single bin labelled as "unknown".

Different mobile networking technologies have been shown to have varying energy consumption performance depending on the task [21]. Based on this observation, one would assume that different mobile network technologies should appear as antecedents for association rules predicting either high or low energy consumption depending on the network usage pattern of the mobile application.

### 3.8 Network Type

Network type is a system property that is reported by the mobile device to indicate the type of the data connection. Typically this is either *mobile* or *wifi*, indicating the use of mobile networking or a wireless local area networking respectively. Some more exotic alternatives can however be found in the data, such as *bluetooth tethering*, where the network access is enabled by bluetooth tunneling through another device.

Figure 8 shows a histogram of all network types found in the Carat data. Values *null* and *unknown* were considered ambiguous and were combined under the label *unknown*.

The choice of networking technology has been shown to affect the energy consumption of mobile devices [18, 10]. As a general rule, using WiFi network for downloading data is more energy efficient than using a mobile networking technology. Therefore a reasonable hypothesis is to expect to see association rules with mobile networking type as an antecedent for rules predicting high energy consumption, at least in case of applications that download a lot of data.

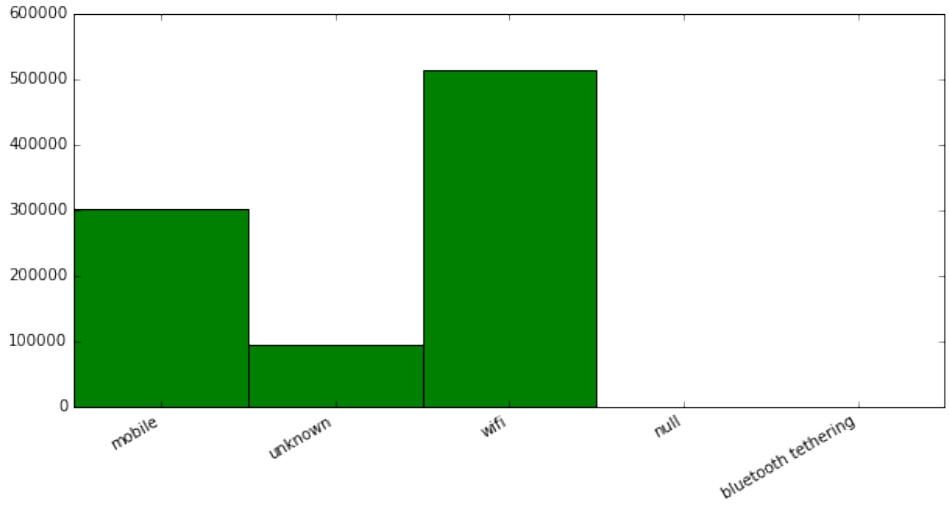



Figure 8: Histogram of network types in Carat data

### 3.9 WiFi Signal Strength

WiFi signal strength is a system property that the mobile device uses to signify the strength of the wireless local area network. WiFi connection strength is reported as an integer value in the range -100 to 0, where 0 signifies the strongest signal. Presumably, the WiFi signal strength is in units of decibels relative to milliwatt (dBm). The Android API seems to report a value of -127 when no reading is available. Values less than -100 were excluded from the data as these are very unlikely to be real measurements.

Figure 9 shows a histogram of WiFi signal strengths in Carat data. The distribution seems to approximate a normal distribution reasonably well. The data was discretized in four bins with equal mass, the boundaries of which were -68.0, -59.0 and -50.0.

Poor WiFi signal strength has been shown to decrease battery life of mobile devices [18]. This is likely due to increase in the noisiness of the connection, leading to increased data loss and retransmissions, which require extra energy. This effect is expected to be seen in the association rules generated for applications that rely heavily on networking.

### 3.10 WiFi Link Speed

WiFi link speed is a system property that the mobile device uses to report the current wireless local area network link speed. The link speed is presumably reported in units of mega bits per second (mbps).

WiFi link speed has been shown to have an impact on the mobile device energy consumption [18, 23]. Increase in link speed seems to lead to an increase in energy consumption, although this connection does not appear

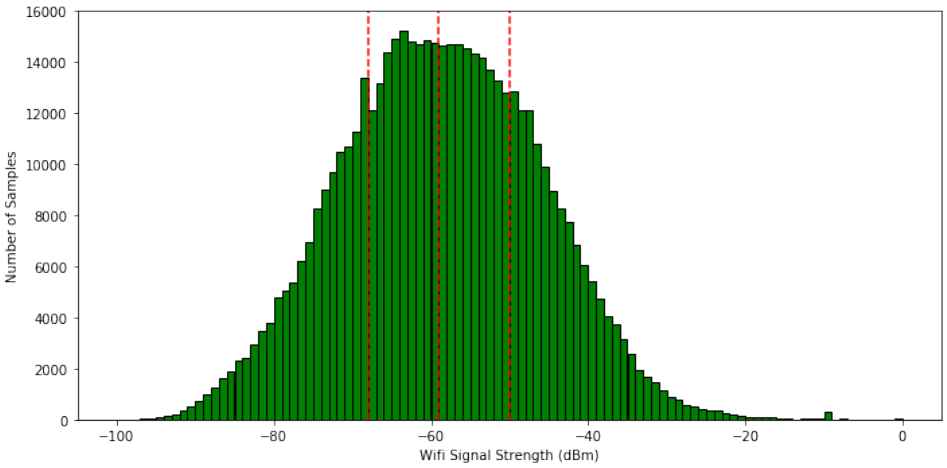



Figure 9: Histogram of WiFi signal strengths readings with 100 bins in green. The dotted red lines show the boundaries of the four equal mass bins.

to be nearly as significant as WiFi signal strength for example. The slight increase in energy consumption might be due to users with faster link being able to consume downloadable content more quickly, thus draining the battery more efficiently.

Figure 10 shows a histogram of the WiFi link speeds in the Carat data. The link speeds were divided to four bins of equal mass. The boundary points of the bins along the link speed -axis were 54.0, 72.0 and 130.0.

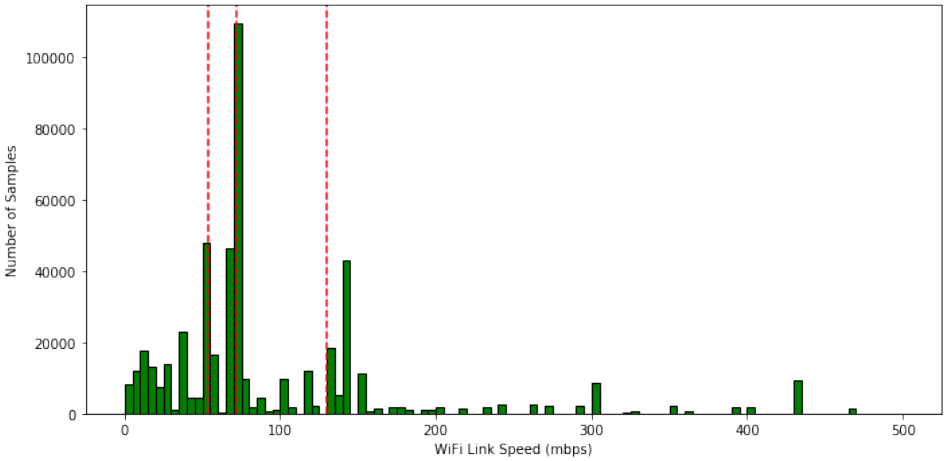



Figure 10: Histogram of WiFi link speed with 100 bins in green. The dotted red lines show the boundary points of the four equal mass bins.

# 4 Association Analysis

Association rule mining is the task of finding associations between *items* in a database of *transactions*. The technique was originally developed in 1993 to identify patterns in consumers grocery purchasing behaviour [2]. Since then however, association analysis has found applications in wide variety of domains.

Association rule mining is an attractive method for modelling non-linear relationships of variables. Many of the mobile device system settings and context variables are highly dependent on one another and therefore their combined effect on the energy consumption is difficult to estimate using linear models. Many non-linear modelling techniques require additional knowledge about the structure of the data. Association rule mining requires relatively few assumptions about the data to be made. In fact the only preprocessing that is necessary in order to apply the association rule mining, is data discretization.

Let us consider a hypothetical dataset shown in table 1. The dataset consists of mobile device system settings and energy usage measurements. The dataset has three continuous valued variables: energyRate, the rate at which the battery is discharging; CPULevel, the device's CPU usage level and screenBrightness, the brightness of the device's screen. Each of these variables takes floating point values ranging from 0.0 to 1.0.

Since association rule mining requires each variable of the database to be binary valued, a discretization of the variables must be performed. To discretize a continuously valued variable, we need to replace the continuous variable with multiple binary valued variables, each corresponding to an interval or cluster of values of the continuous variable. The details of discretization of Carat data are discussed in Chapter 3. For simplicity, let us consider a simple discretization strategy, where each continuous variable is split to two binary variables by creating two bins at cut point 0.5. Tables 2 and 3 demonstrate this idea.

Since every group of variables that is created by discretization is mutually exclusive, a more concise notation for this dataset can be used, as shown in Table 4.

Having transformed the raw data to binary variables, the goal of the

| **energyRate** | **CPULevel** | **screenBrightness** |
|---|---|---|
| 0.21 | 0.58 | 0.30 |
| 0.80 | 0.46 | 0.61 |
| 0.76 | 0.65 | 0.93 |
| 0.58 | 0.99 | 0.54 |

Table 1: Hypothetical mobile device measurements inspired by Carat dataset

| **energy=low** | **energy=high** | **CPU=low** |
|---|---|---|
| True | False | False |
| False | True | True |
| False | True | False |
| False | True | False |

Table 2: Hypothetical mobile device measurements after discretization using long notation. First half of the columns.

| **CPU=high** | **screen=low** | **screen=high** |
|---|---|---|
| True | True | False |
| False | False | True |
| True | False | True |
| True | False | True |

Table 3: Hypothetical mobile device measurements after discretization using long notation. Second half of the columns.

| **energyRate** | **CPULevel** | **screenBrightness** |
|---|---|---|
| low | high | low |
| high | low | high |
| high | high | high |
| high | high | high |

Table 4: Hypothetical mobile device measurements after discretization using concise notation

association analysis is then to produce a list of association rules, given some measure of interestingness. For the database given above, an association rule mining algorithm might find the following association rule

$$\{CPULevel = high, screenBrightness = high\} \Rightarrow \{energyRate = high\}$$

This rule implies that high CPU utilization together with high screen brightness associates with high level of energy consumption.

## 4.1 Formal Problem Definition

Let $I = \{x_1, x_2, ..., x_n\}$ be a set of binary variables called items. A transaction database $T$ is then a multiset of subsets of $I$, where each element of $T$ denotes a transaction. To give the exact problem of association rule discovery, concepts of support and confidence need to be introduced.

Support of an item set $X$ in database $T$ is defined as the fraction of all transactions in $T$ that contain the item set [9].

$$supp(X) = \frac{|\{X' \in T \mid X \subseteq X'\}|}{|T|}$$

Confidence of a rule $X \Rightarrow Y$, where $X$ and $Y$ are item sets of $T$, is defined as the fraction of transactions in $T$ containing item set $X$ which also contain $Y$ [9].

$$conf(X \Rightarrow Y) = \frac{supp(X \bigcup Y)}{supp(X)}$$

The problem of association rule discovery can now be formalized the following way. Given a transaction database $T$, minimum support level $s$, where $0 \leq s \leq 1$ and minimum confidence level $c$, where $0 \leq c \leq 1$, find all rules $X \Rightarrow Y$ where $conf(X \Rightarrow Y) \geq c$, $supp(X) \geq s$ and $supp(Y) \geq s$ [9].

The association rule discovery problem can be further divided into two distinct sub-problems, namely frequent pattern mining problem and rule generation problem. A frequent pattern $P$ of database $T$ is a subset of $I$ such that $supp(P) \geq s$. The frequent pattern mining problem is the task of finding all frequent patterns from a given database. The rule generation problem on the other hand, is the task of generating all association rules with sufficient confidence from the frequent patterns.

## 4.2 Frequent Pattern Mining Using Frequent Pattern Growth

Frequent pattern growth is an efficient algorithm for the frequent pattern mining problem [8]. The FP-growth algorithm has been shown to outperform the time consumption of traditional Apriori pattern mining algorithm [3]

by more than an order of magnitude when mining large databases [8]. The algorithm utilizes a specialized data structure called FP-tree, a kind of prefix tree, to speed up the frequent pattern generation. The FP-tree data structure consists of nodes, each of which have tree fields: item name, item count and node link. The item name tells which item the node represents. The item count signifies the number of transactions containing the item, that can be reached by following the path of nodes in the FP-tree leading to this node. The node link contains a pointer to the next node in the FP-tree with the same node name. An exception to this format is the root node of the FP-tree, which does not have any of these fields, but only has links to child nodes. Algorithm 1 shows how to constructs a FP-tree for a transaction database [8].

