Learning usage behavior based on app feedback

Erik Bager Beuschau

Kongens Lyngby 2012
IMM-M.Sc.-2012-85

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2012-85

Abstract

As people spend more and more time with their smartphones, it is necessary to have a solid understanding of modern smartphone users' behaviour in order to get a better indication of their needs in various situations and contexts.

In this project, an Android application has been build which continuously collects data on the user's active application. This data is sent to a server, which has also been created during the project, and it is with background in this collected data that further analysis have been conducted. Some of these analysis have already been seen in other studies, while some are new to this area.

Some of the obtained results indicate a number of significant trends within usage behaviour, and they give a vital insight into how users navigate and operate within various contexts. For instance, a lot of games are used mostly around and after midnight, social applications (e.g. Twitter and Facebook) will most often be used right before a browser application, and it has been possible to detect similar usage behaviour within specific distinguishable geographical areas for single users. It has also been shown that more than 50% of users' application usage is registered within a 500 metres radius.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The supervisors of the thesis are Jakob Eg Larsen and Michael Kai Petersen from the Department of Informatics and Mathematical Modelling at the Technical University of Denmark.

Vanløse, 6 August 2012

Erik Bager Beuschau

Acknowledgements

I would like to thank all of those who helped me collect the usage data by installing my Android application. I could not have finished this thesis without your help.

Contents

Chapter 1

Introduction

1.1 Motivation

As my current job is developing Android applications, it is very important to me how and when users utilise mobile applications. Smartphone users tend to use a lot of different applications during a day, but knowledge about users' usage patterns is limited. By gaining knowledge on user behaviours, both application developers and users can benefit. Application developers, as they will have a better understanding of what the user needs in various situations, and users, as they will get more clever applications which adapt to their specific needs and the environment they are in. Furthermore, it is relevant to see if users tend to have specific usage patterns related to their respective locations, as locations often reflect the contexts users are in.

1.2 Project Goals

The overall goal of the project is to get a better understanding of mobile users' usage patterns. That is, investigating the relationship between application usage and users' locations, examine transition patterns between applications, and look into application usage over time.

In order to achieve this, a number of sub-goals need to be accomplished. First of all a mobile application prototype must be build, which continuously registers the active application on an Android device. It must upload this data to a server, which will store the data along with each user's ID. This server must also be created, and should be available for clients to connect to at all times. This server should also provide individual application recommendations to each user, based on their own and other users' application usage behaviour, on a user's request. A recommendation, in this context, is considered as a relevant suggestion of a new application which the user has not currently installed on his/her device.

1.3 Potential

Potentially, there is a great number of areas where this kind of knowledge could be interesting. First of all, access to the data set, and the conclusions made on background of it, might be valuable to companies developing mobile applications. Both for developers who have already launched their application, but also for developers who are in the process of building an application. Some of the possibilities for developers are described below.

Advertising – The information could be used for putting intelligent advertisements in mobile applications, based on a user's location or context or based on the time of the day[1]. This could turn out to be quite profitable, and ads could be sold in real-time to the highest bidder; this is already seen with Google Ad Auction[2]. The ads could include links to other applications which were in the same category as either the user's active application or one of the user's most popular applications within a certain context.

Links in an application to other applications or features, which are likely to be launched in sequence with the given application, might give a better user experience.

Categorisation – Finding trends for applications of similar type could make developers aware of trends within specific categories. This could make it easier to adapt applications to specific user needs based on users' application usage within each application category.

[1] This kind of advertising is already seen, however not with the proposed level of detail in targeting, see for instance Google AdMob (for Android): https://developers.google.com/mobile-ads-sdk/docs/android/intermediate#targeting

[2] http://support.google.com/adsense/bin/answer.py?hl=en&answer=160525#1

By benefiting application developers, users will ultimately get better applications. But applications could also be developed specifically for helping users in various situations with background in the generated data set. A home screen application or widget[3] could, during the day, recommend different applications (perhaps only installed applications) to a user based on the time of the day/week and the user's location and context. This would provide a clever shortcut solution to the most relevant applications in each specific situation during the day or week.

An indication of the potential of this kind of knowledge is seen by the growing competition within mobile application development. Recently, a German mobile analytics firm claimed that only 1 of 3 published applications on Apple's App Store is ever used (downloaded or rated).[4] This explains why multiple companies (to name a few: Apptrace, keen.io, countly and flurry)[5] start delivering full-scale solutions with built-in analytics features for analysing how users utilise their application and for tracking every development in the number of active users and downloads in various ways.[6] This is an area where there is an explicit need for detailed information on how users utilise applications.

1.4 Contributions

A mobile application prototype, named "Appalyzer", has been developed, tested and published on Google Play[7] which continuously registers the active application on a device. The application communicates with a server which collects and stores the registered user data. The server calculates internal relationships between users, and maintains a large set of individual application recommendations which are available to users on request.

Another server has been build which runs a few web-pages where it is possible to get insights into the collected data set through different visualisations. These pages can be accessed here: http://89.233.45.28

[3] http://developer.android.com/guide/topics/appwidgets/index.html

[4] See: http://tiny.cc/pemiiw

[5] http://www.apptrace.com/, http://keen.io/, http://count.ly/ and http://www.flurry.com/

[6] Some of this might also be caused by Apple's ecosystem around their App Store, and the trend is not necessarily the same for Google Play. However, with so many available applications (around 500,000 according to http://www.appbrain.com/stats/), it is evident that the competition is intense.

[7] See: https://play.google.com/store/apps/details?id=com.dtu.appalyzer

Finally, data analysis have been carried out, with help from the web-pages described above, in three major areas:

- Locations
- Application transitions
- Application usage over time

Chapter 2

Related Work

Previous, and much larger, studies have been carried out in this area [BHS+11, Nok], where a lot of usage data has been collected. While it is hard to find a lot of information on exactly what they collected in the Nokia Data Challenge [Nok], it is more transparent in the first study [BHS+11].

What they did, was to create an application (called "Appazaar beta") much like the one proposed in this project, which continuously registered a lot of information on users and their application usage. They measured several parameters, even users' velocity when they where using different applications. They also looked at the length of a series of interactions, to see if there were any interesting tendencies related to the number of applications used in one series of application interactions (before the screen was turned off on the device). The data set they describe in their article has more than 4.000 users and has registered more than 22.000 different applications. An attempt was made to gain access to their data set, but the authors/administrators would only offer access to this if one was at their site, within their firewalls (to make sure no data was leaked publicly due to privacy issues).

The "Appazaar beta" also introduced a recommendation feature which could recommend new applications to users based on their own usage pattern. However, based on the feedback on Google Play (previously Android Market), it seems that this application has a number of challenges to overcome. It seems the

Name	Description	Running Tasks
"Appazaar beta"	Scientific application related to [BHS+11] which with an advanced approach based on application sequences and user behaviour tries to recommend different applications	Yes
"One! Best Recommendations"	recommends applications based on user feedback and Android Market analysis	No
"Appreciate"	takes a social approach and lets you see which applications your friends use	No
"Appmom"	This is not available in Denmark, and therefore has not been tested	Yes
"Appolicious"	Looks at installed applications and gives recommendations based on a huge database with information on application categories, user ratings, and applications which are considered alternatives to each other	No
"Best Apps Market"	Recommends applications based on expert reviews	No

Table 2.1: A subset of the recommendation applications available on Google Play. The running tasks column indicates whether the application uses information on a device's running tasks.

recommendations are not very good, the application is quite unstable, and the application consumes a lot of power[1]. Yet another scientific approach has been presented in [LKKK11], where an attempt is made to recommend applications in a social context using semantic relations.

Similar approaches have been made to develop applications which can recommend applications to users on Android. A comprehensive list of the most popular ones can be seen in Table 2.1. A number of other applications exist, but they all seem to use a recommendation feature based on an expert reviewer or a similar approach.

Many popular recommendation systems exist in other contexts than application

[1] https://play.google.com/store/apps/details?id=net.appazaar.appazaar

recommendation, for instance Last.fm[2], Netflix[3], Google Ads[4] and StumbleUpon[5]. Some of these systems use interesting ways of recommending items to users - some of which might be relevant to this project.

Other projects have tried analysing mobile users' behaviour in other contexts than application usage. For instance users' music listening patterns in relation to the respective situations they are in [ZHLP10]. This kind of study has also been used to try and predict and recommend music for users based on their context [PYC06].

Another study related to predicting user behaviour, has shown interesting results regarding the lack of randomness in mobile users' movement, where the predictability of the average user was more than 80% [SQBB10]. This is relevant to this project, as the distribution of locations has a big impact on the number of applicable techniques when performing location analysis.

[2]See: http://www.last.fm
[3]See: http://www.netflix.com
[4]See short description: www.google.com/adsense/
[5]See: http://www.stumbleupon.com/

Chapter 3

Analysis

With background in the Related Work chapter, the requirements for the mobile application will be outlined in this chapter. Furthermore, the recommendation feature will be analysed and finally, the background for the data analysis of the collected data will be described.

3.1 Mobile Application Analysis

The application prototype must fulfil certain requirements in order to support the possibility for thorough data analysis later. The requirements will be described below.

3.1.1 Non-functional Requirements

3.1.1.1 User Experience

Whenever doing a project which requires users to contribute, it is a good idea to see if it is possible to give the users something in return. In this project,

users' usage data is collected, and in order to make this interesting to the users they must benefit as well. If users do not gain anything from the application, it seems unlikely that they will continue to use it or have it installed for that matter.

First of all the proposed application must be able to recommend applications to the user. This will give users a useful product of their benefit. The application should also allow the user to investigate the data collected on the individual user through relevant visualisations on the user's own device. This should make the application more attractive to use, encourage users to keep the application installed on their device, and might even give users interesting and new information on their own application usage patterns.

Since users might not be familiar with the type of this application, it is important that users can get information on the purpose and usage of the application from within the application.

3.1.1.2 Efficiency

It is important that the application does not drain the user's battery by constantly processing data, looking for new locations, or uploading data to the server. Therefore, it must run in the background, only check for the location when there is a transition (in the active application), and only periodically upload data to the server. This will preserve battery, give a better user experience, and thereby make sure that users do not uninstall the application due to bad performance. More information could be collected (as they did in [BHS+11]) but this might quickly lead to heavy computations and high battery consumption. Some of these extra features might include accurate location estimations, velocity calculations and registration of application usage with extremely high frequency.

3.1.1.3 Privacy

When logging user interactions, it is important to be aware of any privacy related issues. To ensure that users' identities are kept safe, there must be no way of tracking a registered user back to an identity or device. This means that no data should be collected which is not related to the specific task at hand - this means phone number, zip code, age, date of birth etc. The only necessary information which must be transferred between client and server is a unique ID which represents each device. This is in order to register continuous interactions

related to the right person/device. Besides that the ID might be stolen during server-client communication, this seems as a sufficient[1] solution which hides the user's identity.

However, as location data is collected it might be possible to detect a specific registered user if one knows the locations where this user usually stays (e.g. where he/she works, lives etc.). This is inevitable if the relation between a bunch of location data and a specific user needs to be ensured. However, by lowering the accuracy of the locations collected, using this data to identify a user will be a lot harder. Furthermore, as more users become active, it will make it more difficult to distinguish a single user's movement pattern compared to others. This sort of de-anonymisation has been seen before [NS08], and it should be a major concern for every project (or company/organisation) collecting this kind of data to minimise this risk.

3.1.2 Functional Requirements

Following the requirements described above, the functional requirements will now be outlined. The application should be able to perform the following:

- Continuously collect data of the running application
- Periodically upload the data to a server
- Present the user with visualisations and key figures on the user's application usage
- Allow the user to receive application recommendations, and review previously received recommendations
- Allow the user to disable/enable the collection of data as well as the upload of data
- Allow the user to choose for how long the collected data should be stored
- Give the user information on what the application does and how it works

These requirements should cover the basis for the mobile application - an application which is called "Appalyzer".