The algorithm consists of two procedures. The entry point for the algorithm is the *buildFPTree*-procedure, which takes no parameters and returns the newly built FP-tree. The sub-program *insertTree* takes a transaction *Trans*, that is sorted by descending item frequency, and an incomplete FP-tree $T$ as its parameters. The *insertTree* procedure walks down $T$ in a path determined by the items in *Trans*, incrementing the counts of nodes on the path. If there is no existing path to walk down on at any point, the *insertTree* creates a new node with count 1 as a child of the last node that was walked over.

The *buildFPTree* procedure proceeds the following way. In line 1, the transaction database is scanned and the different item names together with their frequencies are collected to variable $F$. In line 2, keys of $F$ are sorted in descending order of support and items which have support less than minimum support $s$. The sorted and filtered key-count-pairs are stored in variable $L$. In line 3, the root of node of the FP-tree is created and stored in variable $T$. In line 4, a for loop that iterates over each transaction in the transaction database is entered. The loop yields the transactions in a variable called *Trans*. In line 5, the contents of variable *Trans* are sorted according to the ordering induced by $L$ and infrequent items (items which have support that is less than $s$) are filtered out of *Trans*. In line 6, the sub-program *insertTree* is called passing parameters *Trans* and $T$. The for loop entered on line 4 is exited. Finally on line 7, $T$, which now contains the complete FP-tree, is returned.

The *insertTree* procedure consists of the following steps. In lines 1-3, we check if parameter *Trans* is empty and return if so. In lines 4-5, we store the first item of parameter *Trans* in variable $N$ and the rest of the items in variable *tail*. In line 6, we enter an if statement with the conditional " $T$ has a child $h$ such that $h$ has the same item name as $N$". In line 7 we increment the count of $N$ by one. In line 8 we call *insertTree* recursively with parameters *tail* and $h$. In line 9, we exit the then-branch of the if statement and enter the else-branch. In line 10, we create a new node with item-name equal to $N$, count equal to 1 and we link $T$ as the node's parent. The newly created

```
Data: A transaction database DB, minimum support threshold s
Result: FP-tree for the database
Algorithm buildFPTree(DB, s)
1   | F ← Scan DB and collect a map of items and their frequencies
2   | L ← Sort keys of F in descending order of support filtering out
    | items which have support < s
3   | T ← root node of the FP-tree
4   | for transaction Trans in DB do
5   |   | sort Trans according to L and filter out infrequent items
6   |   | insertTree(Trans, T)
    | end
7   | return T
Procedure insertTree(Trans, T)
1   | if Trans is empty then
2   |   | return
    | end
3   | else
4   |   | N ← first item of Trans
5   |   | tail ← tail of Trans
6   |   | if T has a child h such that N.item-name == h.item-name
    |   | then
7   |   |   | h.count += 1
8   |   |   | insertTree(tail, h)
    |   | end
9   |   | else
10  |   |   | n ← create new node with item-name = p.item-name and
    |   |   | count = 1 and link T as its parent
11  |   |   | insertTree(tail, n)
    |   | end
    | end
```

**Algorithm 1:** Fp-tree construction

| Items | Frequency Ordered and Filtered Items |
|---|---|
| a, b, c, d, e, h | c, e, h, a, b, d |
| c, d, e, h | c, e, h, d |
| c, e, h | c, e, h |
| a, b, c, e, j | c, e, a, b |
| c, h | c, h |
| d, i | d |

Table 5: Example transaction database for illustrating FP-tree generation.

node is stored in variable *n*. In line 11 we call *insertTree* recursively with *tail* and *n* as its parameters.

To better illustrate the process of building an FP-tree from a transaction database, let us go through an example that uses a simple made up transaction database. First step of building a FP-tree, is to sort each transaction by decreasing order of frequency of its items in the database. Infrequent items are also filtered out based on the minimum support. Table 5 shows the example transaction database as well as the frequent items in descending order of frequency. A single traversal over the database is required in order to sort the transactions.

For this example, let us consider a minimum support of 0.3. Since the number of transactions in our database is 6 and $\frac{1}{6} < 0.3 < \frac{2}{6}$, all items with frequency less than 2 can be filtered out. Thus items i and j are not included in the second column of table 5.

To construct the FP-tree we start with a node labelled as "root". We then take the first transaction and walk over its elements in the frequency order. We start from the root node of the FP-tree. Since the roo node has no child nodes, we add a child node labelled with "c" and set its count to 1. We remove the item "c" from the transaction being handled. Then we continue the walk from the node labelled with "c". The procedure is repeated with the remaining items of the transaction, continuing the walk from the node labelled "c". The FP-tree after processing the first transaction is shown in Figure 11.

Next, the second transaction is processed as follows. We again start walking down FP-tree from the root node processing items in the second transaction. Since the root node has a child node labelled with "c" we increment its count to 2 and remove item "c" from the transaction being processed. Since the node has a child node labelled with "e" and our next item to be processed is "e", we move to the node labelled with "e" and increment its count by one and remove item "e" from the transaction being processed. The same process is applied to item "h" and the node labelled with "h". The next item to process is "d", but the node labelled with "h" does not have a

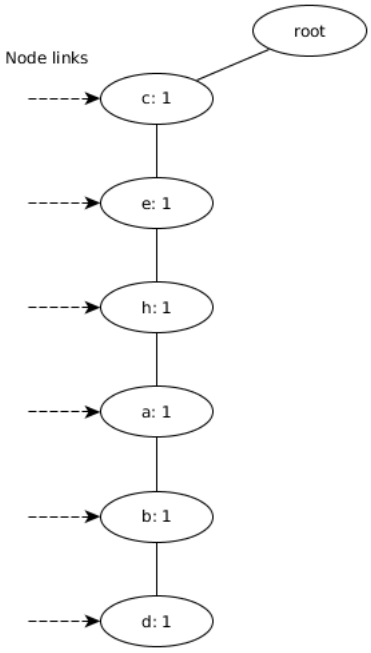



Figure 11: FP-tree after processing the first transaction

child node labelled with "d". We therefore add a child node labelled with "d" and add 1 to the count of the node labelled with "h". Figure 12 shows the progress after processing the second transaction of the database.

Repeating the same procedure for the rest of the transactions yields complete FP-tree as illustrated in Figure 13.

The dashed arrows in Figure 13 represent node links. The FP-tree maintains a table of linked lists for each distinctly named item. Whenever a node is added to the FP-tree, a pointer to that node is also added to the linked list corresponding to that nodes label.

The way an FP-tree is constructed guarantees, that all FP-trees have the so called *node-link property* [9]. What it means, is that for any frequent item, all frequent patterns containing that item can be constructed by following the item's node-links starting from the item's head in the FP-tree header. Let us consider the FP-tree in Figure 13. Starting from the bottom of the node links, we examine which frequent patterns can be extracted by following the node link. For item "d", we can trivially extract a frequent pattern "d:3". In addition, we have to consider the three prefix paths of "d" which are accessible from the node links of item "d": (c:5, e:4, h:3, a:1, b:1, d:1), (c:5, e:4, h:3, d:1) and (d:1). These three paths form the *conditional pattern base* [9] of "d". To extract all frequent patterns containing item "d", we start by substituting all the item counts in each of the prefix paths in the conditional pattern base of "d" with the count of "d" in the prefix path. This

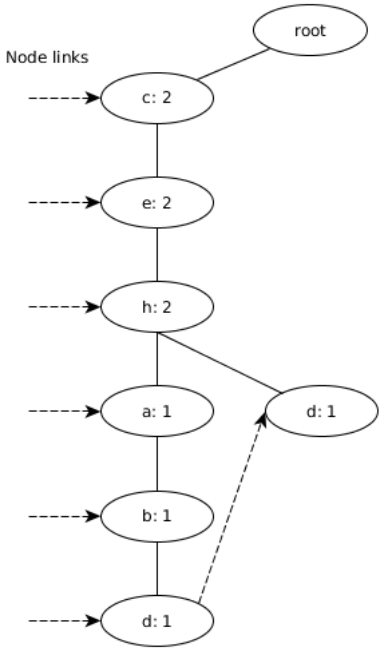



Figure 12: FP-tree after processing the second transcation

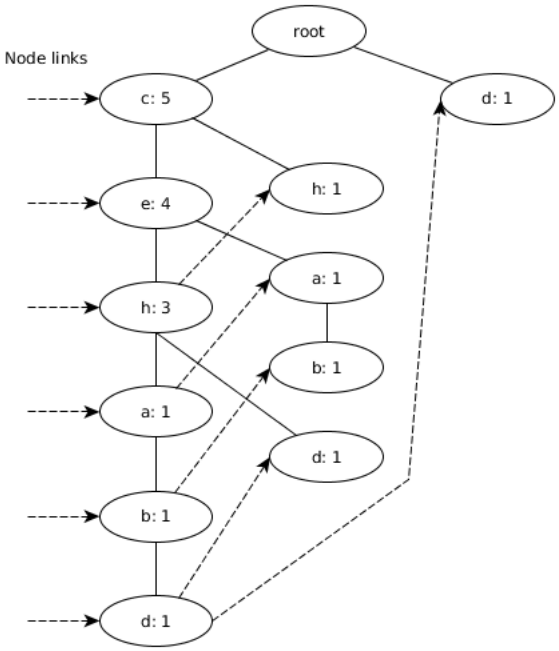



Figure 13: Complete FP-tree

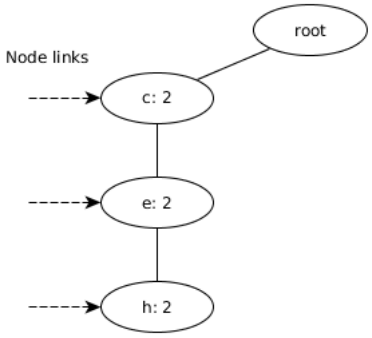



Figure 14: Conditional FP-tree of item d.

is done to account for the fact that on each of the prefix paths of "d", the other items appear together with "d" exactly as many times as "d" itself. We also leave out the element "d" itself. We now get the transformed conditional pattern base (c:1, e:1, h:1, a:1, b:1), (c:1, e:1, h:1). The next step towards the frequent patterns containing "d", is to construct a *conditional FP-tree* of "d" [9]. The construction of the conditional FP-tree conforms to the same set of rules as a normal FP-tree. The only difference is that a prefix path in the conditional pattern base, such as (x:n, y:n, z:n) should be interpreted as $n$ distinct transactions containing the items x, y and z. Figure 14 shows the conditional FP-tree of item d constructed from the transformed conditional pattern base. Since there is only a single path in the conditional FP-tree, one can generate all the remaining frequent patterns containing item d by taking all combinations of items c, e, h and concatenating them with d. Each such pattern has support equal to the minimum support among the items in the combination. In this case, we would get frequent patterns {cd:2, ed:2, hd:2, ced:2, chd:2, ehd:2, cehd:2} in addition to the trivial pattern d:3. But what would happen if the conditional FP-tree contained multiple paths? In this case one would recursively construct a new conditional FP-tree for the item whose node link yields multiple paths. Following this procedure for each node link in the original FP-tree, one can generate all the frequent patterns.

Formalizing the method discussed above yields Algorithm 2 [9]. The algorithm *FPGrowth* takes as its arguments an FP-tree and a minimum support threshold. In line 1 of the algorithm, a procedure *patternGrowth* is called with the FP-tree and an empty sub-pattern denoted by value *null*. The procedure *patternGrowth* has two parameters. The first parameter is a conditional FP-tree and the second parameter is the sub-pattern for which the conditional FP-tree was built. Initially, the procedure will be called with the whole FP-tree and empty sub-pattern. The algorithm will recursively simplify the conditional FP-tree until the tree only contains a single branch, from which the frequent patterns can be easily generated. In line 1, the

**Data**: *FP-tree* constructed using Algorithm 1, minimum support threshold $s$

**Result**: All frequent patterns of the database

**Algorithm** `FPGrowth(`*FP-tree, s*`)`

```
1  patternGrowth(FP-tree, null)
```

**Procedure** `patternGrowth(`*tree, sub-pattern*`)`

```
1  if tree contains only single path P then
2      for each combination C of items in P do
3          generate frequent pattern C ∪ sub-pattern with support =
           minimum support of items in C
       end
   end
4  else
5      for each node in the header of tree do
6          C ← pattern node.item ∪ sub-pattern with support =
           node.support
7          generate pattern C with support = node.support
8          B ← construct conditional pattern base of C constrained by
           minimum support threshold s
9          T ← construct the conditional FP-tree of C from B
           constrained by minimum support threshold s
10         if T is non-empty then
11             patternGrowth(T, C)
           end
       end
   end
```

**Algorithm 2:** Fp-growth

procedure checks if the conditional FP-tree only has one branch. If so, the frequent patterns of the conditional FP-tree are outputted in lines 2-3 after which the procedure is finished. If the conditional FP-tree had more than one branch, we go to line 5, where we start to iterate over all nodes reachable from the conditional FP-tree's header's node-links. In line 6, we create a new pattern by appending the node's item to the current sub-pattern. The pattern is stored to variable $C$. The new pattern's support is set to be equal to the support of the node that is being iterated. In line 7, the pattern $C$ is appended to the output of the program. In lines 8-9, a conditional FP-tree of pattern $C$ is constructed and stored in variable $T$. In lines 10-11 we recursively call *patternGrowth* with the newly constructed conditional FP-tree $T$ and pattern $C$. After line 11, we exit the iteration loop and the procedure is finished.