[1] Sufficient for the scope of the project. To avoid that the ID is stolen during this transaction, one should use a secure connection and possibly encrypt the ID.

3.2 Recommendations

Recommendation systems exist in many different contexts, and are used for all kinds of recommendations [AEK00, PB07]. A lot of research is done on recommendation systems in order to improve their accuracy and the quality of their predictions [AT05]. Well-known types of recommendations include music-[2] and film-recommendations[3], but recommendations are also seen within all kinds of e-commerce sites and search engines.

Recommending (Android) applications is comparable to recommending films or music - based on the user's taste in applications, he/she will be presented with similar applications based on other users' reviews. The reviews/ratings of the applications will, for this project, be considered as the number of interactions with each application. The choice of recommendation approach for this project is outlined in Section 4.3.

3.3 Data Analysis

A thorough analysis of the collected data has been conducted, and can be seen in Chapter 7 on page 45. Some of the underlying reasoning for the focus of these analysis will now be presented.

3.3.1 Application Interactions

The differences in application usage first of all tells something about application popularity, but by analysing when and how much each application is used, it is possible to tell something about usage patterns during a day or a week. By accumulating application usage over time it is possible to see indications and trends occurring periodically. These analysis can be used in various ways and are able to reflect a high level of detail if presented properly.

The transitions between applications is also very relevant to investigate. This tells something about the relationships between individual applications, and allows for further analysis of applications popular as either predecessors or successors, predecessor being the application from which the successors is "launched".

[2] For instance Last.fm and iTunes Genius, see: http://www.apple.com/itunes/features/
[3] For instance Netflix and jinni, see: http://www.jinni.com/

3.3.2 Categorisation

By categorising the different applications, trends within each category can be found. This will add yet a layer of information to the collected data, and can make conclusions even more powerful by generalising them on multiple applications under the same category. On Google Play categories already exist, describing each application's type. However, these categories are quite general and might not prove detailed enough for the purpose of this project. Moreover, the application category is not chosen by an authority but selected by the developer(s) when publishing an application, leading to subjective and inconsistent categorisations.

3.3.3 Location Analysis

According to [ZHLP10], it seems that there is a correlation between users' music listening patterns and the respective context they are in (at least whether a user is static or moving). These observations indicate that users tend to use their device differently in different contexts. Assuming that mobile users use different applications in different contexts and that these contexts exist at different locations, it is interesting to analyse the collected data in this project to see how a user's location and application usage correlates.

These results can benefit application developers adapt their applications to specific situations or environments and ultimately improve the user experience. These optimisations might include built-in shortcuts to other applications, battery preserving functionalities or explicit information which might be of interest to users using an application in a certain context. A good example of an application which has already tried to provide users with one of these shortcuts is "Endomondo Sports Tracker"[4]. An illustration of this kind of shortcut can be seen in Figure 3.1. They benefit from the fact, that users of sports applications are likely to use some sort of music application simultaneously while exercising. Similarly, the social Android application Foursquare[5] utilises the built-in map functionality for visualising their different locations where people can check in (illustrated in Figure 3.2). This kind of intelligent shortcuts should be easier for developers to add to their applications with background in this project.

[4] See: https://play.google.com/store/apps/details?id=com.endomondo.android&hl=en
[5] See: https://play.google.com/store/apps/details?id=com.joelapenna.foursquared

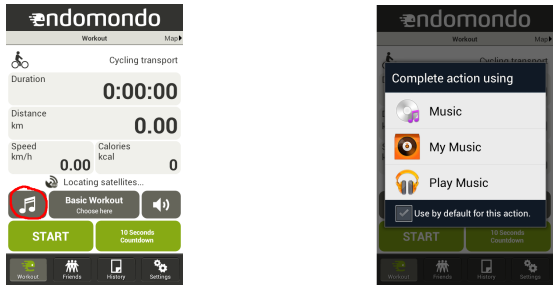


Figure 3.1: Shortcut to music applications from "Endomondo Sports Tracker"

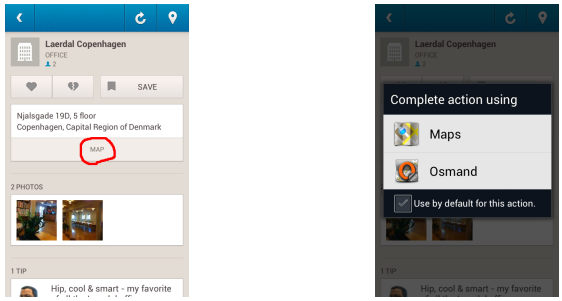


Figure 3.2: Shortcut to map applications from "Foursquare"

3.3.3.1 Location Distribution

As mentioned in Chapter 2, a study shows that mobile users' movement patterns are 80% predictable. Moreover, it seemed that users spend 70% of their time within their most popular location [SQBB10]. The study was based on a large number of users who had their location determined, with use of the mobile tower they were connected to, whenever they called someone.

By investigating the distribution within the data of this project, it is possible to see if it follows the same tendency or if the pattern is different when locations are registered more frequently (every 2 seconds instead of when a user calls someone).

Chapter 4

Design

In this chapter the different design choices for the implementation will be described, and the background for the data analysis in Chapter 6 will be outlined.

4.1 Mobile Application Development

Many of the considerations related to the mobile application development will be described below.

4.1.1 Information Architecture

The features outlined in Section 3.1.2 can be collected into three categories: Settings, Statistics, and Recommendations. This makes it obvious to choose a navigation infrastructure in the application witch contains 3 different menus, each accessible from a 4th menu which connects the navigation. A diagram of the proposed navigation can be seen in Figure 4.1. The choice on sub-menus in each of the 3 menus are based on the requirements earlier mentioned. The fact

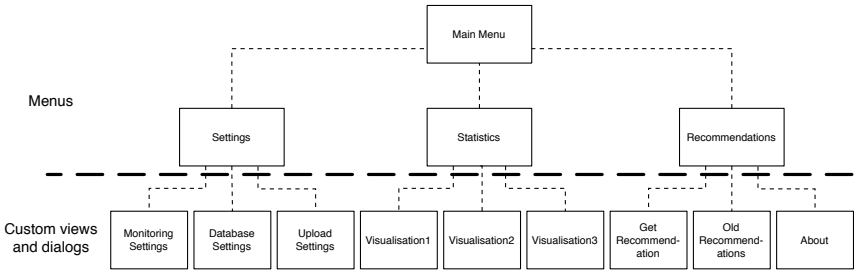


Figure 4.1: Navigation structure in "Appalyzer"

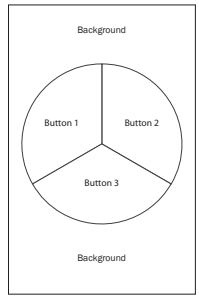


Figure 4.2: Mock-up of the proposed design.

that the "About"-sub-menu is in the "Recommendations"-menu is not necessarily intuitive, except for the fact that it explains the basis for the recommendations.[1].

The navigation set-up described above is achievable in various ways. Even though guidelines exist for implementing Android user interfaces[2], a radical interface is proposed which emphasises the fact that the application handles statistics. This is achieved by letting each menu (in Figure 4.1) consist of a pie chart-looking area in the centre of the screen, which holds the 3 relevant buttons for each menu. A mock-up of this layout can be seen in Figure 4.2

[1]The default way of presenting a menu such as an About-menu (on Android) would be putting it in an "Options Menu", see: http://developer.android.com/guide/topics/ui/menus.html#options-menu

[2]http://developer.android.com/design/index.html

4.1.2 Visualisations

In order to provide the user with some feedback on his/her application usage, some different visualisations will need to be implemented in the Android application. Based on the data collected it is obvious to create a list of the different applications, which the user has interacted with, and an indication of how much time he/she has spent on each of these. This will give a good overview of where the user spends most time, and the relations between application usage time. In this overview, it should also be possible to see which applications that are most often used as predecessors for each application. This option should be possible when clicking an application in the list.

A graphical overview of the relations mentioned above can be achieved using a pie chart. This will give the user an even better insight into his/her application usage and provide a visually comparable illustration of the application usage time.

Finally, the fact that location data is collected makes it obvious to display this information on a map. Therefore, a map visualisation should be implemented, making it possible to see where in the world each application is used the most. This could be done by adding coloured dots on top of a map, where each colour represented a specific application used.

4.1.3 Filtering

Instead of listing all the used applications in each of the proposed visualisations, a user should be able to select/deselect specific applications in order to get a better insight in differences among certain applications. This filtering mechanism should be used throughout the different visualisations, such that a change in selection in one visualisation, must be reflected in the other visualisations as well. This will allow the user to better customise the visualisations.

4.1.4 Data Collection

The parameters chosen for the data collection should cover the requirements presented earlier. Below, the necessary parameters to acquire the desired level of detail are outlined:

- Name of the application
- The location where the application was used
- The name of the previously used application (that opened this one)
- The duration of the interaction with this application (start- and end-time)

These parameter should suffice, as they cover the basic interactions between applications, their duration, and location. Information on the nearby wireless networks, phone state (e.g. roaming), battery levels etc. could also be collected to see if there was a trend within this data. However, to limit battery use and focus the project, the number of collected parameters was kept low.

4.2 Communication

This section describes the technical background and different decisions made for the implementation of the server-client communication. Furthermore, the design choices for the server implementation will be described.

The data collection must be stable, continuous and the transfer of data from client to server should happen regularly. Furthermore, the same data should never be uploaded twice from a client. The communication flow includes collecting, storing, transferring, and processing the data. This flow can be seen in Figure 4.3.

The basic idea is to have the client collect data continuously and to let the server update itself regularly (each server update should update the recommendations for clients). A client should be able to get an ID to identify itself, it should be possible for a client to upload collected data to the server and it should be possible for the client to ask for a recommendation.

This communication flow requires a few functionalities from the server set-up which will now be described.

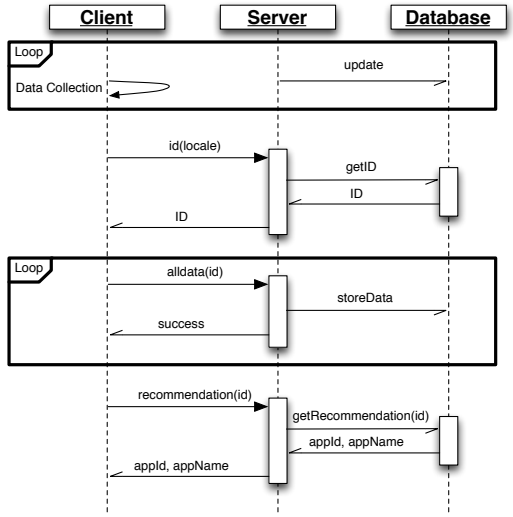


Figure 4.3: Flow-chart of the data flow between client, server, and database

4.2.1 Web Services

Following the illustration in Figure 4.3, a few web services, related to the communication between client and server, can be identified:

- **id** - this must take one argument, the user's "locale"[3], create a new user, and reply with the new user's id. The locale might be useful later on.
- **allApps** - this must take three arguments: the user id, all apps installed on the user's device[4], and the interactions performed since last upload. This data should be stored in the database.
- **recommendation** - must take one argument, the user's id, and reply with either a recommendation or an error message that no recommendations exist.

In order to update the recommendations regularly, yet another web service must be created. This should be invoked periodically to ensure up-to-date recommendations and similarity measures of users in-between. This web service, called

[3]See: http://developer.android.com/reference/java/util/Locale.html
[4]This is necessary in order to avoid recommendations of applications which a user has already installed

update, must initialise an update process which re-calculates all recommendations for all users. Optimally, this calculation should be performed during off-peak hours in order to ensure fast response time to clients when they are most likely to connect. When a new client connects however, he/she will have no recommendations available until the calculations have been made. This suggests that calculations should also be done after a new client has uploaded his/her first data.