### 4.3 Generating Association Rules from Frequent Patterns

The frequent patterns may reveal some interesting structures in the transaction database, as they effectively capture frequent co-occurrence of various items. Although this may be enough for some application domains, it is often the case that one would ultimately want to reveal some causal relationships between items or sets of items in the database. The association rules offer a rudimentary but fairly scalable solution for discovering "if X then Y" type of relationships between subsets of items in the frequent patterns. It should be noted here, that the field of causal inference expands far beyond scope of this thesis work and the association rules should at best be considered as candidates for causal relationships, not as proofs of such.

Given a set of frequent patterns $F$, and a minimum confidence threshold $c$, the task of generating the association rules can be accomplished by the following method:

1. For each set of items $I$ in $F$
    1.1. For each non-empty subset $A$ of $I$
        i. If $conf(A \Rightarrow I \setminus A) \geq c$ then output rule $A \Rightarrow I \setminus A$

While the proposed method does work when the size of the frequent patterns is small, this approach quickly becomes infeasible as the length of the patterns grow. Given a frequent pattern $I$ with $\mid I \mid$ items, there are a total of $2^{|I|}$ subsets of which $2^{|I|} - 2$ need to be tested as the set $I$ itself and the empty set can be discounted. Fortunately, a technique exists for pruning the search space, as described by Agrawal et al. [4]. The confidence measure has the anti-monotone property, meaning that for any frequent pattern $I = \{A, B, C, D\}$ the following inequality holds:

$$conf(ABC \Rightarrow D) \geq conf(AB \Rightarrow CD) \geq conf(A \Rightarrow BCD)$$

**Data**: A set of frequent patterns $F$, a minimum confidence threshold $c$
**Result**: Association rules discovered from the frequent patterns
**Algorithm** `generateRules(`$F$, $c$`)`

```
1   for each frequent pattern P in F do
2       C ← candidate rule generator of P ordered by increasing
        length of candidate rule's consequent set length
3       while C has more candidate rules do
4           r ← get next candidate rule from C
5           if conf(r) ≥ c then
6               output r
            end
7           else
8               Remove all candidates rules r' from C where
                r.consequents ⊂ r'.consequents
            end
        end
    end
```

**Algorithm 3:** Generating association rules from frequent patterns

Since moving items from the antecedent set to the consequent set never increases the confidence of the rule, one can potentially prune an enormous amount of candidate rules by iterating the candidate rules in ascending order of length of the candidate rule's consequent set [4]. After each iteration, if the candidate rule being processed did not have high enough confidence, then all remaining candidate rules whose consequent set contains a subset of the current candidate rules consequent set, can be pruned since the anti-monotone property guarantees that any such rule has confidence no higher than the current candidate rule.

Algorithm 3 gives an outline on how to generate association rules from frequent patterns with search tree pruning. The details of how to generate the combinations of items and how to implement the pruning of the search space are not presented here. The algorithm takes a set of frequent patterns and a minimum confidence threshold as its arguments and returns the set of association rules discovered from the frequent patterns. In line 1 we start to iterate over the frequent patterns. In line 2 we initialize the candidate rule generator storing it to variable $C$. In line 3, we enter a while loop which is terminated when generator $C$ has no more candidate rules to generate. In line 4 we store the next candidate rule from $C$ to variable $r$. In lines 5-6, we test if the confidence of candidate rule $r$ is greater or equal to the minimum confidence threshold and output the rule if the criteria is met. If the confidence criteria was not met, we prune the candidate rule space of generator $C$ in lines 7-8.

# 5 Spark and the MapReduce Programming Model

MapReduce is a programming model and an associated implementation that emerged to simplify common tasks associated with big data processing. These include managing parallel and distributed computing and ensuring fault tolerance of the computations [5]. The model is heavily influenced by functional programming, a programming paradigm that emphasises the use of pure functions and avoiding mutable state. As the name *MapReduce* suggests, the computational model of a MapReduce system is based on two higher order functions, *map* and *reduce*.

Spark is a MapReduce system implemented in the Scala programming language that is built around an abstraction called Distributed Resilient Datasets (RDDs) [25]. The RDD abstration allows Spark to implement efficient fault tolerance. A common way of achieving fault tolerance is by maintaining multiple redundant copies of all datasets. Each time a dataset is mutated, all copies are mutated as well. While this certainly makes the system tolerant to lost datasets, it imposes quite significant overhead as each update needs to be replicated and extra space is required by the copies. Instead of maintaining redundant copies of each dataset, Spark solves the problem of fault tolerance by maintains the lineage of its datasets. The lineage of an RDD is a collection of instructions that specify how the RDD was computed from other RDDs. In case the Spark system loses an RDD, it can recreate the lost dataset by tracing its lineage back to existing RDDs and applying the instructions to recompute the lost dataset.

The Spark runtime consists of a *driver* application and one or more *worker* application to which the driver application connects to and assigns tasks [25]. In a typical scenario, the *worker* applications reside on separate compute nodes of a computer cluster.

The programming interface that Spark provides is analogous to the collection library in the Scala standard library. The interface offers two types of functions: transformations that construct new RDDs from existing ones, such as *map* and *filter*, and actions that either save data to disk or return values to the application, such as *collect* and *reduce* [25]. Transformations are computed lazily, which allows pipelining consecutive transformations and constructing a lineage graph of the computation before computation even takes place. Actions are used to execute work flows constructed by transformations and return the results to the application or write them to the disk. The map function, defined for class *RDD[T]*, where $T$ is the element type parameter of the RDD, has essentially the following type signature

$$map[U](f : (T) \Rightarrow U) : RDD[U]$$

The function produces a new RDD with element type $U$ by applying function $f$ to each element of the original RDD. Similarly, function *filter* has

the type signature

$$filter(f : (T) \Rightarrow Boolean) : RDD[T]$$

and works by constructing a new RDD by including those elements of the original RDD where predicate $f$ is true. The action *collect* has the simple type signature

$$collect() : Array[T]$$

as it merely returns the results of the computation performed by transformations on the RDD to the application. The *reduce* action has the signature

$$reduce(f : (T, T) \Rightarrow T) : T$$

and it works conceptually by applying the binary operator $f$ iteratively to elements of the RDD until there is only one element left. The order and pairing of the elements is not specified to allow parallel computation and thus the operator needs to be commutative and associative in order to guarantee deterministic results.

## 5.1 Spark Usage

Let us look at a simple example to illustrate the usage of Spark. Suppose we have a text file where each line contains a decimal number and we wish to calculate the sum of all positive numbers in the file. The following Scala program implements this task.

```
1  object SumNumbers {
2
3    def initSpark(): SparkSession = {
4      val sparkMaster = Properties
5        .envOrNone("SPARK_MASTER").get
6
7      SparkSession.builder()
8        .appName("SumNumbers")
9        .master(sparkMaster)
10       .getOrCreate()
11   }
12
13   def main(args: Array[String]) {
14     val session = initSpark()
15     val sc = session.sparkContext
16
17     val numbers: RDD[Double] = sc
18       .textFile(args(0))
19       .map(_.toDouble)
20     val positive: RDD[Double] = numbers
21       .filter( _ >= 0)
22     val sum: Double = positive.reduce(_ + _)
23     println(s"Sum of positive numbers is ${sum}")
24   }
25 }
```

Explicit type annotations are included at lines 17, 20 and 22 to clarify the execution context of values *numbers*, *positive* and *sum*. The value *numbers* as well as value *positive* describe transformations applied to an RDD of strings, created at lines 17 and 18 by calling $sc.textFile(args(0))$. This reads the text file given as the first command line parameter to the program and creates an RDD whose elements are the lines in the text file. In line 19, the RDD of strings is converted to an RDD of double precision floating point values by applying method *toDouble* to each element. In lines 20 to 21, the RDD of doubles is further transformed by calling *filter(_ >= 0)*. The filter function, as explained above, applies a predicate to the RDD returning a new RDD containing only the elements whose value is greater than or equal to zero. Finally, at line 22, the call to action *positive.reduce* causes the computation defined by the RDD objects to materialize and calculates the sum of elements of RDD *positive*. Readers who are not familiar with Scala programming language might find expressions such as *_.toDouble* and *_ + _* slightly confusing. These expressions are merely syntactic sugar for expressing anonymous functions and can be written equivalently as $x => x.toDouble$ and $(x, y) => x + y$ respectively.

Each RDD consists of partitions, which are independent subdivisions of the dataset [25]. Each partition may be computed in parallel or even on different machines. The programmer can control the number partitions that an RDD should be divided into by calling the RDD object's *repartition* method and providing the desired number of partitions. Partitions have their individual lineage graph specifying which partitions of the parent RDDs they depend on. When executing a work flow, Spark groups together transformations that can be pipelined to form stages. An RDD operation that depends on multiple RDDs will create a stage boundary, because both dependency RDDs will need to be materialized before the operation can be executed.

Figure 15 shows a lineage graph of the example work flow discussed above, assuming the default number of partitions is three. The tall rectangles represent RDDs and the small squares inside represent the partitions of the RDDs. Above each RDD is the name of the function that is responsible for creating the RDD. All the RDDs are wrapped inside a single stage, because the operations can be pipelined. The black arrows in between the partitions represent the lineage of the partitions.

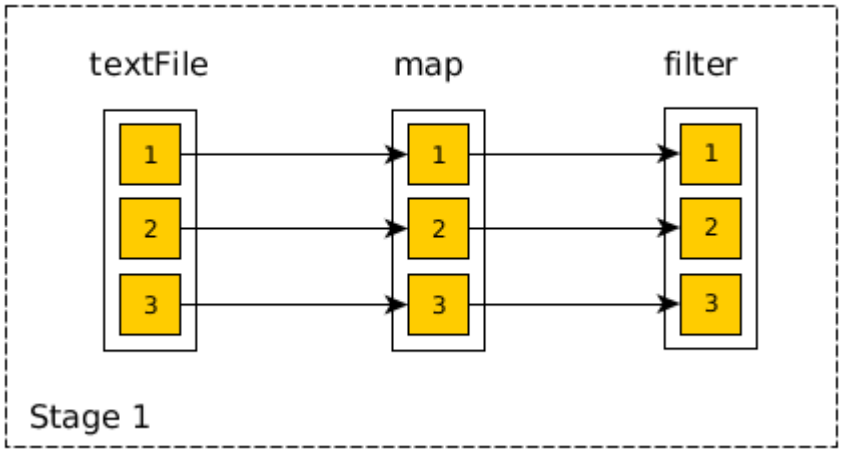



Figure 15: Lineage graph of the example Spark work flow. Tall rectangles represent RDDs and squares inside them represent partitions.

# 6 Implementation

This section discusses the implementation of the Carat developer API prototype. The API aims to provide mobile application developers the ability to discover how a client mobile device system settings and context factors affect the energy consumption of the device.

The design of a service like this poses multiple challenges, as discussed by Peltonen et al. [19]. The dataset itself is large and incrementally changing as time passes, which makes static statistical analysis inconvenient. It is therefore practical to design the service in such a way that the analysis can be executed dynamically whenever the API is accessed. Another challenge is to protect the privacy of the participants of the Carat project.

Association rules were selected as the basis of the API for a number of reasons. The association rules effectively hide all the details about individual Carat users, protecting their privacy, while also enabling reasonably detailed view on how a devices system settings and context factors affect the energy consumption. Association rule mining also allows convenient data processing, as discussed in Chapter 3. Finally, efficient parallel algorithms exist for generating association rules from huge datasets as discussed in detail in Chapter 4.

As discussed by Peltonen et al. [19], the intention of the Carat API is to allow application developers to retrieve information about their application by authenticating with their developer key. The prototype described here, does not include authentication of application developers, but rather demonstrates the functionality that the API could offer provided that an authentication has been successfully completed.

The implementation consists of three main components. These are the front end web server, back end web server and the analysis engine. Figure 16 shows network level layout of these components and the way these components communicate when the API is accessed. When a client accesses the API, the following flow of requests takes place:

1. Client sends a HTTP request to the front end server. The request may contain parameters that control the way the association rules are generated.

2. Front end web server sends a HTTP request to the back end web server passing along the parameters from the client.

3. The back end web server dispatches a job to the analysis engine running on Spark-cluster. Parameters provided by the client are used to control the analysis.

4. Analysis engine sends generated association rules to the back end web server.

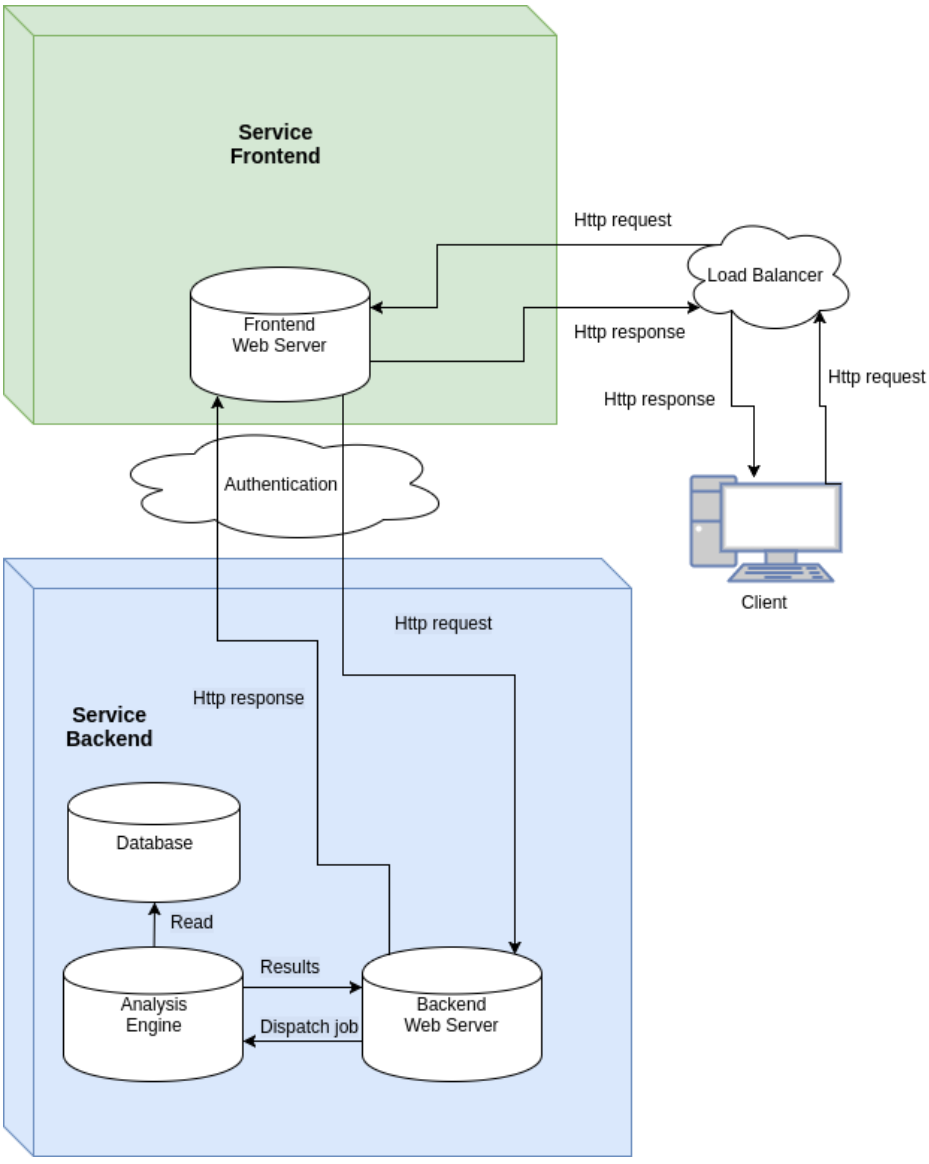



Figure 16: High level network architecture of the Carat API prototype

5. Back end server sends HTTP response containing the generated rules to the front end web server.

6. Front end server uses the association rules to generate a view for the client.

Figure 16 shows the flow of control between the different components of the Carat API. The prototype only implements the frontend and the backend of the service. Implementing load balancing and authentication falls outside the scope of this thesis. Some implementation constraints of the authentication module are discussed by Peltonen et al. [19].

## 6.1 Service Front End

The service front end is implemented using a simple web server written in Scala programming language using Scalatra web framework version 2.4.0 and using Javascript programming language to create the interactive content in the web based graphical user interface. Figure 17 shows to graphical user interface of the service. Initially, a search form will be displayed to the user, containing three input fields for specifying the name of the application to be analysed, the minimum support threshold and the minimum confidence threshold. Additionally, the form contains a check box for each variable to be excluded from the association rule generation.

The first input field accepts the name of the application. The input field was constructed using selectize.js, a Javascript library for creating searchable drop down lists. As the user starts typing the name of the application in the input field a list of matching application names will be displayed under the input field. By clicking an application name in the list, the user can select the application of interest. To enable this functionality, the application will fetch a list of all available application when the web page loads using an AJAX request to the web server. Figure 18 shows how the searching and selecting of applications by name works in practise. In this imaginary scenario, the user is searching applications whose name includes the string "spot". The user is then given a list of options from which to choose. Application names such as "com.spotify.music", "fr.pb.freewifispot" and "ekawas.blogspot" will be displayed as each of them contain the substring "spot". The matching sections of the names are highlighted in light blue color.

The second and third input field allow the user to specify the minimum support threshold and the minimum confidence threshold respectively, for the association rule generation. Above each of these two input fields are icons with question marks. These icons, when hovered over, will display an explanation about the variable in question. This functionality is illustrated in Figure 19 for the minimum support threshold. The icon for minimum confidence threshold functions similarly.

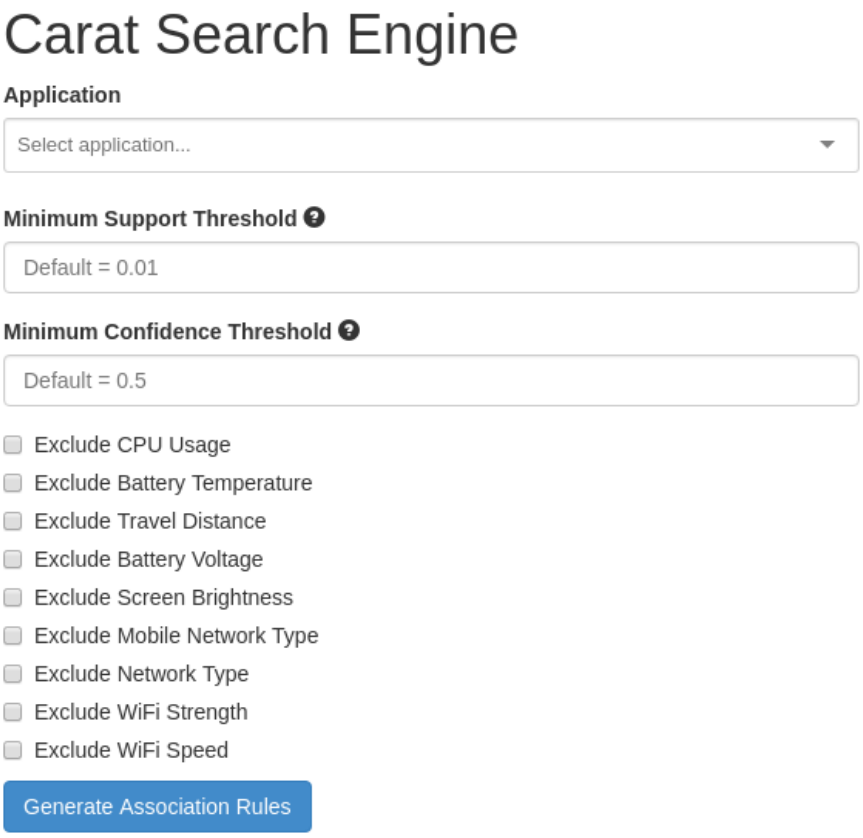



Figure 17: Overview of the graphical user interface of the front end

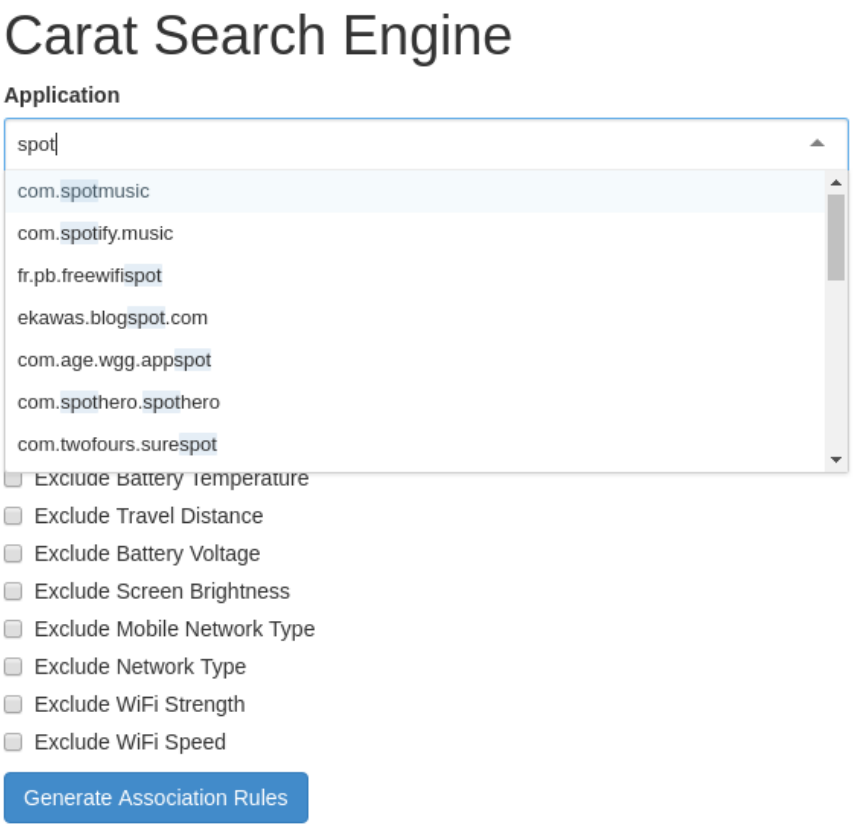



Figure 18: Searching and selecting applications by name

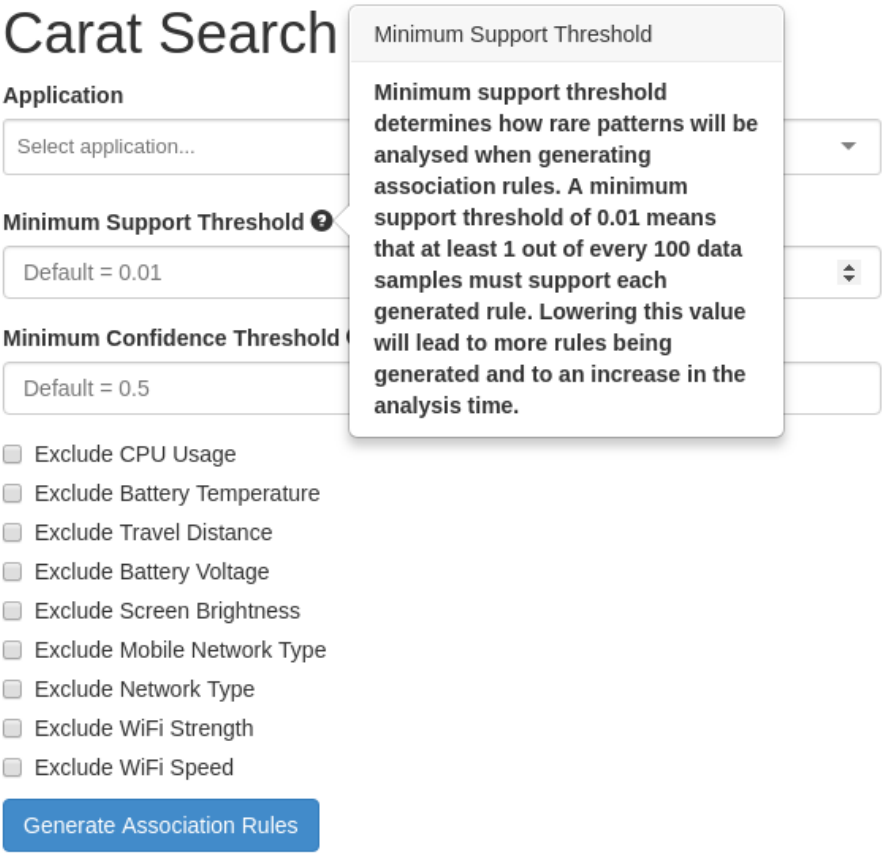



Figure 19: Helper icon pop over explaining the meaning of minimum support threshold

Finally, the form contains a check box for each variable that is included in the association analysis. By checking any of these check boxes, the user can exclude a variable from the analysis.