4.3 Recommendations

For recommending applications to users, a recommendation algorithm needs to be chosen. As this part of the project is not the primary focus, a basic, yet commonly used (for recommendations)[5], algorithm has been chosen based on the correlation coefficient. This is also known as the Pearson Correlation Coefficient [Seg07, Ch. 2] and will be outlined below. It is based on the principle of collaborative filtering [RRS11], taking similar users and their "rating" patterns into account when recommending items for a particular user. An item-based approach could also have been used, looking at the similarity between items (applications) instead of users [SKKR01], or a completely different recommendation algorithm could have been used, e.g. Slope One [LM08].

The Pearson Correlation Coefficient (r), describes the linear dependency between two variables X and Y. It can be computed as follows:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

Where $\bar{X}$ and $\bar{Y}$ are the means of the populations X and Y respectively and n is the number of common items in X and Y. In relation to this project, X and Y each represent a user's "rating" of different applications.

Having calculated r, it is possible to calculate a predicted rating/popularity for the user a for a specific item i. This is denoted $p_{a,i}$:

$$p_{a,i} = \bar{X}_a + \frac{\sum_{u=1}^{n}(X_{u,i} - \bar{X}_u) \cdot r_{a,u}}{\sum_{u=1}^{n} r_{a,u}}$$

All users having rated (used) the item i contribute with their rating scaled by their similarity ($r_{a,u}$) to the user at hand. This gives a weighted prediction of

[5] Example implementation of a Pearson recommendation system for the Netflix data set, developed by Billy McCafferty: http://devlicio.us/blogs/billy_mccafferty/archive/2006/11/07/netflix-memoirs-using-the-pearson-correlation-coefficient.aspx

the user's rating, and the item with the highest rating is then an obvious choice for a recommendation to this user.

4.4 Data Analysis

This section covers the background for the data analysis, and outlines the tools for data analysis and visualisations. The details of the implementation of this part is found in Section 5.3.

4.4.1 Time Data

Following the analysis in Section 3.3.1, histograms are chosen as the primary visualisation method, as they are useful for displaying accumulated data over time. The visualisations will prove as a vital element in the data analysis, and could potentially be useful for users or developers who want to investigate tendencies within application usage over time.

In order to properly investigate the collected data, an interactive implementation of these histograms is necessary. The implementation must support the selection of a single user or all users. It must be possible to display the accumulated data over a week and over a day. A category-based selection should also be available, and it should also be possible to display the most used applications within each of the described selections.

4.4.2 Location Data

Since users' location data is collected, it is vital to see how this information can be utilised. First of all the data must be visualised on a map in order to make it easy to detect patterns and tendencies. Furthermore, when analysing the location data, it is essential to look at the distinction between different locations. Firstly, a naive algorithm will be outlined which filters the locations at hand and only leaves the distinct ones. Secondly a more sophisticated algorithm will be discussed.

```
function filterLocations(S)
    deleted ← []
    d ← 0.5                                   ▷ Threshold in kilometres
    for i = 0 → length(S) − 1 do
        for j = i + 1 → length(S) − 1 do
            if !j in deleted then
                if dist(S[i], S[j]) < d then deleted.append(j)
    deleted ← sort(deleted)                   ▷ Sort descending
    for k = length(deleted) − 1 → 0 do
        S.delete(k)
```

Algorithm 4.1: Naive location filtering algorithm

4.4.2.1 Simple Noise Reduction

Assume a number of different locations exist in the set S. These are sorted descending according to the number of application interactions on each location, meaning that the location $S[0]$ is the location with most application interactions. Now these locations will be filtered by the following rule:

For each location l which has not been removed, starting with the most popular one, remove all locations from the set S which are within the distance d of l.

This approach is outlined in pseudo-code in Algorithm 4.1, and will greatly reduce the amount of "noise" in the location data by reducing several locations close to each other to a few (or even 1) locations. The bad thing about this algorithm however, is that the locations which it leaves untouched do not necessarily represent good estimations of the centre of a dense area. It will rather focus on the most popular locations, which might lie in the border region of such an area. Furthermore, it will not preserve information about a user's transportation route (with bus or train for instance), as only some of the intermediate locations will be left untouched.

The naive algorithm has its advantages, but to introduce a more sophisticated approach, the next section will present a method based on clustering.

4.4.2.2 Clustering

A common way of determining distinct location groups, is to use clustering [XZL10]. Many clustering algorithms exist, but to find the most suitable one

function DBSCAN($S, \epsilon, MinPts$)
  $clusterID \leftarrow nextID()$
  for $i = 1 \rightarrow length(S)$ **do**
    $p \leftarrow S[i]$
    if $p.clusterID = UNCLASSIFIED$ **then**
      if $expandCluster(S, p, clusterID, \epsilon, MinPts)$ **then**
        $clusterID \leftarrow nextID()$

Algorithm 4.2: DBSCAN algorithm (from [EpKSX96])

for the data at hand, the data needs to be described in more details.

It can be assumed that the location data on a single user/device will be distributed among a number of different locations. Some areas will have a high density of registered locations (e.g. at home, school or work), and some areas will probably only have a few registered locations (when a user visits a new place he/she will not return to). It is plausible, that the areas with high density will have more or less the same density, as users will normally be static whenever they are at home/school/work.[6]

This behaviour is well fit for density-based clustering algorithm, as there will be areas with varying densities, and no predetermined number of clusters [KKSZ11]. One density-based clustering algorithm is the "DBSCAN" algorithm (described in details in [EpKSX96]). This algorithm uses two parameters to separate points into either clusters or noise. The ϵ parameter indicates the minimum radius in which at least $MinPts$ points must lie. If less than $MinPts$ points are within the radius ϵ of a point, this point will be considered as noise.

The algorithm takes a set of points S all having their $clusterID$ set to $UNCLASSIFIED$. The pseudo-code can be seen in Algorithm 4.2 and Algorithm 4.3. The latter relies on the method `regionQuery`, which finds the points within the distance ϵ of p over the set of points S. This method is the part of the algorithm which takes up most of the running time. However, this can be dramatically reduced (from $O(n^2)$ to $O(n \cdot log\ n)$) by using a `R*-tree` structure instead of a linear structure when searching for points within a given radius [EpKSX96].

This algorithm fits to the problem of this data, and will be used to identify patterns in users' locations.

[6] The density will become higher around these places even though the user's location does not necessarily change during a whole day's static presence at one location, since the Android location determination will sometimes change slightly even though the device has not moved.

```
function expandCluster(S, p, clusterID, ε, MinPts)
    seeds ← regionQuery(S, p, ε)
    if length(seeds) < MinPts then
        p.clusterID ← NOISE
        return false
    else                          ▷ All points in seeds are reachable from p
        for i = 1 → length(seeds) do
            seeds[i].clusterID ← clusterID
        seeds.delete(p)
        while length(seeds) > 0 do
            newP ← seeds.first()
            newSeeds ← regionQuery(S, newP, ε)
            if length(newSeeds) > MinPts then
                for j = 1 → length(newSeeds do
                    resultP ← newSeeds[j]
                    id ← resultP.clusterID
                    if id = UNCLASSIFIED or id = NOISE then
                        if id = UNCLASSIFIED then
                            seeds.append(resultP)
                        resultP.clusterID ← clusterID
            seeds.delete(newP)
        return true
```

Algorithm 4.3: expandCluster algorithm (from [EpKSX96])

4.4.2.3 Clustering Processing

Once a user's locations have been processed, vital features can be extracted from the clusters. Firstly, a number of visual interpretations can be made, which can outline and explain how locations are connected. Secondly, it is possible to process the clusters even furtherer and see if any characteristics can be found. The approach chosen here, will try to enlighten whether the application usage within a cluster follows a specific pattern - meaning that the usage pattern within a cluster is different from the overall trend. This can be calculated by using the Pearson Correlation Coefficient (described in Section 4.3), and see if this coefficient is larger within a specific cluster than when comparing to the locations outside the cluster, and to the average coefficient.

The calculation of Pearson Correlation Coefficients for each pair of locations is quite demanding, especially with regards to memory consumption. The runtime is $O(n^2)$, and can be calculated like this:

Calculation of averages: $O(n)$
Calculation of correlation coefficients: $O(\frac{n^2-n}{2}) = O(n^2)$
In total: $O(n^2)$.

Note that only one coefficient is needed for each set of locations, therefore only half of $n^2 - n$ needs to be calculated - the subtraction of n is due to the fact that all coefficients between a location and itself is 1, making these calculations irrelevant.

Comparing locations and the usage pattern at each location reveals new possibilities for clustering analysis. This actually makes it possible to run the clustering algorithm, described above, with a different distance measuring algorithm based on Pearson Correlation Coefficients. If implemented properly, this might show indications of new trends within users' behaviour. However, this approach also heavily depends on parameter estimation, and it might be harder, visually, to decide on the quality of a certain set of parameters.

4.4.2.4 Clustering Validity

A number of different techniques exist for validating and evaluating results of a clustering algorithm's output. Basically, assessing the validity of a clustering results is often done visually in order to decide on the number of clusters for instance. For a density-based clustering algorithm it is not possible (directly) to adjust the number of clusters. However, the clusters generated will be affected by the input parameters and indirectly this will affect the number of clusters as well as the shapes and sizes of these. More advanced analysis can also be conducted, looking at a clustering result's external, internal and relative criteria. Here, the external criteria tests for randomness in the data, and it is necessary to have some knowledge about the cluster structures. The internal criteria evaluates the result of a clustering in terms of the data in the clusters. Finally, the relative criteria (for an algorithm where the number of clusters is not a parameter), seeks to find the best set of parameters by adjusting these parameters within a wide range, providing a stable number of clusters and choose the middle of this range [HBV01].

In the clustering analysis in this report, the emphasis will be on the visual validation of clusters. As there is no preceding knowledge of the clusters' shapes

the external criteria analysis will not be conducted. The internal criteria will be evaluated based on the Pearson Correlation Coefficient (this could also be done using the Cophenetic Correlation Coefficient) and the relative criteria will be used to define the best set of parameters for the DBSCAN algorithm.

4.4.3 Application sequences

The relationship between applications is interesting, since it reveals if there are indications of one transition being more likely to happen than another.

A transition from one application to another is considered as two dependent events - one being that the first application must be the successor, the other that the second application must be the predecessor. They are assumed to be dependent, as it is not random which application will be opened, once you have another application open. By making this assumption, it is possible to treat transitions with conditional probability [Seg07, Appendix 3, p. 319]. This makes it possible to calculate a probabilistic guess of how often a transition will take place, and also to see if some successors are more dependent on specific successors than others. This makes it possible to detect tendencies and relations between applications.

In general, the probability of event B given event A is denoted: $P(B|A)$, and can be calculated like this:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

The probability of event B given any event is denoted: $P(B)$, and is calculated like this:

$$P(B) = \frac{occurrences\ of\ B\ as\ successor}{the\ number\ of\ transitions}$$

Putting it into context, the probability of application B being launched after application A has been opened is:

$$P(B|A) = \frac{occurences\ of\ A \rightarrow B}{occurences\ of\ A\ as\ predecessor}$$

To investigate whether application B is opened more frequently by application A than regularly, $P(B|A)$ and $P(B)$ can be compared. If $P(B|A) > P(B)$ it means that there is a (positive) dependency between application A and B, and the magnitude of the difference indicates the level of dependency. This difference can be compared between a number of different transitions to get an indication of strong and weak dependencies across different application sequences.

Chapter 5

Implementation

This chapter describes the technical choices and challenges related to the implementation. Firstly, the mobile application (client) development will be described and secondly, the server will be outlined. The client-server communication follows the proposed design from Section 4.2. Finally, the details on the visualisations and data analysis will be described.

The description of the tests for the implementation, as well as the background for these, can be found in Appendix B on page 75.