Once the user initiates the analysis by pressing the submit button, labelled with "Generate Association Rules", a spinner element is displayed to tell the user that analysis is in process. Once the analysis is complete and the back end returns the analysis results, the spinner element will be removed and the generated rules will be displayed underneath the search form. The rule list is organized in tabs, one for each percentile of the energy rate variable. By clicking on a tab, the rules in which the corresponding energy rate is a consequent, will be displayed. The way the rules are rendered is a simple HTML table, where rules are presented in rows. The first column of the table lists the antecedents of a rule, the second column displays the consequent of a rule and the third column displays the confidence score of a rule. Figure 20

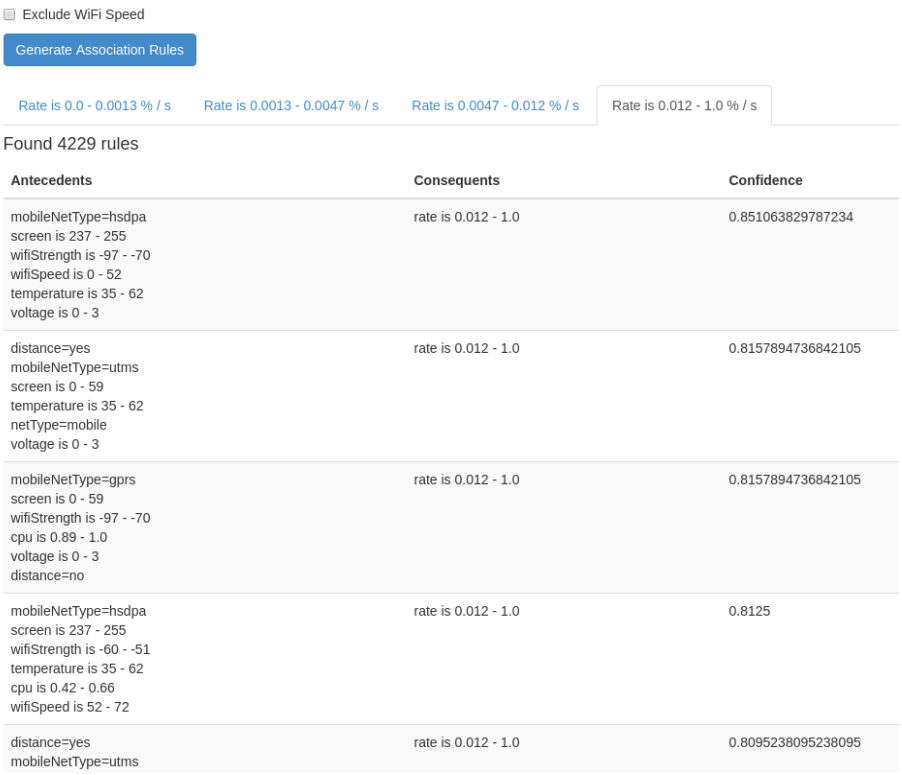



Figure 20: Example of generated association rules for the Facebook mobile application. A minimum support threshold of 0.0001 and a confidence threshold of 0.75 was used.

shows an example of generated rules for the com.facebook.katana mobile application using a minimum support threshold of 0.0001 and a minimum confidence threshold of 0.75. The displayed rules are sorted in an ascending order of confidence to display the most significant rules first.

## 6.2 Service Back End

The service back end is a simple web server implemented in Scala programming language using Scalatra web framework version 2.4.0. The webserver listens for HTTP GET requests, accepting application name, minimum support, minimum confidence and a list variable names to be excluded in the requests URL parameters. Once a valid GET request is received, the server creates a Spark job script based on the request parameters, and submits the Spark job to the analysis engine. The shell script which submits the Spark job is executed in a thread pool asynchronously to avoid making the web server unresponsive while a Spark job is running. Once the analysis engine has successfully executed the job, the shell script prints the association rules

in JSON format to its standard output stream. This output is captured and returned as a HTTP response to the client.

The server consists of merely three components, a servlet, a service and a bootstrap component. The bootstrap components purpose in Scalatra is to mount services to certain URL paths. The bootstrap component simply mounts our servlet to match with any path, as denoted by asterisk character

```
class ScalatraBootstrap extends LifeCycle {
  override def init(context: ServletContext) {
    context.mount(new MainServlet, "/*")
  }
}
```

A servlet is a server component that routes incoming requests to associated controller routines. The servlet in question is implemented is Scalatra as follows

```
class MainServlet extends ScalatraServlet with
   FutureSupport with JacksonJsonSupport {

  val conf = ConfigFactory.load()
  override val asyncTimeout = conf.getInt("
     timeout") seconds
  protected implicit lazy val jsonFormats:
     Formats = DefaultFormats
  implicit val executor = ExecutionContext.global

  before() {
    contentType = formats("json")
  }

  get("/") {
    val applicationName = Try(params("
       applicationName")).toOption
    val minSupport = Try(params("minSupport").
       toDouble).toOption
    val minConfidence = Try(params("minConfidence
       ").toDouble).toOption
    val excluded = Try(params("excluded")).
       toOption.getOrElse("")

    applicationName.map { applicationName =>
      SparkRunner.runSpark(
        applicationName,
        minSupport = minSupport,
        minConfidence = minConfidence,
        excluded = excluded
      )
    }.getOrElse {
      BadRequest(reason = "Missing '
         applicationName'")
    }
  }
}
```

The traits *FutureSupport* and *JacksonJsonSupport* indicate, that the response is computed asynchronously and is JSON formatted. The method call to *get("/")* registers a controller routine to the root path of the servlet. The controller routine, which is given as a call-by-name function to the second parameter list of the *get* call, parses URL parameters from the incoming HTTP request and submits a new task to the *SparkRunner* service. In case the


Figure 21: Overview of the analysis engine pipeline

*applicationName* is missing from the request, a *BadRequest* HTTP response is produced, otherwise the service method *runSpark* is executed asynchronously and a response will be sent once the analysis engine has accomplished its analysis. For example, suppose that the server is running on localhost and listening on port 8888. Upon receiving a GET request to an URI such as *localhost:8888?applicationName=com.facebook.katana&minConfidence=0.9*, the servlet would route the request to the controller routine configured under *get("/")*. The routine would parse the URI parameters *applicationName* and *minConfidence* and dispatch a Spark job to the analysis engine by invoking *SparkRunner.runSpark*.

## 6.3 Analysis Engine

The primary function of the analysis engine is to generate association rules from the Carat data based on provided query parameters. The query parameters consist of application name, minimum confidence, minimum support and an optional list of attribute names which are to be excluded from the analysis. The application name is used to filter out all Carat energy rate samples in which the the application is not present. The minimum support and minimum confidence parameters affect the association rule generation as described in Chapter 4. The excluded attributes list controls the association rule generation by completely ignoring all included attributes.

Figure 21 describes the process of generating association rules as a simple pipeline consisting of four steps. We will now go through each step providing snippets of code, taken from the analysis engine implementation, that will shed light on the implementation in Spark programming framework.

Reading Carat samples is very simple in Spark, as is evident from the following code snippet.

```
1 def readCaratRates(sampleDir: String)(implicit sc
     : SparkContext): RDD[fi.helsinki.cs.nodes.
     carat.sample.Rate] = {
2   sc.objectFile[fi.helsinki.cs.nodes.carat.
       sample.Rate](s"${sampleDir}")
3 }
```

The *objectFile* method of the *SparkContext* object will read the dataset from a given directory. The dataset is stored as an RDD (Resilient Distributed Dataset), that is serialized to the disk in a folder given by the *sampleDir* parameter. RDD is is the data structure that Spark framework uses to

store, access and transform distributed datasets. The Carat samples are initially read as instances of class *fi.helsinki.cs.nodes.carat.sample.Rate*. Each *Rate* object contains two consecutive samples from a mobile device. From these samples, the system settings and running mobile applications can be extracted as described in Chapter 3

The next task that the analysis engine carries out, is to filter out all Carat rate samples where the requested application was not running. Using the *readCaratRates* method, one can compose an expression which reads Carat rate data objects, filters out all rate objects that do not have the requested application running and transforms the resulting rate objects to a simplified object type of class *Sample*.

```
1 val samples = readCaratRates(ratePath).collect {
2     case rate if rate.allApps().contains(
          applicationName) =>
3         Sample.fromCaratRate(rate)
4 }
```

*collect* is a method defined for all instances of class $RDD[T]$ (where $T$ is a type parameter). It is analogous to the *collect* method from Scala standard library, taking a partial function of signature $PartialFunction[T, U]$ (where $T$ and $U$ are type parameters) and returning a new instance of $RDD[U]$, containing the image of the $RDD[T]$ mapped by the partial function.

The *Sample* is a simple case class that merely stores all the relevant system settings. The case class also has a companion object, in which the method *fromCaratRate* is defined. The method simply constructs a *Sample* instance from a *Rate* instance.

```
1  case class Sample(
2      rate: Double,
3      cpu: Double,
4      distance: Double,
5      temp: Double,
6      voltage: Double,
7      screen: Double,
8      mobileNetwork: String,
9      network: String,
10     wifiStrength: Double,
11     wifiSpeed: Double
12 )
```

The next step in the analysis work flow, is to discretize the variables of the data. All numerical variables were discretized to bins of equal mass, as explained in depth in Chapter 3. The following code snippet shows how to find the break points of the bins for continuously valued variables.

```
def getQuantiles(
  data: RDD[Double],
  buckets: Int,
  relativeError: Double = 0.0001,
  partial:
    PartialFunction[Double, Option[String]] = Map
       .empty)
  (implicit sqlContext: SQLContext):
Array[Double] = {

  import sqlContext.implicits._

  val percentiles = (for(i <- 1 to (buckets - 1))
      yield (1.0 / buckets) * i).toArray
  val notDefined = data.filter(x => !partial.
     isDefinedAt(x))

  try {
    notDefined.toDF("col").stat.approxQuantile("
       col", percentiles, relativeError)
  } catch{
    case ex: java.util.NoSuchElementException =>
       Array[Double]()
  }
}
```

The method *getQuantiles* takes as an its input *data*, an *RDD* containing the values to be discretized; *buckets*, the number of bins to create; *relativeError*, the maximum relative error that is allowed when approximating the break points of the percentiles; *partial*, a partial function that is used to filter out values that should not be taken into account when approximating the percentiles. The method uses Spark *DataFrame* API to approximate the percentiles. Using a *relativeError* parameter larger than 0, makes the generation of the association rules non deterministic, since the approximated percentile break points are allowed to vary from the exact value. However, calculating exact values for the breakpoints (by setting the *relativeError* parameter to zero) slows down the generation of the association rules considerably, as all of the data needs to be processed in order to calculate the break points as opposed to calculating the break points from a sampled dataset.

Having computed the quantiles of a continuously valued variable, discretization can be achieved by using the following method

```
def getFeatureFromQuantiles(
    dataPoint: Double,
    featureName: String,
    quantiles: Array[Double],
    partial: PartialFunction[Double, Option[
       String]] = Map.empty
): Option[String] = {
    if(partial.isDefinedAt(dataPoint))
        return partial(dataPoint).map { x =>
            s"${featureName}=${x}"
        }

    var index = quantiles.indexWhere(q => q >=
       dataPoint) + 1
    if (index == 0) index += (quantiles.length +
       1)
    Some(s"${featureName}=q${index}")
}
```

The method *getFeaturesFromQuantiles* takes as its input *dataPoint*, a single point of data to be discretized; *featureName*, the name of the feature; *quantiles*, the quantile break points for the variable and *partial* a partial function that is used both to filter out invalid data as well as to bypass the discretization altogether.

As a concrete example, let us examine how one could go about discretizing variable *screen*, which gives the screen brightness of the mobile device. As discussed in Chapter 3.6, the variable takes values between -1 and 255, where value -1 signifies a special case, where the screen brightness is automatically adapted to the brightness of the surroundings of the device. One could encode these preconditions to a partial function of the following form:

```
val screenPartial: PartialFunction[Double, Option[
   String]] = {
    case x if x == -1 => Some("auto")
    case x if x < -1 => None
    case x if x > 255 => None
}
```

Using this partial function in conjunction with the methods *getQuantiles* and *getFeatureFromQuantiles* for each data point will give the discretized form of the variable *screen*.

To summarise: in order to discretize a variable such as *screen*, which is assumed to be a collection of type *RDD[Double]*, one could fist calculate the quantiles of data using the method *getFeatureFromQuantiles*. One must then define a partial function, such as the one mentioned above, that filters out

invalid data points and handles values with special significance. Finally, the method *getFeatureFromQuantiles* could be applied to each data point of the collection using the partial function and the quantiles.

Having discretized all the variables of the dataset, using the procedure discussed above, one ends up with one array of strings for each sample, which is represented in Spark by type *RDD[Array[String]]*. To generate the association rules from these discrete features, MLlib, a machine learning library for Spark was used. The library implements a parallel FPGrowth algorithm [13] for this purpose. The FPGrowth implementation has a limitation however, in that it can only generate rules with single consequent. The limitation is not terribly severe for the purpose of this thesis, as the main interest lies in finding out which variables affect the battery usage. Therefore being limited to rules which have as their sole consequent a feature extracted from the energy rate variable, should be sufficient. The following snippet of code illustrates how to generate association rules using the MLlib API.

```
1  val fpg = new FPGrowth()
2            .setMinSupport(minSupport)
3
4  val model = fpg.run(features)
5
6  val rulesFiltered = model.
     generateAssociationRules(minConfidence)
7    .filter { rule =>
8      rule.consequent.find { item =>
9        item.startsWith("rate=")
10     }.isDefined
11   }
```

The final stage of association rule generation workflow is filtering out redundant rules. To demonstrate what is meant by redundancy in this context, let us consider two rules such as

$$\{A, B\} \Rightarrow \{X\} \tag{1}$$

$$\{A, B, C\} \Rightarrow \{X\} \tag{2}$$

Additionally, let us assume that both rules (1) and (2) have equivalent confidence. Since both rules predict the same consequents and the antecedents of rule (1) is a subset of the antecedents of rule (2), one can conclude that the rule (1) is the more general of the two rules, as adding item $C$ to the antecedents of rule (1) does not improve the associative power of the original rule. Since the objective of this thesis is to identify variables which have the

strongest association with the battery consumption of a mobile application, we consider rule (2) redundant in the context of this thesis work.

More generally, given a set $S$ of association rules, we consider rule $R \in S$ redundant if there exists a rule $r \in S$, such that $R \neq r$ and $r.antedecents \subset R.antedecents$. The following excerpt of code shows how this definition translates to a redundancy filtering algorithm in Scala programming language

```
1  def pruneRules(rules: RDD[Rule[String]]): RDD[
       Rule[String]] = {
2    val pruneCandidateGroups = rules.groupBy{ rule
       =>
3      (rule.consequent.sorted.mkString, rule.
         confidence)
4    }
5
6    pruneCandidateGroups.flatMap { case (key, group
       ) =>
7      val groupSorted = group.toSeq.sortBy(rule =>
         rule.consequent.length)
8      var groupAsSets = group.map(rule => (rule,
         rule.antecedent.toSet))
9
10     val toPrune: Set[Rule[String]] = (for {
11       testRule <- groupAsSets
12       otherRule <- groupAsSets
13       if testRule != otherRule && testRule._2.
           subsetOf(otherRule._2)
14     } yield(otherRule._1)).toSet
15
16     groupSorted.filter(rule => !toPrune.contains(
         rule))
17   }
18 }
```

Method *pruneRules* takes a collection of association rules and returns a collection of association rules which contains no redundant rules.

This concludes the analysis engine part of the implementation. From usability perspective, generating the association rules should be fast enough not to hinder interactive use, which implies that the rule generation should not take more than few seconds. However, heavily optimizing the analysis engine falls outside the scope of this thesis. In practice, the implementation described here achieves run times of a couple of minutes with a rather modest dataset size of 16 gigabytes.

# 7 Results

We will now look at the results of this work in two parts. In the first part we will be looking at the performance of the application and how it affects its usability. In the second part we will take a closer look at some examples of generated rules. The last part of this chapter focuses on the impact of these results and suggests ways to improve on this work.

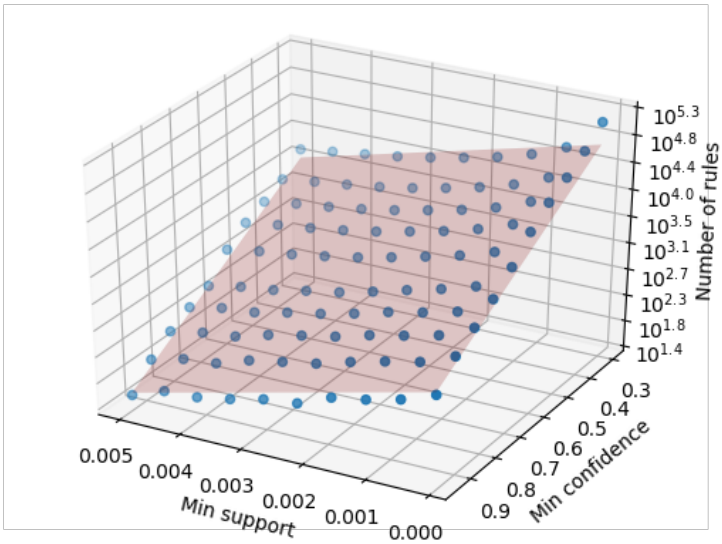
## 7.1 Performance Evaluation

In order to understand the relationship between the number of generated rules and minimum support and confidence thresholds, a series of measurements were conducted on the Carat API prototype server. Figure 22 shows these measurements for Facebook application and Figure 23 shows the measurements for Spotify application. The figures show the relationship of generated rules as a function of minimum support threshold and confidence threshold as a three dimensional plot. Five series of measurements were conducted for each application. In each series, a minimum confidence threshold range of 0.3 to 0.9 and a minimum support threshold range of 0.0001 to 0.005 were both divided evenly by 10 points creating a grid of 100 data points where the measurements were taken. The blue dots represent average of the five measurements at each point of the support-confidence -grid. In sub figure A, a plane was fitted to the measurement points using least squares method. This was done to better illustrate the spatial configuration of the measurements as well as to showcase how well the measured points are aligned on the plane. In sub figure B, error bars were plotted to the measurements using one standard deviation of the five measurements as the size of the error.

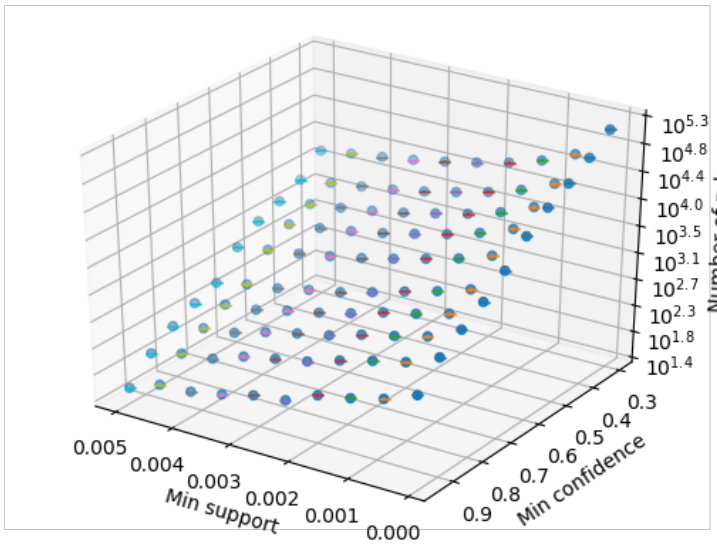
The number generated rules seems to grow exponentially on both axes when approaching zero, as can be seen by how well the measurements align with the least squares plane. This explosion in the number of generated rules makes it difficult for the user to extract useful rules from the system when small values for the thresholds are used. To mitigate this problem, the system provides two features:

- The generated rules are sorted in the ascending order of their confidence, giving the more reliable rules a greater priority.
- Attributes can be excluded from the analysis - potentially greatly reducing the number of generated rules.

Even though there is a stochastic component in the rule generation, which arises from the sampling of data in the variable discretization stage of the analysis, the number of generated rules does not seem to vary much, as can be seen from the error bars, which are barely visible.

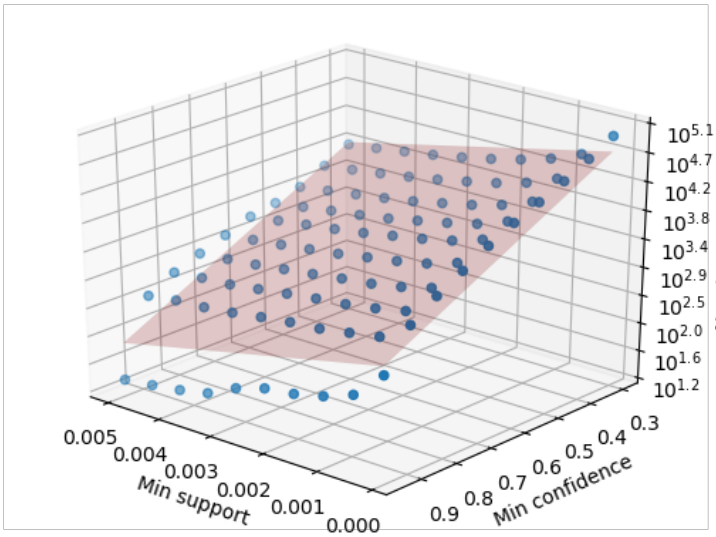



(a) Number of rules with fitted plane

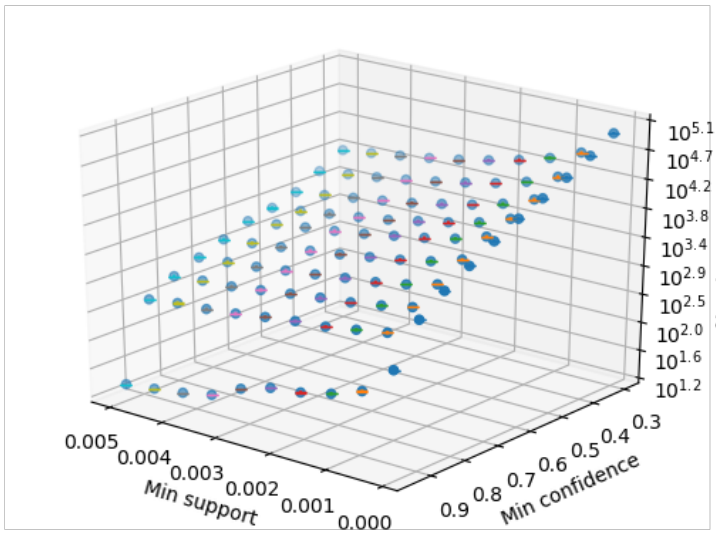



(b) Number of rules with error bars

Figure 22: Number of generated rules for Facebook measurements as a function of minimum support threshold and minimum confidence threshold

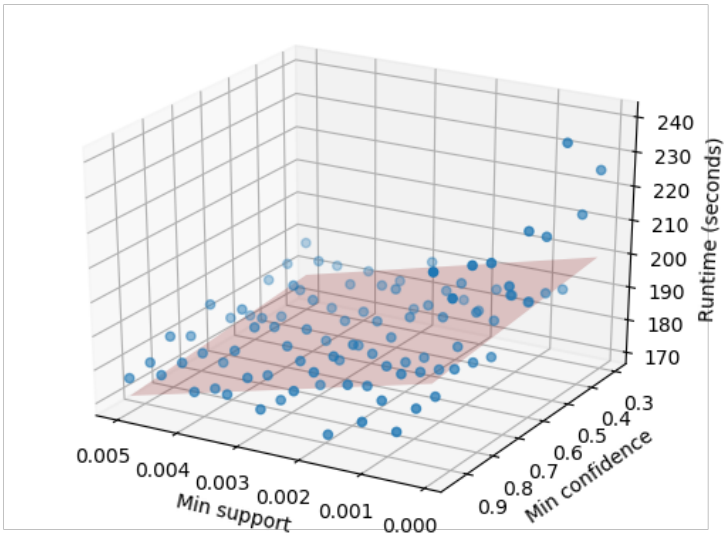



(a) Number of rules with fitted plane

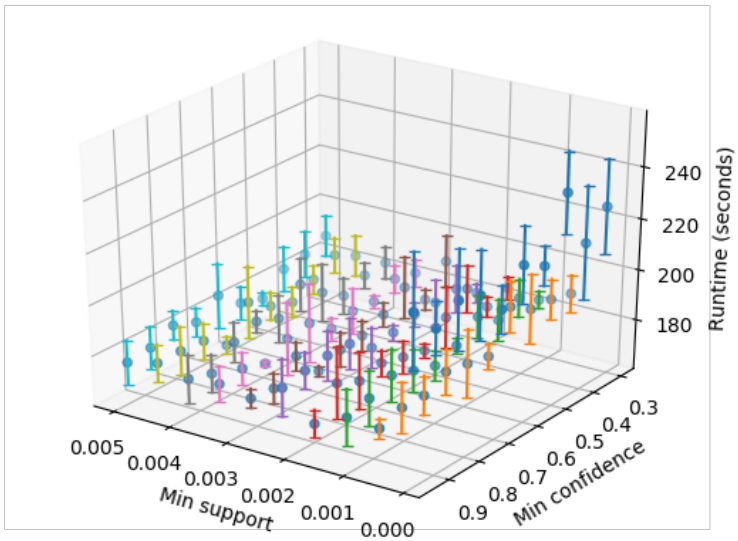



(b) Number of rules with error bars

Figure 23: Number of generated rules for Spotify measurements as a function of minimum support threshold and minimum confidence threshold




(a) Rule generation time with best fitting plane




(b) Rule generation time with error bars

Figure 24: Rule generation time for Facebook measurements as a function of minimum support threshold and minimum confidence threshold.