5.1 Mobile Application Development

A brief overview of the different important components in Android application development will be covered in this section. Furthermore, the development of the application, "Appalyzer", will be described with regards to the monitoring itself. For more technical details on the application development, please refer to Appendix A on page 71.

5.1.1 Android Development

The most common component in an Android application is an `Activity`[1], which most often represents a single screen in an application which the user can interact with. Activities communicate with `Intents`[2], which can parse arguments, values, and basically intentions.

Another important component is the `Service`[3], which is designed to run long running processes (as a music player for instance), or to provide functionality to other applications on the device.

A `BroadcastReceiver`[4] is designed to receive all kinds of `Intents` from both the operating system, and from other applications[5]. This means that once registered, it does not necessarily need to be running to receive a notification that an event occurred. For instance, it might be notified if a text message is received, if the wifi state changes, or if the battery charge level changes. It is possible to register for several events at a time, using an `IntentFilter`[6]. `BroadcastReceivers` can be registered in the `Manifest` of the application, or programatically (for instance in an `Activity` or `Service`). The latter, however, requires that the component which registered it, is still active – otherwise the `Intent` will not be received. The registration for some events can only happen programatically.

5.1.2 Application Monitoring

Monitoring which application the user is currently using on an Android device involves several steps. When knowing the Android framework, it might lead one to believe that it is possible to "listen" for a change in the currently active application. This is a relevant consideration, based on the previous description of the `BroadcastReceiver`. However, there exists no `Intent` which indicates that the application in use has changed. Therefore, the task of continuously registering the active application quickly gets quite complicated.

[1] http://developer.android.com/reference/android/app/Activity.html
[2] http://developer.android.com/reference/android/content/Intent.html
[3] http://developer.android.com/reference/android/app/Service.html
[4] http://developer.android.com/reference/android/content/BroadcastReceiver.html
[5] For a comprehensive list of possible intents, see http://developer.android.com/reference/android/content/Intent.html
[6] http://developer.android.com/reference/android/content/IntentFilter.html

5.1.2.1 Finding the active application

Finding the currently active application on an Android device requires only a few lines of code, and can be seen in the code listing in Figure 5.1. It fetches the `ActivityManager`[7], requests a list (of 1) of `RunningTaskInfo`[7], pulls the first element, and returns the package name of this task's base activity.

```
1  public static String getCurrentlyRunningApp(Context context) {
2    ActivityManager m = (ActivityManager)context.getSystemService(
         Context.ACTIVITY_SERVICE);
3    if(m!=null) {
4      List<RunningTaskInfo> tasks = m.getRunningTasks(1);
5      if(tasks!=null && tasks.size()>0) {
6        String latest = tasks.get(0).baseActivity.getPackageName();
7        return latest;
8      }
9    }
10   return null;
11 }
```

Figure 5.1: Code for getting the currently running application/task in Android

This package name will represent the "active" application - meaning the application which the user sees, and therefore currently uses, on the device.

5.1.2.2 Continuously register the active application

There are several different ways of running a process for a longer period of time. The `Service` is, as described, designed for this situation, but as the registration should not really happen "continuously", rather every other second, this might be too heavy a task. Therefore, the chosen solution is to use a `BroadcastReceiver`, which is triggered with a given interval by the `AlarmManager`[8]. The `AlarmManager` is particularly useful for this task, since the registration of the application in use should also happen when the application ("Appalyzer") is not running cf.:

> "The Alarm Manager is intended for cases where you want to have your application code run at a specific time, even if your application is not currently running."[8]

[7] http://developer.android.com/reference/android/app/ActivityManager.html
[8] http://developer.android.com/reference/android/app/AlarmManager.html

This approach is straightforward and utilises the operating system's built in mechanisms for running a part of an application continuously at given intervals. However, for the purpose of this project, it is not relevant to register the application running when the screen of the device is off. This means that the user is not actively interacting with an application[9], and the `BroadcastReceiver` should be stopped in this situation. When the screen is turned off/on - either by the user or by the operating system, an `Intent` will be broadcasted. These two `Intents` (`Intent.ACTION_SCREEN_ON` and `Intent.ACTION_SCREEN_OFF`) cannot be registered in the application's `Manifest` file[10], meaning that they will only be triggered as long as the application is running. Therefore, a Service is implemented, with the sole purpose of registering a `BroadcastReceiver` for when the screen turns on/off.

5.1.3 SQLite Database

The continuous registration of the running application was stored locally on each device in a `SQLite`[11] database. Instead of logging data every other second, a row in the database was only created once the application in use changed. The data stored for each interaction with an application was the following:

- Title – Package name of this application
- Latitude – the latitude registered (if any) when the use of the application ended
- Longitude – the longitude registered (if any) when the use of the application ended
- Predecessor – The application used right before this application (if any was used)
- Start time – when the use of this application started
- End time – when the use of this application ended

[9] However he/she might listen to music from an application running either as a foreground or background application. However, as background applications (`Services`) are not considered for application monitoring, this would not give an accurate measure of the consumption of e.g. the music player anyway.

[10] Also mentioned here: http://thinkandroid.wordpress.com/2010/01/24/handling-screen-off-and-screen-on-intents/

[11] A smaller version of a full `SQL` implementation. It is a built-in component in the Android framework, and provides easy access to a powerful database integrations, see: http://www.sqlite.org/

- Uploaded – initially false, indicating that this row has not been uploaded to the server yet

As more and more application interactions are registered, the size of the database will grow. Therefore, the database is "cleaned up" once in a while to remove old data (default is to remove data older than 14 days). This ensures that the application will not use too much storage; especially as the stored data takes up very little space in an optimised database environment such as `SQLite`.

5.2 Server Development

In order to collect the data from the different devices, a server was set up. The server was a machine running `Centos`[12], provided kindly by Adapt A/S[13]. An early attempt to use Amazon Elastic Compute Cloud (Amazon EC2)[14] was made with little success, due to poor performance and limited resources provided by Amazon.

Below, the overall structure of the implementation will be described.

5.2.1 Server Set-up

For implementing the web services described in Section 4.2.1, `Node.js`[15] is chosen as the server language. `Node.js` has many usefull features, is fast to implement, and is a modern way of writing a web server [TV10].

The implementation can basically be divided into two parts: The communication part which allows access from clients, and the processing part which interacts with the database and makes calculations. The communication part creates the server and listens for connections. Once a connection is made, it determines whether the incoming request is a htttp-get or http-post request, and makes sure that the appropriate method handles the request. All communication between this server and the clients are done with `JSON`[16], which is easily parsed

[12] http://www.centos.org/
[13] http://www.adapt.dk
[14] http://aws.amazon.com/ec2/
[15] Platform for building scalable network applications written in Javascript: http://nodejs.org/
[16] http://www.json.org/

on an Android device – even though it might be a bit slower and consume more memory than XML data [Abl].

The communication interface to the server is pretty basic, and the complications lie solely in the data processing part of the server. Here, the fact that `node.js` is a non-blocking system, means that most function calls need a callback method in order to return any feedback to the caller-function. This in itself is not complicated, but once further calculations depend on a lot of nested calls, this means that a lot of callback will be involved. Therefore, the `Step`[17] library is used extensively when performing more complex tasks on the server.

To run the server, and make sure it keeps running, a `node.js` module called `forever`[18] was used. This allowed automatic reboots of the server on errors and maintained error- and output-logs.

To ensure that the correlation coefficients were computed continuously, a cron-job[19] was setup to call the server's update service once every 10 minutes.

5.2.2 Database

The choice of database system ended on `MongoDB`[20], as this has a straightforward integration with `Node.js`, for instance through the module used in the implementation, called `mongojs`[21]. When building a database in `MongoDB`, the structure consists of collections - each representing a set documents. The choice of collections is important in order to ensure fast lookup times and minimal data duplication. Below, the different collections chosen for the implementation are listed:

- **users** - holding information on all users (id and locale)
- **apps** - holding information on all applications (id, package name and alias)
- **interactions** - all interactions are stored here with user id, application id, start time, end time and predecessor application
- **pearsons** - holding all Pearson Correlation Coefficients between users (user id1, user id2, and pearson correlation)

[17] Control-flow library for node.js, which enables serial execution: https://github.com/creationix/step

[18] Simple tool ensuring a script runs continuously: https://github.com/nodejitsu/forever

[19] Scheduler which allows running tasks periodically on a machine.

[20] See:http://www.mongodb.org/

[21] Module which emulates the official `MongoDB` API, see: https://github.com/gett/mongojs

- **recommendations** - holding all recommendations for all users (user id, application id, calculated rating, and whether the recommendation has been given (sent to the client) or not)

- **stats** - holding the accumulated "rating" of each application for each user

The above collections support all the operations needed to perform the different calculations, and ensure that key figures are easy to extract - for instance a user's "rating" of a specific application. The **interactions** collection hold the raw data, and all the other collections' data can be reproduced from this one.

5.3 Visualisations and Data Analysis

In order to implement some intuitive and fast visualisations of the collected data, a few web-pages were build on a different server than the one used for data collection[22]. This server was also written in `Node.js`, as it would have to interact very closely with the database, and particularly the collection `interactions`. The front-end was written in `HTML` and `Javascript` using `D3.js`[23] and `Polymaps`[24].

The emphasis of the web pages has been on functionality, and little effort has been made to optimise the visual impression by any of the pages. Furthermore, there is no cross-browser compatibility, and the implementation has only been tested in Google Chrome[25].

The web services utilised by the pages are outlined in Table 5.1.

5.3.1 Histograms

The histograms have been implemented using the `D3.js`-library, and supports all of the customisations presented in Section 4.4.1. It uses a minimum number of calls to web services in order to ensure fast and smooth transitions between states. Only when changing category or from one user to another (or to all users), a web service is used.

[22] To ensure that one server did not impact the other.
[23] Upcoming DOM-manipulating JS-based visualisation tool, good for working with SVG-files, see: http://d3js.org/
[24] JavaScript library for making dynamic, interactive maps, see: http://polymaps.org/
[25] https://www.google.com/chrome

Name	Description	Used by
dayinteractions?userid	Fetches all interactions performed by the selected user	Histograms
allinteractions	Fetches all interactions performed by all users	Histograms
categoryInteractions ?category	Returns all interactions performed by all users in the given category	Histograms
all_users	Fetches all users available	Histograms and Map
locations?userid	Fetches all registered locations on the selected user	Map
popular_apps_location ?userid+latitude+longitude	Fetches the applications used by the selected user at the specific location	Map
locationAnalysis?userid	Fetches the applications used by the selected user for all of his/her registered locations	Map
all_apps	Fetches all registered applications	Interactions
interaction?appid	Sends all the interactions on the specified application	Interactions

Table 5.1: Web services used for visualisations

The categories are fetched from Google Play, and all of the applications which are not available on Google Play[26] are categorised as "Uncategorised". As Google Play has no public available API, these categories are collected with basic `HTML` scraping.

The implementation supports viewing application usage accumulated over a week or a day, and also has an option of showing the most used applications. These options are available when investigating a single user or all users. For both selections, it is possible to investigate a single application or all the used applications.

By default, the implementation displays accumulated usage, but it is also possible to select a normalised visualisation, where the area under the histogram sums up to 1 (100%). If this visualisation is selected, while investigating all users, every user's contribution to the overall count is divided by the total number of interactions by this user, in order to eliminate too big contributions from users with a lot of interactions.

5.3.2 Transitions

For the transition analysis, the method mentioned in Section 4.4.3 is used. This part of the project does not have a finished GUI[27], but has still been implemented as a web page where it is possible to pick an application. The output of the transition calculations is written as a `html` table on the page, in a format ready for further processing in Excel. The results of this processing is seen in Section 7.2.