In addition to the number of generated rules, another metric that is a good indicator for usability of the system, is the time taken to generate the association rules. To measure the time of the rule generation as a function of minimum support threshold and minimum confidence threshold, a similar set up as with the number of generated rules was used. Figure 24 shows these measurements for the Facebook application and Figure 25 shows the measurements for the Spotify application. Like before, the blue dots represent the average value in five measurements series of 100 measurement points. The red plane represents a plane that was fitted to the points using the least squares method. The size of the error in the error bars is again the standard deviation of the measurement at each measurement point.

The rule generation time increases as either axis approaches zero. The deviance is not huge however, as all the measured run times fall between 160 and 260 seconds. While this is a notable difference from the users perspective, the system remains usable even when the number of generated rules is in the order of $10^5$.

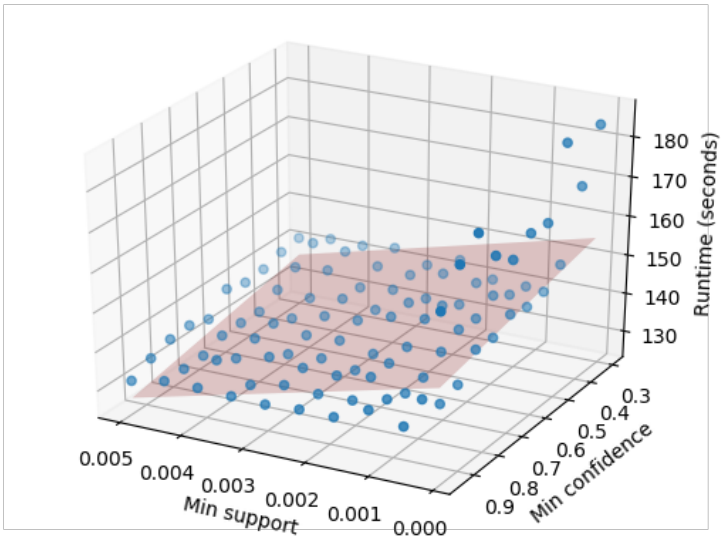
All the experiments were conducted on a Spark cluster which had a single computing server. For each run, 45 CPU cores and 1500 gigabytes of memory were reserved, although a much smaller amount of memory would have been sufficient. To mitigate the effect of any potential file server load, the dataset was stored in memory using Linux shared memory file system (/dev/shm). The dataset consisted of Carat samples from 22.6.2016 to 22.8.2016, the size of which was a little over 16 gigabytes.

## 7.2 Overview on Generated Rules

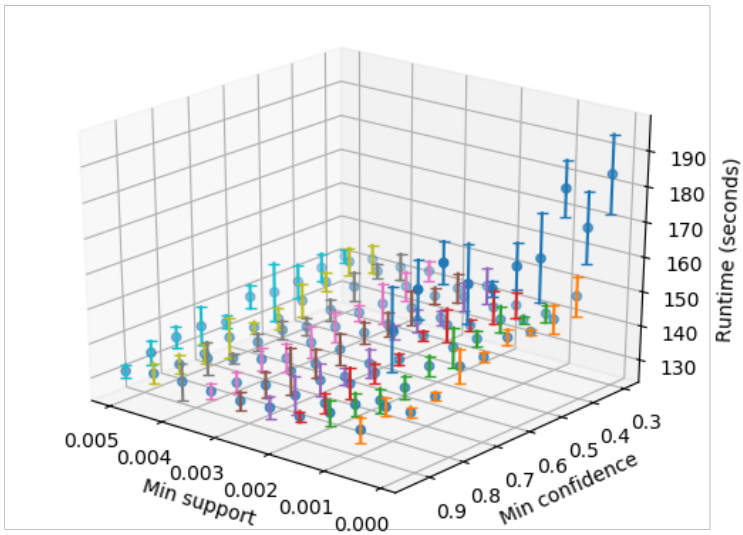
For this section, four popular Android applications were selected to be inspected with the rule generation. The selected applications were

- com.facebook.katana
- com.google.android.chrome
- com.google.android.apps.photos
- com.spotify.music

To achieve comparable results, the minimum support threshold was fixed to 0.001 and the minimum confidence threshold was fixed to 0.5. Given that the least popular of these applications in the dataset used for this analysis contained little over 420,000 data points, a minimum support threshold of 0.001 means that for each generated rule, there should always be at least 420 data points supporting that rule. For each of these four applications, a total of six of the generated rules were selected for further inspection. These six rules were selected by taking the three most confident rules that




(a) Rule generation time with best fitting plane




(b) Rule generation time with error bars

Figure 25: Rule generation time for Spotify measurements as a function of minimum support threshold and minimum confidence threshold

predicted high energy consumption rate and the three most confident rules that predicted low energy consumption rate.

Table 6 shows the generated rules for the com.facebook.katana mobile application. Looking at the top three rules that predicted hight energy rate for the application, it seems that using HSDPA type mobile network connection and a high screen brightness were clearly connected to high energy consumption. When these two factors were combined with high battery temperature, we get the most confident of these three rules with a confidence score of 0.5811. Slow network connections have been shown to be linked with increased energy consumption [18]. Since HSDPA is a 3G technology, it is reasonable to believe that its usage might contribute to increased energy consumption. As the highest confidence rule in this category had a confidence score of less than 0.6, one can say that there were no clear explanations of high energy consumption to be found by these variables for this particular application. Looking at the three most confident rules for low energy consumption, the common factors seem to be using WiFi connection, low battery temperature and low or medium low CPU usage. Notably, the rules for low energy consumption had significantly greater confidence than the rules for high energy consumption. The highest confidence for these three rules was 0.9985 while the lowest was 0.9964.

Table 7 shows the selected rules for the com.google.android.apps.photos application. In this case, the factors that indicated high energy consumption were using GPRS for mobile networking, having weak WiFi singnal strength and low WiFi signal speed and low battery voltage. The low battery voltage may indicate a certain set of mobile device models that perform poorly when the other factors are also present. The confidence of these rules were reasonable, ranging from 0.7100 to 0.7074. The most confident rules for low energy consumption all had the common factors of using UTMS mobile network connection, high WiFi link speed, medium low CPU usage and weirdly enough, high screen brightness.

Table 8 shows the selected rules for the com.android.chrome mobile application. There were no rules, within the given confidence constraint, that predicted an energy consumption rate in the highest quantile, so instead the three top rules in the table are the three highest confidence rules that predicted an energy consumption rate in the third quantile within the samples where the Chrome application was running. Within the rules which predicted high energy consumption, common factors were high screen brightness, high battery temperature, fast WiFi link speed and using LTE type mobile networking. The confidence of these rules ranged from 0.6519 to 0.65. Looking at the rules which predicted low energy consumption, common factors were low battery temperature, fast WiFi link speed, mobile networking type UTMS and again, oddly enough, a high screen brightness. The confidence of the rules predicting low energy consumption were again very high, ranging from 0.9985 to 0.9964.

| Antecedents | Consequent | Confidence |
|---|---|---|
| mobileNetType=hsdpa<br>screen is 222 - 255<br>temperature is 35 - 88 | rate is 0.011 - 1.0 | 0.5811 |
| mobileNetType=hsdpa<br>screen is 222 - 255<br>cpu is 0.39 - 0.62<br>voltage is 0 - 3<br>distance=no | rate is 0.011 - 1.0 | 0.5464 |
| mobileNetType=hsdpa<br>screen is 222 - 255<br>cpu is 0.39 - 0.62<br>voltage is 0 - 3 | rate is 0.011 - 1.0 | 0.5432 |
| mobileNetType=unknown<br>screen is 222 - 255<br>wifiStrength is -99 - -69<br>cpu is 0.0 - 0.39<br>temperature is 5 - 28<br>voltage is 3 - 4 | rate is 0.0 - 0.00017 | 0.9985 |
| mobileNetType=unknown<br>screen is 222 - 255<br>wifiStrength is -99 - -69<br>wifiSpeed is 54 - 72<br>cpu is 0.39 - 0.62<br>temperature is 5 - 28<br>voltage is 0 - 3 | rate is 0.0 - 0.00017 | 0.9974 |
| mobileNetType=unknown<br>screen is 222 - 255<br>wifiSpeed is 0 - 54<br>cpu is 0.39 - 0.62<br>temperature is 5 - 28<br>voltage is 3 - 4 | rate is 0.0 - 0.00017 | 0.9964 |

Table 6: Selected rules for the com.facebook.katana mobile application

| Antecedents | Consequent | Confidence |
|---|---|---|
| mobileNetType=gprs<br>wifiStrength is -100 - -68<br>wifiSpeed is 0 - 54<br>netType=wifi<br>voltage is 0 - 3<br>distance=no | rate is 0.011 - 1.0 | 0.7100 |
| mobileNetType=gprs<br>wifiStrength is -100 - -68<br>wifiSpeed is 0 - 54<br>voltage is 0 - 3<br>distance=no | rate is 0.011 - 1.0 | 0.7082 |
| mobileNetType=gprs<br>wifiStrength is -100 - -68<br>wifiSpeed is 0 - 54<br>netType=wifi<br>voltage is 0 - 3 | rate is 0.011 - 1.0 | 0.7074 |
| screen is 220 - 255<br>mobileNetType=utms<br>wifiSpeed is 144 - 6477<br>wifiStrength is -68 - -59<br>cpu is 0.42 - 0.67 | rate is 0.0 - 0.0015 | 0.9966 |
| screen is 220 - 255<br>mobileNetType=utms<br>wifiSpeed is 144 - 6477<br>wifiStrength is -68 - -59<br>cpu is 0.42 - 0.67<br>voltage is 3 - 4 | rate is 0.0 - 0.0015 | 0.9965 |
| screen is 220 - 255<br>mobileNetType=utms<br>wifiSpeed is 144 - 6477<br>cpu is 0.42 - 0.67<br>voltage is 3 - 4 | rate is 0.0 - 0.0015 | 0.9936 |

Table 7: Selected rules for the com.google.android.apps.photos mobile application

| Antecedents | Consequent | Confidence |
|---|---|---|
| screen is 206 - 255<br>wifiSpeed is 135 - 4728<br>temperature is 34 - 88<br>mobileNetType=lte<br>netType=wifi | rate is 0.0046 - 0.011 | 0.6519 |
| screen is 206 - 255<br>wifiSpeed is 135 - 4728<br>temperature is 34 - 88<br>mobileNetType=lte | rate is 0.0046 - 0.011 | 0.6506 |
| screen is 206 - 255<br>wifiSpeed is 135 - 4728<br>temperature is 34 - 88<br>mobileNetType=lte<br>netType=wifi<br>distance=no | rate is 0.0046 - 0.011 | 0.65 |
| screen is 206 - 255<br>mobileNetType=utms<br>wifiSpeed is 135 - 4728<br>wifiStrength is -68 - -59<br>voltage is 3 - 4 | rate is 0.0 - 0.0015 | 0.9893 |
| screen is 206 - 255<br>mobileNetType=utms<br>wifiSpeed is 135 - 4728<br>temperature is 5 - 28<br>voltage is 3 - 4 | rate is 0.0 - 0.0015 | 0.9822 |
| screen is 206 - 255<br>mobileNetType=utms<br>wifiSpeed is 135 - 4728<br>temperature is 5 - 28 | rate is 0.0 - 0.0015 | 0.9749 |

Table 8: Selected rules for the com.google.android.chrome mobile application

In Table 9 are listed the selected rules for the com.spotify.music mobile application. In the context of the rules that predict high energy consumption, factors high screen brightness, high WiFi link speed, quite low WiFi signal strength, medium high battery temperature and medium high CPU usage are all shared. Two out of three of these rules also share the factor mobile networking type LTE and low battery voltage. The confidence of these rules ranged from 0.8852 to 0.8821, which compared to the other application's high energy rules, seems very high. Among the rules that predicted low energy consumption, shared factors were mobile networking type UTMS, a medium high CPU usage, and a medium low wiFi signal strength. Factors that were shared by two out of the three rules included high screen brightness, low battery temperature and high WiFi link speed. The confidence score of all three of these rules was 1.0.