The implementation can be found at:
http://89.233.45.28/interactions.html

5.3.3 Locations

The implementation of the map visualisation was done by adding `svg`-elements[28] (circles) on top of a `Polymap`[29]. The map data was provided by

[26] Either because the application is a system application, or is an application which has not been released but only tested by a developer.
[27] Graphical User Interface
[28] Scalable Vector Graphics, see: http://www.w3.org/Graphics/SVG/
[29] See: http://polymaps.org/

`OpenStreetMaps`[30] through `CloudMade`[31].

All locations collected by the Android application are recorded with at least 7 decimals. However, since the locations have not been obtained through each device's GPS, but only requesting locations with "coarse accuracy"[32], the accuracy cannot be considered to be this high[33].

Therefore, the locations are rounded to 4 decimals[34], and colliding locations and their application usage are merged (on the server-side). This greatly reduces the number of locations, but also provides more realistic estimates of the actual locations. The locations given from the server to the client however, are provided with 8 decimals for practical reasons in the client implementation.

The processing of locations and calculations of cluster similarity follows the procedure presented in Section 4.4.2.3, and the output of this processing is written as a `html` table on the page when finished.

Below, the algorithms used to process the locations and some of the difficulties doing so will be outlined.

The implementation can be found at `http://89.233.45.28/locations.html`

5.3.3.1 DBSCAN

The implementation of the `DBSCAN` algorithm follows the approach outlined in Section 4.4.2.2. It is implemented on the client side, as the client has already all of the locations on a user at hand. In order to ensure the best performance, the `regionQuery`[35]-method is implemented using an `R-tree` implementation[36]. This unfortunately means that the distance measuring method is not 100% accurate, as it relies on rectangles instead of circles around a given point. Nevertheless, it also means a huge performance improvement when searching for locations near a given point. This performance improvement outweighs the drawbacks, since it lowers the overall running time of `DBSCAN` from $O(n^2)$ to $O(n \cdot log\ n)$. It would have been preferable to use an `R*-tree` implementation

[30] See: `http://www.openstreetmap.org/`
[31] See: `http://cloudmade.com/`
[32] See: `ACCURACY_COARSE` on `http://developer.android.com/reference/android/location/Criteria.html`
[33] The accuracy seems more likely to be below 200 ft $\approx$ 61 meters, see: `http://developerlife.com/tutorials/?p=1375`
[34] Approximately 11 meters precision in latitude and 6 meters precision in longitude
[35] Method which finds locations near a given point.
[36] Developed by Jon-Carlos Rivera, see: `https://github.com/imbcmdth/RTree/`

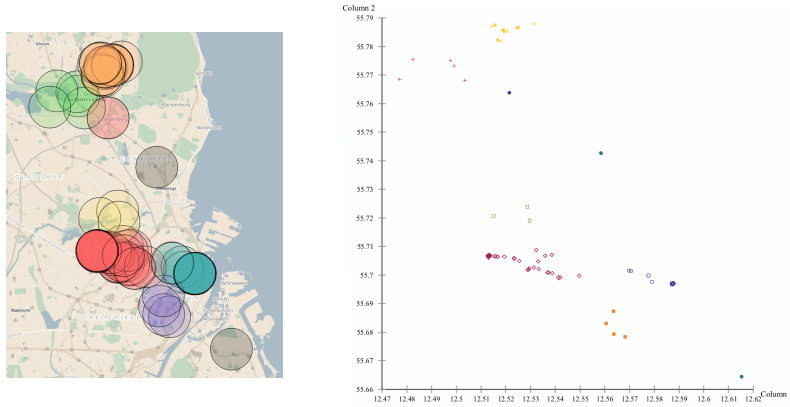


Figure 5.2: Comparison of results obtained by implemented `DBSCAN` method (on the left), and results from the "ELKI" tool (on the right).

instead of the `R-tree` implementation as it has a few advantages, but considering the straightforward implementation of the `R-tree`, this suffices.

To validate the implementation of the `DBSCAN` algorithm, the obtained (visual) results were compared to results produced by a project called "ELKI"[37], which incorporates a lot of clustering algorithms. In order to compare the results properly, the `regionQuery` method was implemented using a standard "haversine"[38] distance measure. The comparison can be seen in Figure 5.2, and shows that the obtained clusters match 100% given identical parameters $(\varepsilon, minPts) = (1, 2)$.

As mentioned, the distance measure used by the `R-tree` implementation is not as accurate as the haversine measure, and the differences in cluster findings is illustrated in Figure 5.3. Even though the two methods agree on most clusters (for the given set of locations), a difference worth mentioning is the introduction of a new cluster when using the `R-tree` implementation.

5.3.3.2 Memory Optimisations

When calculating Pearson Correlation Coefficients between different locations, they all have to be stored in memory in order to access them later for calculations of individual cluster averages. First of all, each set of locations is only calculated

[37] Environment for Developing KDD-Applications Supported by Index-Structures developed and maintained by developers at Ludwig-Maximilians-Universität in München, see: http://elki.dbs.ifi.lmu.de/wiki

[38] See: http://www.movable-type.co.uk/scripts/latlong.html

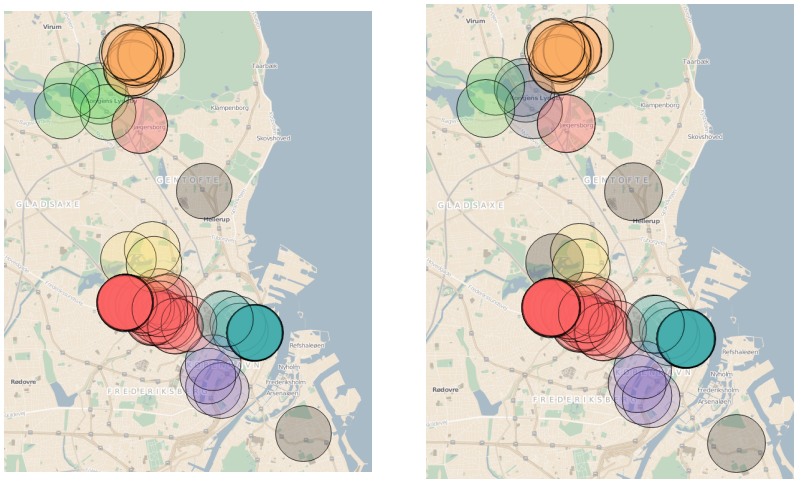


Figure 5.3: Illustration of the differences between using a haversine distance measure (on the left) and a rectangular distance measure for the `DBSCAN` algorithm (on the right).

once (following the discussion in Section 4.4.2.3). Furthermore, in order to ensure a quick lookup time of each coefficient, these had to be indexed based on the two locations they represented. The most intuitive way of doing this was to generate a key (the key was for an associative array[39] in Javascript) consisting of a string concatenation of both locations, e.g.:

$$key = 55.69106710 + 12.51429970 + 55.69104490 + 12.51432740$$

Unfortunately, this quickly leads to a large memory consumption as the key itself takes up a relatively big amount of memory, considering that the number being stored at a specific key is a Pearson Correlation Coefficient rounded to four decimal places. The solution was to reduce the key size with a simple hash function, which multiplies the latitudes and longitudes of the two locations together[40], and uses the last 10 decimals of this number as the key. Using this hash function greatly reduced the memory consumed by the keys (from 40 characters to 10 for each key), and is still relatively robust against hash collisions. The probability that two sets of locations ($2x2$ locations) produce the same hash key can be calculated using the birthday paradox. In general, the probability that at least two numbers are the same when drawing n random

[39] See: http://blog.xkoder.com/2008/07/10/javascript-associative-arrays-demystified/

[40] It sorts the latitudes and longitudes respectively before multiplying to ensure that the calculated hash key is the same no matter which order the two sets of locations were given in. Otherwise rounding errors might give different hash keys.

numbers from a distribution from $[1, d]$ is[41]:

$$p(n, d) \approx 1 - (\frac{d-1}{d})^{n(n-1)/2}$$

With the hash function described above, this means drawing n sets of locations from a distribution from $[1, 10^{10}]$.[42] The user with most locations (at the moment) has registered 671 different locations.[43] This makes the number of combinations of locations: $\frac{671^2-671}{2} = 224,785 \approx 2.2 \cdot 10^5$ This makes the probability for collision:

$$p(n, d) \approx 1 - (\frac{10^{10}-1}{10^{10}})^{2.2 \cdot 10^5 (2.2 \cdot 10^5 - 1)/2} \approx 0.92$$

This means that a collision is almost inevitable. The fact that there are hash collisions however, is not necessarily a problem if only these are handled properly and are not too great in numbers. Running the algorithm for the user with most registered locations, it turns out that there are "only" 5 collisions (out of $2.2 \cdot 10^5$ insertion attempts), and at most 2 different sets of locations collide on a single hash key. Utilising this fact, the indexed Pearson Correlation Coefficients are still stored as numbers - however, when a collision is detected, the new set of locations are stored under the same key but with information on the latitudes and longitudes of this colliding set as elements in a list.[44] The resulting code for inserting a coefficient in the indexed list can be seen in Listing 5.1. The lookup of a coefficient also gets a bit more complicated, since it has to be checked whether the entry is a list or a number. The code for the lookup can be seen in Listing 5.2. This makes the lookup a bit slower, but as long as there are not too many entries with the same key, it is not significant.

```
1  if(indexedPearsons[hashKey] != undefined) { //Collision
2    if(type(indexedPearsons[hashKey])=='Array') //Not first collision
3      indexedPearsons[hashKey].push({la: lat, lo: lon, la1: lat2, lo1:
           lon2, p: myround(pearson,2)});
4    else //First collision with hashKey
5      indexedPearsons[hashKey] = [indexedPearsons[hashKey],{la: lat, lo:
           lon, la1: lat2, lo1: lon2, p: myround(pearson,2)}];
6  } else { //No Collision
7    indexedPearsons[hashKey] = myround(pearson,2);
8  }
```

Listing 5.1: Procedure for inserting a Pearson Correlation Coefficient in indexed list

[41] See: http://en.wikipedia.org/wiki/Birthday_problem

[42] Technically 0 is also a possibility, but is ignored in this discussion in order to use the described formula.

[43] After rounding locations to 4 decimals. With 8 decimals precision, the user with most locations has 3772 locations.

[44] This technique is also known as "hashing with chaining" [CLRS09, Ch. 11]

```
1  function getPearson(lat, lon, lat1, lon1) {
2    var v = indexedPearsons[hash(lat,lon,lat1,lon1)]; //Read the entry
3    if(v==undefined) {
4      console.log('Pearson error ...');
5      return v;
6    }
7    if(type(v)!='Array') return v; //v is just the coefficient
8    if(v.length>maxPLength) maxPLength = v.length;
9    for(var i = 1; i<v.length; i++) {
10     if(v[i].la==lat && v[i].lo==lon && v[i].la1==lat1 && v[i].lo1==
           lon1
11         || v[i].la==lat1 && v[i].lo==lon1 && v[i].la1==lat && v[i].lo1
               ==lon) {
12       return v[i].p; //The list item matched
13     }
14   }
15   return v[0]; //No match - the coefficient must be the first entry
16 }
```

Listing 5.2: Procedure for reading a Pearson Correlation Coefficient from indexed list

5.3.4 Asynchronous Data Processing

All interactions with web services are done asynchronously using `jQuery`[45]. This however only makes the client-server interaction asynchronous. In order to ensure that the browser remains responsive while performing heavy computations on the client side, a popular pattern is used for making these calculations more or less asynchronous.[Lec07]

This pattern is used extensively for the `DBSCAN`-algorithm, the calculation of Pearson Correlation Coefficients and for the post-processing of these. The pattern utilises the asynchronous `setTimeout` method in javascript in order to give the browser periodic intervals for responding to user events. The `setTimeout` method is called between loop iterations in the heavy algorithms, with an interval of 0 and recursively continues the loop. Even though the interval is 0, this gives the browser a chance to remain responsive.

For most users (in the database) the use of this pattern is really not necessary, as their number of registered locations is quite small (below 200). However, some users have more than 3000 different registered locations, and this takes a lot of time to process - potentially blocking the browser for minutes.

[45] See: http://jquery.com/

Chapter 6

Experimental Work

In this chapter the experimental work and some of the challenges within this work will be described.

6.1 Application Release

After finishing the implementation of the Android application and the server solution, the application: "Appalyzer"[1] (screenshot in Figure 6.1) was released on Google Play on 9 May 2012 - 3 months (and 3 days) after the development had started. In order to get people to download (and use) the application, friends and co-workers were encouraged to install it through various social media. It quickly had more than 20 active installations, and now (1 August 2012) has 17 active installations with a total number of installations (over time) of 46.

More work could have been done in order to promote the application and thereby get more downloads. However, the number of users seemed adequate for the project scope, especially since the server at hand only had 10 gigabytes of memory for user data, and is presently already half full.

[1] See: https://play.google.com/store/apps/details?id=com.dtu.appalyzer

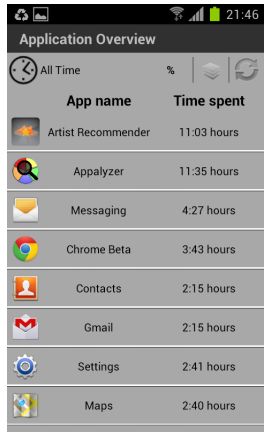


Figure 6.1: Screenshot of "Appalyzer"

6.1.1 Time Zone Compensation

When the application was released it was assumed that only a few friends would download and use it, and therefore the system was not ready to handle different time zones. It turned out however, that some users were from Asia, Canada and the United States. As the times registered on each device is the local time, this meant that the registered times for these distant devices had to be handled when displaying histograms in order to generate correct results. But as the system was not initially prepared for handling time zones other than the Danish (UTC+1), this meant that the time zone had to be calculated on basis of the locations registered along with each time registration. Unfortunately, this meant that data generated by users, who had not allowed the registration of locations (on their device), was no longer valid for generating histograms, as there was no knowledge of the time zone of these registrations.

Actually the locale of each user was recorded when users started using the application, but this merely indicates the language and country of the device owner, and does not necessarily reflect if the user is travelling.

For the implementation of this compensation, two files with locations[2] and time zone offsets[3] respectively were used in combination with the earlier mentioned `R-Tree` implementation to estimate offsets for the registered interactions.

[2]See: https://raw.github.com/gist/1769458/dc79733325f9b468ff2d44ff3813924d6d1e6382/lat,lng,timezone

[3]See: http://download.geonames.org/export/dump/timeZones.txt

6.1.2 Collected Data

During the period from release up until the delivery of this report, more than 200,000 interactions were collected (169,000 after time zone compensation), where each interaction represents a single user's transition from one application to another with a timestamp and location attached. Furthermore, 1468 unique applications (1272 after time zone compensation) and 11,774 unique locations (11,064 after time zone compensation) were recorded. This dataset represents the data used for further analysis in Chapter 7 and Chapter 8.

6.2 Application Feedback

The overall feedback has been positive but sparse. One user mentioned, that application names on system applications, such as the native launcher application on Samsung mobiles ("TwLauncher"), might not be that meaningful when presented to the average user. This is something which should be dealt with for future work, as some system application names might be confusing to some users.

One user reported that the application had registered his browser being in use all night, even though he turned of his device. This is a known issue and is described below.

Feedback was also received from a user who enjoyed the application, and suggested a new feature where it was possible to categorise applications in order to make it possible to see how much time was spent on games, browsing etc. This feature was actually implemented at an early stage, but was left out since it was considered to be too confusing and complex to some users.

Some users mentioned that they registered a small change in their battery consumption after installing "Appalyzer", but none of these were sure that it was because of this. Furthermore, the application was not visible under the listed applications when investigating each device's battery consumption. The battery consumption by "Appalyzer" might also be device specific.

6.3 Known Issues

6.3.1 Faulty Data

When developing the Android application, work was done to ensure that if a device was shut down, the application would be notified in order to register the end time of the current application use. However, it seems that some devices had issues related to registering the end time of an application use, as the application or device had been shut down in the mean time. When the application was started again, this meant that the previous application use was ended, leading to wrong end times. The start time however, should be valid as well as the number of interactions (number of registered application starts) and transitions between applications.

6.3.2 Server Limitations

The server implementation ran into some memory limitations related to the big number of registered users and applications. The server kept running, but was occasionally not able to calculate user recommendations and similarity scores. It also meant that users trying to receive a recommendation either did not succeed or had to wait quite some time. As this was not the primary focus for the project, work was not done to fix this issue, and after collecting a sufficient amount data, the recommendation feature has been discontinued (the server will respond with “no recommendations available”).

Chapter 7

Results

Results and illustrations from the data analysis will be described and presented during this chapter. More thorough analysis and reflections on these results will be conducted in Chapter 8.

7.1 Application usage

A normalised histogram of all users' application usage (each use of an application counts as 1) accumulated over a week and a day can be seen in Figure 7.1. It gives a clear overview of users' behaviour over time, and indicates an increasing application interaction activity during the day, with the most application interactions seen during afternoons and evenings. Furthermore, it seems that the number of application interactions is lower during the weekend.

When considering interaction counts for all users, only users with registered interactions on at least 2 different days are considered. This is to avoid having users with few interactions disturb the overall picture.

Looking at individual users, the results vary a lot, as some users have registered a lot of interactions and some only a few, but for some users it is possible to

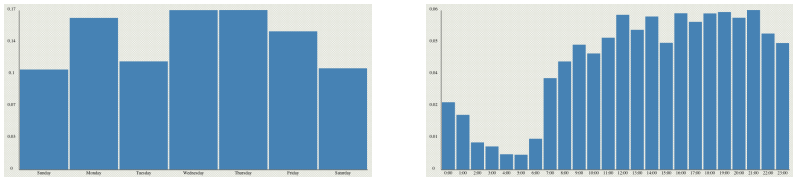


Figure 7.1: Accumulated normalised application usage over a week (on the left) and over a day (on the right) for all users

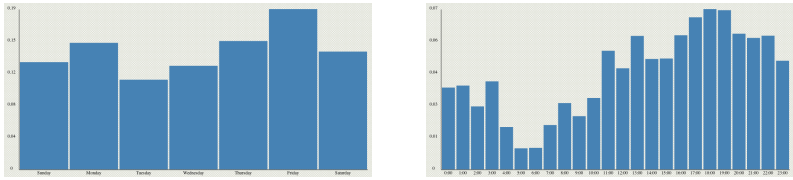


Figure 7.2: Accumulated normalised application usage over a week (on the left) and over a day (on the right) for a single user (`4fc95b060806783e16014003`)

see the same tendencies as in Figure 7.1. This is illustrated in Figure 7.2, and even though some days and time periods differ from the overall picture, it is evident that the pattern (for the daily usage at least) is similar. The weekly usage pattern seems to be more individual.

Next, the use of single applications by all users is investigated. This can be seen in Figure 7.3, where two applications are visualised. The application on the left is a popular multi-player puzzle game, which seems to be rather popular - even after midnight. Looking at other games[1], the tendency seems to be the same: playing games is a late evening activity. The application on the right is a weather application, and is very popular in the morning and evening.

Looking into Google Play's application categories, it is interesting to see if different categories are used differently over time (as with the games mentioned above). In Figure 7.4 two different categories are illustrated, indicating that communication applications follow the overall trend described earlier with little activity during weekends, and shopping applications are used mostly during noon and afternoon.

More histograms can be generated and customised at: `http://89.233.45.28`

[1] "Angry Birds Rio" (com.rovio.angrybirdsrio), "ETERNITY WARRIORS" (com.glu.android.warriors), "GUN BROS MULTIPLAYER" (com.glu.android.gunbros_free), and "Rule the Kingdom" (com.gameinsight.kingdom)

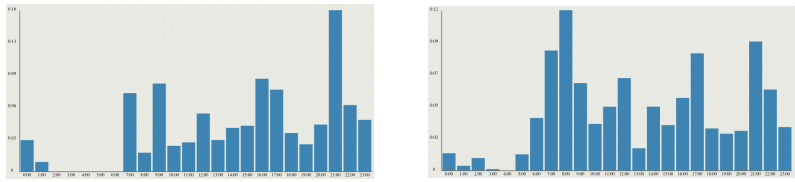


Figure 7.3: Accumulated normalised application usage of "Wordfeud Free" (on the left) and "Danish City Weather from DMI" (on the right) over a day.

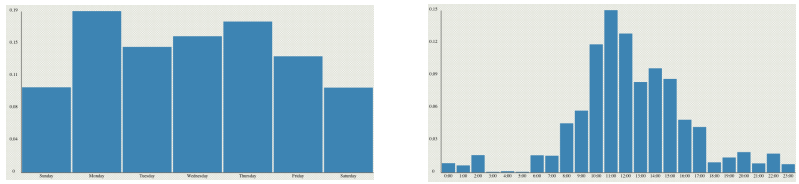


Figure 7.4: Weekly and daily accumulated normalised application usage respectively of the categories "Communication" (on the left) and "Shopping" (on the right).

7.2 Transition Analysis

Using the implementation described in Section 5.3.2, a number of popular applications are investigated to present their relation with other applications. Note, that for each investigated application, a few successors have been removed, as these applications did not seem interesting for the investigation at hand, and would confuse more than enlighten things. The majority of these were home screen applications. In all the figures used, the red horizontal line indicates a neutral dependency between the predecessor and successor. All markers above the red line indicate a positive dependency, and those below indicate a negative dependency.

In Figure 7.5, "Facebook for Android" (com.facebook.katana), denoted Facebook, is investigated, and the package names of its successors (the applications which it has launched) are listed. It is clear that there is a large positive dependency between Facebook and the built in video player (as this is utilised for playback of videos in the Facebook application), and a positive dependency between Facebook and some of the popular browsers. Furthermore, it seems that there is a very weak dependency between Facebook and most of the communication applications (with exception of a built-in conversation application from Sony

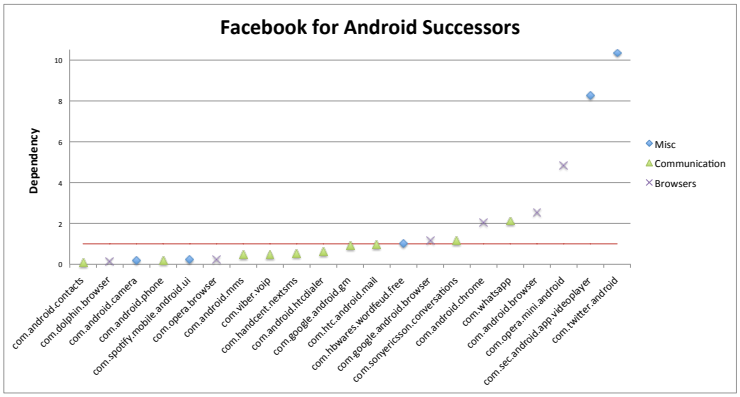


Figure 7.5: Investigation of Facebook's successors

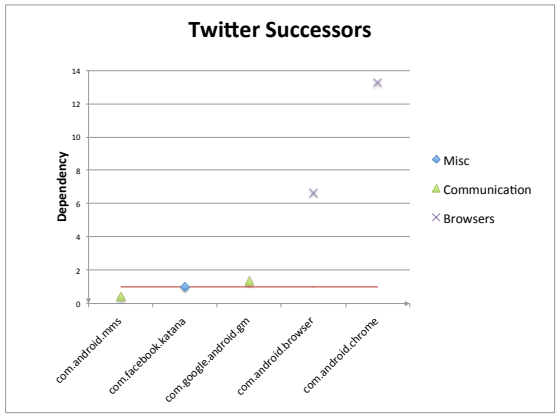


Figure 7.6: Investigation of Twitter's successors

Ericsson and a custom text messaging application called What's App[2]).