The generated example rules are generally not very intuitive and some of the relations, such as the contribution of high screen brightness to low energy consumption, are outright counter intuitive. On the bright side, the system is able to find rules with very high confidence even with a reasonably low support threshold of 0.001. The choice of application also seems to have a reasonable impact on the generated rules, which is promising for the usability of the system. Perhaps interesting is the fact that more confident rules seem to be generated for the low energy consumption than for the high energy consumption. Even if the system is not able to predict with any reasonable accuracy which factors lead to large levels of energy consumption when using a certain application, it might be useful to be able predict with acceptable accuracy which combinations of factors lead to low levels of energy consumption. At the very least, this kind of prediction could be a valuable addition to a recommendation system like the one described by Peltonen et al. [20]

| Antecedents | Consequent | Confidence |
|---|---|---|
| screen is 210 - 255<br>wifiSpeed is 144 - 866<br>wifiStrength is -68 - -59<br>temperature is 34 - 60<br>cpu is 0.61 - 0.84<br>mobileNetType=lte<br>voltage is 0 - 3 | rate is 0.012 - 1.0 | 0.8852 |
| screen is 210 - 255<br>wifiSpeed is 144 - 866<br>wifiStrength is -68 - -59<br>temperature is 34 - 60<br>cpu is 0.61 - 0.84<br>voltage is 0 - 3 | rate is 0.012 - 1.0 | 0.8821 |
| screen is 210 - 255<br>wifiSpeed is 144 - 866<br>wifiStrength is -68 - -59<br>temperature is 34 - 60<br>cpu is 0.61 - 0.84<br>mobileNetType=lte | rate is 0.012 - 1.0 | 0.8821 |
| mobileNetType=utms<br>wifiSpeed is 144 - 866<br>wifiStrength is -68 - -59<br>cpu is 0.61 - 0.84<br>temperature is 12 - 28<br>voltage is 3 - 4 | rate is 0.0 - 0.0016 | 1.0 |
| screen is 210 - 255<br>mobileNetType=utms<br>wifiStrength is -68 - -59<br>cpu is 0.61 - 0.84<br>temperature is 12 - 28 | rate is 0.0 - 0.0016 | 1.0 |
| screen is 210 - 255<br>mobileNetType=utms<br>wifiSpeed is 144 - 866<br>wifiStrength is -68 - -59<br>cpu is 0.61 - 0.84 | rate is 0.0 - 0.0016 | 1.0 |

Table 9: Selected rules for the com.spotify.music mobile application

## 7.3 Discussion

This thesis work has presented a method for generating association rules from Carat dataset in order to estimate how mobile device system settings and context factors impact the level of energy consumption of a mobile device when using a particular mobile application. These association rules reveal non-trivial and perhaps unexpected connections between these settings and context factor and the level of energy consumption within the context of multiple mobile applications. For some reason, the generated association rules that predict low levels of energy consumption, seem to have much higher confidence than the rules which predict high levels energy consumption. This may be due to various reasons. One reason might be that while the association analysis seems to be able to capture at least some circumstances which consistently lead to low energy consumption, the system settings and context variables available within the Carat dataset are inadequate for explaining unusually high energy consumption levels. It could even be that the users whose devices have high energy consumption are generally running multiple mobile applications at the same time, which would naturally generate more noise to data points coming from those users. One could potentially test this hypothesis by adding the number of running applications to the list of variables from which the association rules are generated from. If this was the case, then one would expect to see rules where high number of running applications predicts high energy consumption.

Another goal of thesis work was to implement a web based interface, so that users could search these association rules easily. The implementation has two web servers that communicate to one another using a simple HTTP based API. The back end of the service resides on a Spark cluster where it can execute the analysis engine on user supplied parameters as requested. This way the data analysis can be wrapped inside a single exchange of HTTP request and response. The front end of the service handles all things related to the graphical user interface: rendering the search form, fetching the rules from the back end and rendering the results. The front end of the service can reside wherever as long as the service back end can be reached by HTTP. This two-tier architecture allows the remote use of computational resources of a Spark cluster without exposing the Spark cluster environment to potential security vulnerabilities that a globally accessible web server might impose.

This API could potentially be used by developers to diagnose which factors affect the energy consumption of their application's users. This information could be useful for optimizing the application for scenarios where the application is likely to consume unusually large amounts of energy. The information could even reveal bugs which cause anomalous energy consumption in certain conditions. Yet another use could be simply to give a coarse-grained estimate of the energy consumption rate of the application.

The implementations of both the data analysis and the user interface

could be further improved. First of all, due to performance reasons, the data set had to be limited to around 16 GB, which is more than an order of magnitude less than the whole amount of available data. It is quite possible that the association analysis might reveal more fine grained dependencies between the context factors and system settings and the energy consumption of a device, if the analysis was performed using more of the available data. Different discretization strategies for the data might also affect the generated rules. Discretization of most numerical variables was done using quite an arbitrary number of percentiles, namely four. The implementation could easily be extended to allow the user to specify the number percentiles used in the discretization.

The user interface could be improved in multiple ways. The user interface does not show the units of measurement nor does it show values of the break points of the percentiles, which could give the user a clearer sense of how a certain value range of a variable compares to the average value of the variable. Rendering of the rules could also be improved. Paging of the rules would be a useful feature to implement because browsing through as many as thousands of rules in a single page is cumbersome. The user could also benefit from a searching and filtering functionality in the front end of the service to be able quickly find the rules that the user considers interesting.

# 8 Conclusion

This thesis work has shown that the association analysis can effectively be applied to the domain of mobile device energy consumption modelling. Additionally, an implementation of a web based prototype for a developer API for the Carat dataset has been presented. The current state of the research of mobile device energy modelling as well as the relevant parts of the theory of association analysis have been reviewed.

The Carat data consists of samples which are collected from users of the Carat mobile application for the purpose collaborative energy modelling of mobile devices. Each sample contains a list of currently running mobile applications, energy consumption rate, CPU usage, travel distance, battery temperature, battery voltage, screen brightness, used mobile network technology, type of network, WiFi signal strength, and WiFi connection speed of the mobile device. This thesis has described in detail each of these variables as well as the discretization and preprocessing of the data that must be performed in order to make the association rule discovery applicable. In summary, the variables have been divided to either three or four bins of equal mass. Some assumptions about the feasible range of the variables have been applied in preprocessing stage to exclude potentially corrupted data points.

This work has presented the essential theoretical background of the association analysis. It has introduced the FP-growth algorithm and the associated data structure FP-tree as a way of discovering frequent patterns from a dataset. It has also discussed how association rules can be generated from the frequent patterns without having to consider all candidate rules, giving an outline of an algorithm for pruning candidate rule search tree.

This thesis has shown how to implement a web based query engine that can be used to discover association rules based on the Carat data. The implementation has three major components, an analysis engine which handles all data analysis tasks, a back-end web server that uses the analysis engine as a service and exposes a JSON based HTTP API, and a service front-end that handles all user input and uses the API of the back-end server as a service for generating the association rules. The analysis engine has been built using Spark programming framework and specifically MLlib, a machine learning library for Spark which implements a parallel and distributed FP-growth algorithm.

The resulting association rules generated from the Carat data are somewhat promising. The analysis engine is consistently able to find high confidence rules from various different mobile applications. It seems however, that rules predicting high energy consumption are overall more rare and less confident than the rules which predict low or near average energy consumption. This thesis suggests some potential reasons as to why this may be the case. In order to evaluate the performance of the implementation, the runtime and the number generated rules have been measured as a function of the minimum

confidence and minimum support thresholds of the association analysis for two popular Android mobile applications. The evaluation shows that the runtime of the analysis clearly depends on these variables, which is to be expected. While the runtime of the analysis shows a clear decreasing trend along both the minimum confidence threshold and the minimum support threshold axes, the magnitude of the analysis runtime does not change even when small values for these variables are used. A more problematic result from the usage point of view is the number of generated rules. Similarly to the runtime, it shows a clear decreasing trend along both the minimum support threshold and the minimum confidence threshold axes. Unlike the the runtime however, the number of generated rules ranges from as low as a dozen to tens of thousands.

Further work is needed in order to improve the implementation of the search engine. The current implementation cannot handle tens of thousands of generated rules in a user friendly way. Additional methods should be considered for finding the interesting or important rules. It is also unintuitive that the user must specify the minimum support threshold and the minimum confidence threshold for the algorithm or settle with arbitrary default values. The engine should ideally be able to guess or iteratively find reasonable values for these variables based on the user preferences. Additionally, the effect of increasing or decreasing the number of discretization percentiles on the generated rules should be studied, or alternatively the user could be allowed to specify the number of percentiles for each of the variables. Further optimizing the implementation or performing experiments with more computing capacity could potentially reveal more intricate association rules.

# References


[1] *International Telecommunication Union, ICT Statistics.* `https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx`, 2018. [Online; accessed 27-March-2018].

[2] Agrawal, Rakesh, Imieliński, Tomasz, and Swami, Arun: *Mining association rules between sets of items in large databases.* SIGMOD Rec., 22(2):207–216, June 1993, ISSN 0163-5808.

[3] Agrawal, Rakesh and Srikant, Ramakrishnan: *Fast algorithms for mining association rules.* In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] Agrawal, Rakesh and Srikant, Ramakrishnan: *Fast algorithms for mining association rules in large databases.* Technical report, IBM Almaden Research Center, San Jose, California, USA, June 1994.

[5] Dean, Jeffrey and Ghemawat, Sanjay: *Mapreduce: Simplified data processing on large clusters.* Commun. ACM, 51(1):107–113, January 2008, ISSN 0001-0782.

[6] Falaki, Hossein, Mahajan, Ratul, Kandula, Srikanth, Lymberopoulos, Dimitrios, Govindan, Ramesh, and Estrin, Deborah: *Diversity in smartphone usage.* In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 179–194, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-985-5.

[7] Ferreira, Denzil, Goncalves, Jorge, Kostakos, Vassilis, Barkhuus, Louise, and Dey, Anind K.: *Contextual experience sampling of mobile application micro-usage.* In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services*, MobileHCI '14, pages 91–100, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-3004-6.

[8] Han, Jiawei, Pei, Jian, and Yin, Yiwen: *Mining frequent patterns without candidate generation.* SIGMOD Rec., 29(2):1–12, May 2000, ISSN 0163-5808.

[9] Hipp, Jochen, Güntzer, Ulrich, and Nakhaeizadeh, Gholamreza: *Algorithms for association rule mining — a general survey and comparison.* SIGKDD Explor. Newsl., 2(1):58–64, June 2000.

[10] Kalic, G., Bojic, I., and Kusek, M.: *Energy consumption in android phones when using wireless communication technologies.* In *2012 Proceedings of the 35th International Convention MIPRO*, pages 754–759, May 2012.

[11] Karabatak, Murat and Ince, M. Cevdet: *An expert system for detection of breast cancer based on association rules and neural network.* Expert Systems with Applications, 36(2, Part 2):3465 – 3469, 2009, ISSN 0957-4174.

[12] Karabatak, Murat, Ince, M. Cevdet, and Sengur, Abdulkadir: *Wavelet domain association rules for efficient texture classification.* Applied Soft Computing, 11(1):32 – 38, 2011, ISSN 1568-4946.

[13] Li, Haoyuan, Wang, Yi, Zhang, Dong, Zhang, Ming, and Chang, Edward Y.: *Pfp: Parallel fp-growth for query recommendation.* In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-093-7.

[14] Oliner, Adam J., Iyer, Anand, Lagerspetz, Eemil, Tarkoma, Sasu, and Stoica, Ion: *Collaborative energy debugging for mobile devices.* In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[15] Oliner, Adam J., Iyer, Anand P., Stoica, Ion, Lagerspetz, Eemil, and Tarkoma, Sasu: *Carat: Collaborative energy diagnosis for mobile devices.* In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2027-6.

[16] Oliver, Earl: *The challenges in large-scale smartphone user studies.* In *Proceedings of the 2Nd ACM International Workshop on Hot Topics in Planet-scale Measurement*, HotPlanet '10, pages 5:1–5:5, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0177-0.

[17] Oliver, Earl and Keshav, S.: *Data driven smartphone energy level prediction.* Technical report, University of Waterloo, Waterloo, Ontario, Canada, 2010.

[18] Peltonen, E., Lagerspetz, E., Nurmi, P., and Tarkoma, S.: *Energy modeling of system settings: A crowdsourced approach.* In *2015 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 37–45, March 2015.

[19] Peltonen, E., Lagerspetz, E., Nurmi, P., and Tarkoma, S.: *Too big to mail: On the way to publish large-scale mobile analytics data.* In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2374–2377, Dec 2016.

[20] Peltonen, Ella, Lagerspetz, Eemil, Nurmi, Petteri, and Tarkoma, Sasu: *Constella: Crowdsourced system setting recommendations for mobile*

*devices*. Pervasive and Mobile Computing, 26(Supplement C):71 – 90, 2016, ISSN 1574-1192. Thirteenth International Conference on Pervasive Computing and Communications (PerCom 2015).

[21] Perrucci, G. P., Fitzek, F. H. P., Sasso, G., Kellerer, W., and Widmer, J.: *On the impact of 2g and 3g network usage for mobile phones' battery life*. In *2009 European Wireless Conference*, pages 255–259, May 2009.

[22] Qian, H. and Andresen, D.: *Reducing mobile device energy consumption with computation offloading*. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–8, June 2015.

[23] Shye, A., Scholbrock, B., and Memik, G.: *Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures*. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 168–178, Dec 2009.

[24] Wagner, Daniel T., Rice, Andrew, and Beresford, Alastair R.: *Device analyzer: Large-scale mobile data collection*. SIGMETRICS Perform. Eval. Rev., 41(4):53–56, April 2014, ISSN 0163-5999.

[25] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion: *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.