Looking at the most common Twitter-application for Android: "Twitter" (com.twitter.android) and its successors, the trends from the Facebook-investigation seem to recur. This can be seen in Figure 7.6. However, the number of successors to this application is limited, and therefore the validity of the trends can be hard to determine. It is interesting however, that some of the text messaging applications listed in Figure 7.5 are not seen at all in Figure 7.6, as these have not been launched (enough) by the "Twitter" application.

Looking at a different genre than social applications, the most popular mail ap-

[2]See: https://play.google.com/store/apps/details?id=com.whatsapp

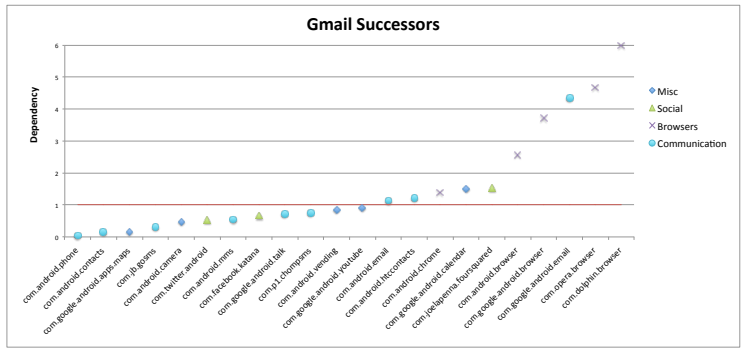


Figure 7.7: Investigation of Gmail's successors (com.dolphin.browser actually has a dependency of 15, but is capped to 6 in order to make the plot more readable.)

plication on Android devices, "Gmail" (com.google.android.gm), is investigated in Figure 7.7. From the figure, it is clear that there is a strong dependency between "Gmail" and the different browsers, low dependency with the most popular social applications (investigated earlier), and overall low dependency with other communication applications (with exception of another mail application).

7.3 Location Analysis

7.3.1 Clustering Analysis

The results of the clustering processing will now be presented along with some results on the cluster analysis.

Since the clustering algorithm (`DBSCAN`) takes two parameters (ε and $minPts$), it is clear that these two parameters have a big impact on the outcome of the clustering analysis. This is illustrated in Figure 7.8. Note that the radius of each coloured circle represents ε, and that the black/grey circles are locations categorised as noise cf. Algorithm 4.2.

It is difficult to determine the optimal parameter setting, and for some distributions (users), one setting might prove better than for other distributions, following the discussion in Section 4.4.2.4. Therefore, the parameter setting for each user follows the proposed technique for the relative criteria.

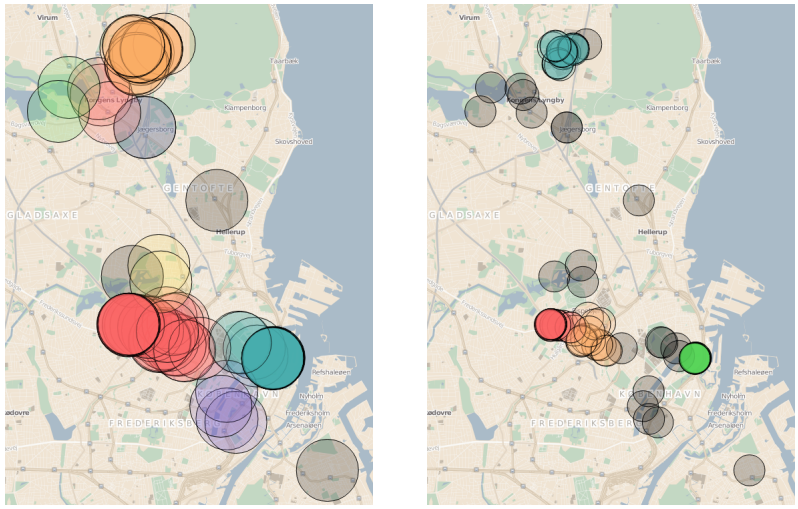


Figure 7.8: Examples of the parameters' influence on the outcome of `DBSCAN` (for user: 4faa5ced9eeb21e307000178).
On the left: $(\varepsilon, minPts) = (1, 2)$, on the right: $(\varepsilon, minPts) = (0.5, 4)$

Looking at specific users, the first one investigated is user: "4ffa3bd76eb92cff1f000005". Running `DBSCAN` with $minPts = 2$ seems to generate the most meaningful results, and hence this parameter is kept static. Varying the ε parameter, a range between $\varepsilon = [0.6, 1]$ seems to keep the number of clusters stable, and the final parameter set is therefore: $(\varepsilon, minPts) = (0.8, 2)$. The Pearson Correlation Coefficients between the resulting clusters is seen in Figure 7.9 in the table on the left, with an illustration of the visual interpretation of the clusters on a map on the right.

Cluster ID	Internal Pearson	External Pearson
Noise	0.54	0.58
1	0.9	0.64
2	0.48	0.55
3	0.75	0.61
4	0.97	0.7
5	0.3	0.48
6	0.1	0.31

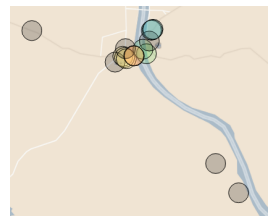


Figure 7.9: Clustering analysis results for user: "4ffa3bd76eb92cff1f000005" using $(\varepsilon, minPts) = (0.8, 2)$ Some locations did not fit in the map.

Cluster ID	Internal Pearson	External Pearson
Noise	0.07	0.12
1	0.31	0.19
2	0.16	0.15
3	0.37	0.23
4	0.56	0.25
5	0	-0.02
6	0	0.02

Figure 7.10: Clustering analysis results for user: "4fb4a1f50806783e160067ac" using $(\varepsilon, minPts) = (1.85, 3)$

Cluster ID	Internal Pearson	External Pearson
Noise	0.37	0.07
1	0.02	0.07
2	0.32	0.09
3	0.56	0.14
4	0.21	0.08
5	0.1	0.04

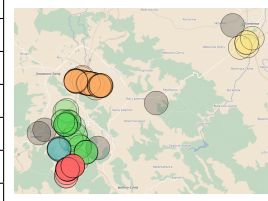


Figure 7.11: Clustering analysis results for user: "4fb29efa0806783e16004f2b" using $(\varepsilon, minPts) = (0.85, 3)$

The same procedure has been used to determine parameters for two other users, and these results are outlined in Figure 7.10 and Figure 7.11. The discussion of the results is found in Section 8.3.2.2.

7.3.2 Location Distribution

To see how users' locations are distributed, the most popular location for each user is investigated. The number of interactions at the most popular location gives an indication of each user's tendency to be within the same area. Summing up these interaction counts, the global trend for all users can be analysed. Looking at the location distribution for all users, it is necessary to be aware of users having very few locations. Therefore, the results presented here, are shown for users with at least 3 registered locations and at least 10 registered locations respectively. Furthermore, the most popular location for each user is investigated with different radii, in order to determine the density around the most popular location. The results can be seen in Table 7.12.

Radius	Amount
0.0 km	22.11%
0.5 km	52.63%
1.0 km	57.40%
1.5 km	61.28%
2.0 km	64.29%
2.5 km	67.54%
3.0 km	69.79%
3.5 km	70.24%
4.0 km	71.64%
4.5 km	72.53%
5.0 km	73.10%

Radius	Amount
0.0 km	20.02%
0.5 km	50.74%
1.0 km	54.89%
1.5 km	59.00%
2.0 km	62.19%
2.5 km	65.63%
3.0 km	68.02%
3.5 km	68.49%
4.0 km	69.96%
4.5 km	70.91%
5.0 km	71.51%

Figure 7.12: Investigation of users' most popular locations for users with at least 3 (on the left) and 10 (on the right) registered locations.

Note that the radius of 0.0 km corresponds to the most popular location. All locations are rounded to 4 decimals as discussed in Section 5.3.3.

Chapter 8

Evaluation

This chapter discusses and evaluates the results described in Chapter 7.

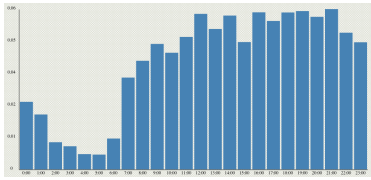
8.1 Time Analysis

The presented results give an overall good indication of the daily and weekly application usage, and thereby of users' usage patterns. For instance, the weather application (on the right in Figure 7.3) is most popular in the morning and evening - probably since this is when people need their daily weather forecast.

The fact that it is possible to generate histograms based on Google Play categories, makes it possible to see how similar applications are utilised. The illustrations in Figure 7.4 illustrate some of the differences among categories, and the fact that shopping applications are primarily used during noon and afternoon might not be obvious to everybody.

Some of these observations might seem obvious, but are nevertheless extremely important to application developers. It is not always obvious when a user will use a particular application - or a specific type of application.

The validity of the results can be hard to determine, however it seems that



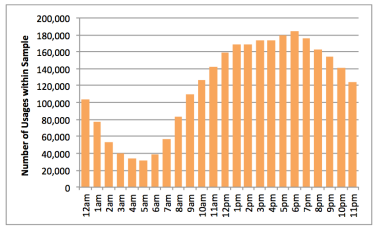


Figure 8.1: Comparison of the results on all users' accumulated application interactions over a day with the results from this project on the left and [BHS+11, p. 51, fig. 3] on the right

many of the basic results are comparable to the results obtained in [BHS+11] - in particular when comparing all users' accumulated application interactions over a day, as seen in Figure 8.1. The small differences between these two distributions most likely relate to the relative small number of users contributing to this project, and perhaps also a difference in demography between the users in the two projects.

8.2 Transition Analysis

The results on application transitions indicate a number of trends applications and application categories in between. The fact that some application categories are more closely related than others is also supported by [BHS+11].

It is evident that both the two social applications ("Facebook" and "Twitter") as well as the mail application ("Gmail") are quite often used as predecessors to browser applications. This is important, as it tells something about how people use these applications, and how often a web page is actually visited because of a link found in one of these applications.

The data seems to be very sensitive to single users' usage patterns. For instance, when investigating the successors to "Gmail", "Twitter" is listed as a successor. However, investigating this relationship, it turns out that the particular transition has only occurred 4 times, making the validity of statistical conclusions based on this transition very low.

The conducted transition analysis is a good tool for finding trends but has it weaknesses. One user can affect the results a lot, and the amount of data means that the number of conclusions which can be made are limited.

8.3 Location Analysis

8.3.1 Location Distribution

The most popular location for each user seems to cover around 20% on average for all users. In order to cover 70% of the registered interactions, it seems that the radius on the most popular location need to be increased to around 3.5 kilometres and 4.5 kilometres respectively for the two thresholds on the number of registered locations. Comparing to the results obtained in [SQBB10], there were no indication of the accuracy they used to determine user's location. However, it seems unlikely that it was as high as 3.5 kilometres. Nevertheless, the data used in this project does not indicate how much time users spend at each location, but only how many times they interact with an application. Therefore, the results cannot be directly compared.

The results obtained do show, however, that users tend to utilise their phone mostly within one specific region, and at least 50% of their interactions are within a 500 metres radius. This gives a clear indication of the location density in the the average user's utilisation of his/her phone. The 70% of the application usage within the 3.5 to 4.5 kilometres radius underline this fact. The majority of all application interaction is within a relatively small area.

8.3.2 Clusters

8.3.2.1 Cluster Creation

For most users, the created clusters seemed to visually fit an expected distinction between their most popular locations - for instance at school and at home. For users with many registered locations however, it is clear that with only two adjustable parameters (for the algorithm) it is hard to separate the many locations properly. In order to build a dynamic solution (which could find the best fit automatically), only data over a limited period should be used.

When users have been travelling it is difficult for the algorithm to consider the transportation route as one cluster, as it is often "connected" to other registered locations particularly at the beginning/end of a route. Using users' velocity as an extra parameter or comparing the locations to actual public transportation lines and roads might prove necessary in order to create this distinction.

8.3.2.2 Cluster Analysis

The results of the few examples which were described (in Figure 7.9 - 7.11), will now be discussed.

For 2 of 3 selected users, the number of clusters with higher internal than external correlation was 4/6 and 3/5 (ignoring noise-categorised locations as a cluster). For the third user, this number was only 3/6. For most of these clusters, the difference between the internal and external correlation was significant. The three users were not picked at random, since some users were simply not fit for the cluster based analysis, as they either had too many (as discussed in the previous section) or too few registered locations.

The internal correlation coefficient does not tell anything about whether the unique number of applications used within a cluster is high or low, or whether the locations in that particular cluster share many applications[1]. It only indicates how well the applications which the locations have in common match in relative usage count.

Based on the results described above, it seems clear that some users' application usage is more alike within certain location based clusters than outside these regions. However, it has not been possible to generate results which indicate this trend for all the clusters created for any user. Either this is not possible with the applied method and the amount of data available, or this means that users' application usage do simply not always follow their physical location. After investigating a large number of users' application usage very closely, looking particularly at the relationship between the number of uses of specific applications, the latter seems most plausible .

The fact that each cluster analysis was performed with different parameters (for the `DBSCAN` algorithm) does not change the validity of the results, as they must merely indicate whether it is possible to find a correlation between distinguishable locations and usage patterns, not whether it is possible over all users with one set of parameters.

8.3.2.3 Method Discussion

Using Pearson Correlation Coefficients as a tool for comparing clusters has its advantages, as it gives a valuable similarity measure which is easy to interpret.

[1] However, two locations needed to have at least two common applications to receive a Pearson Correlation Coefficient higher than 0.

However, it does have some disadvantages. First of all, as it is only common elements (applications used) which determine how alike two locations are, locations with almost no interactions might get a relatively high similarity measure to the most visited locations, as there is a good chance these locations share elements. Furthermore, two popular locations have a bigger chance of getting a lower similarity score, since they have a bigger chance of sharing many elements which might vary a lot in usage count.

8.3.2.4 System Performance

Based on the description of the location visualisation implementation and the overall performance of this visualisations, it is clear that as the amount of data increases, new optimisations need to be implemented. As of now the performance is tolerable and sufficed for generating the necessary results, but if the server had to serve multiple clients and ensure a certain level of performance, more data processing should be done server-side and optimisations would have to be implemented on both the client- and server-side.

Chapter 9

Discussion

This chapter describes some of the issues with the collected data, and reflects on some of the perspectives within the experimental work.

9.1 Data Collection

9.1.1 Measurable Parameters

The collected data allowed extensive analysis of individual application usage, even though the implemented solution only takes the active application into account. This means that information on the application history is neglected, with exception of the latest application. However, longer sequences of applications have already been investigated in [BHS+11], and was not the primary target of investigation in this project.

The information on the running application only gives an indication of which application context the user currently operates in. It does not reveal what the user is currently using an application for (this is not accessible on an Android device), and one application might solve several different tasks. One example of this is the browser, where a web-page can be related to almost any task (it

might even be a web app[1]) and some might run for longer time than others. To investigate users' detailed interactions within an application, it is necessary to embed code in each application and use Google Analytics[2] or similar approaches to track users.

Since information about each user's device was not collected, it is not possible to spot trends within specific kinds of devices (tablets and smartphones isolated). This could have been interesting to investigate, since it can be assumed that tablets are used differently than smartphones.[3]

9.1.2 Data Correctness

Some of the collected data turned out to be faulty, or at least the registered end time of each application usage was not necessarily correct. Nevertheless, the data collected proved significant for several analysis based on application usage count instead of application usage duration. An attempt to remove the faulty data could have been made (e.g. by neglecting all single usage times above half an hour), but this would only remove some of the "noise" as well as potentially removing valid data, and would therefore give little guaranty for the overall validity of the data. This means that all end times have been neglected, and application usage **time** has not been considered for the different analysis - only the start time and the application usage **count** have been used.

In order to solve the issues with faulty end times, several things could be done[4]. One solution would be to use an existing technology for all of the data collection process. One such technology is "funf"[5], which is a framework for building Android applications which uses the built-in sensors to monitor user behaviour; this includes logging the running application(s). If this technology had been available (in its current condition) earlier in the process of this project, this would surely had been taken into consideration when implementing the application monitoring solution. It seems that a lot of the work done with implementing the Android application, as well as building the server solution could have been done much faster using this technology instead of building everything from the bottom up.

[1] Online accessible application, see: http://www.meddb.be/webapplications.aspx

[2] http://www.google.com/intl/en_uk/analytics/

[3] See for instance: http://blog.nielsen.com/nielsenwire/online_mobile/double-vision-global-trends-in-tablet-and-smartphone-use-while-watching-tv/

[4] For instance this implementation: http://android-random.googlecode.com/svn-history/r219/trunk/TestKeepAlive/src/org/devtcg/demo/keepalive/KeepAliveService.java

[5] http://funf.media.mit.edu/index.html

9.1.3 Data Quality

Looking into the number of launches of individual applications, it seems that the daily and weekly usage patterns are quite vulnerable to noise; particularly to single users with a specific usage pattern or a relatively high registration of application usage. This is why the normalised histograms have been used for the results. If more users had been part of the data collection, most of this noise would be neutralised. This would also mean that the validity of the conclusions made on basis of the collected data would be higher, as they would rely on a bigger statistical foundation.

The trends found within the data, however, indicate that the collected data does in fact reflect real usage patterns, especially when comparing it to other sources.

The accuracy of the collected data could have been improved by increasing the frequency of checking for the active application on each device (as of now this is checked every 2 seconds). This however, would have increased the CPU usage time in "Appalyzer", and hence the battery consumption. Every 2 seconds seemed adequate to provide a sufficient level of details in the collected data; especially since the average time for a single application use is known to lie somewhere above 30 seconds, depending on the type of application [BHS+11, p. 50 table 2].

9.2 Recommendations

The recommendation feature of the solution could have been improved a lot, but as this was not the focus of the project, little effort was done on this part. Below follows a discussion of how this could be improved.

First of all, using Pearson correlation as similarity measure has some disadvantages in the given scenario. When finding a recommendation, the item is found based on the highest ratings among users similar to the user at hand. This means that the applications that will be recommended most frequently, will be the most popular applications - being the home screen applications. Since there are many different kinds of home screen applications, a user might be recommended this kind of application several times in a row - which is not desirable.

Moreover, a user does not necessarily want a recommendation for an application which is too similar to an application recently used. On smart phones, applications are small individual components - each of them solving a simple dedicated

task. Therefore, a recommendation of an application too similar to an existing application might not be very popular with the user.

Implementing a more clever algorithm which could take application category into account, and look at previously recommended application categories would be preferable.

To properly recommend applications to users, it is also necessary to have some more information on the device (at least on the Android platform):

- The country the device is located in
- The version of Android the device is running
- The screen size or model number of the device

Accordingly, the following information must be available on every application in the database (for the recommendation system):

- Is the application available on Google Play, or is it a developer version of an application?
- In which countries is this application available?
- Which devices does this application support?
- What is the "Content Rating"[6] of the application? - assumed that a single user (perhaps a child) must only receive recommendations of low maturity

All of this information is necessary to ensure that when recommending an application, this application is actually applicable with the specific user's device. This is the case, since when publishing an application on Google Play, it is possible to limit it to specific API levels (Android versions), specific countries or certain device screen sizes (e.g. only tablet sizes).

The best recommendation experience seemed to be with the "Appolicious" application mentioned in Chapter 2 on page 5. This particular application looks at all the installed applications on a device, and recommends alternatives to the installed applications based on user ratings of the different applications. A combination of this approach combined with a more scientific approach, as the one

[6]Indicates if an application includes sexual or violent references, or if it uses the user's location, see: http://support.google.com/googleplay/android-developer/support/bin/answer.py?hl=en&answer=188189

proposed in [LKKK11], would seem as the most likely to succeed as a serious application recommender. However, it needed to ensure knowledge of all the prerequisites mentioned above, in order to recommend applicable applications only.

To sum up, it seems that in order to build an optimal recommendation system, a proper analysis of the required prerequisites along with a close connection to Google Play should be conducted. Building a complete system would have to take the limitations on locale, device screen size etc. into account from the very beginning in order to avoid unavailable recommendations. Furthermore, the choice of recommendation system should be a category-based system with emphasis on finding similar applications, which can replace each other. The choice of recommendation algorithm should probably be an item-based algorithm with lower memory consumption and where the runtime is not affected by an increasing number of users, as the one proposed in [SKKR01].

9.3 Prospects

9.3.1 Area with Focus

Analysing users' behaviour in different contexts have recently received a lot of attention from some of the biggest companies within mobile application development. Below is a description of some of the different initiatives. Google Play[7]

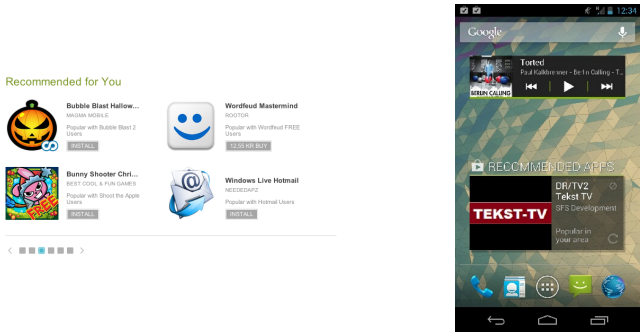


Figure 9.1: Personalised application recommendations from Google Play. From the web-version on the left and from their widget on the right (the bottom widget).

[7] https://play.google.com/store

recently started recommending applications for (Android) users based on various parameters (see Figure 9.1 to the left) on their web-site. Since Google have information on users from various contexts and media (e.g. mail, installed Android applications, +1-indications[8], friend's +1-indications, users' location etc.) they have a solid base for recommending applications. Until recently they only utilised this information for recommendations on their web-based Play Store[9], but with the new version of Android (Jelly Bean, 4.1) they also start recommending applications on their Android-based Play Store cf. the quote below.

> A new set of recommendations widgets use a variety of signals — content that people with similar tastes have purchased, stuff that's popular around where you live, content people in your Google+ circles have +1'ed, and more — to recommend new content like apps, games, music, and movies.[10]

A screenshot of the way they currently utilise this (with a widget) is seen in Figure 9.1 on the right. It is very likely that this development will eventually remove the need for other dedicated recommendation applications (on Android).

Apple and Facebook have started focusing on some of the social contexts within users' application usage. Apple have recently integrated Facebook in their appstore, making it possible to "like" apps.[11] This shows that they are trying to put users' application preferences into a social context which might possibly allow them (or other developers) to come up with application recommendations based on social relations.

Facebook are planning allowing companies to put in advertisements (for instance for other mobile applications) in the official Facebook application. These ads should apparently be based on users' individual application usage, making them as well-fit for the particular user as possible.[12] Furthermore, they try to be a part of the growing mobile application industry by launching "the App Center", giving application developers a new platform for application marketing.[13]

One of the reasons why these companies need to use custom measures to get indications of users' appreciation of applications, is due to the fact that only a small subset of users actually rate applications (in Apple's App Store and on Google Play). A few users might represent the majority when they rate

[8] Similar to Facebook's "like" feature - see: http://www.google.com/+1/button/
[9] http://tiny.cc/41ouhw
[10] http://www.android.com/about/jelly-bean/
[11] See: http://marketingland.com/apple-integrates-facebook-into-ios6-13847
[12] See: http://tiny.cc/xgmiiw
[13] See: http://tiny.cc/7fmiiw

applications, but if the users who rate are always the same, the ratings might not reflect the average opinion. As an example, at most 0.4% of the users having installed the `GMail` application from Google Play have actually rated it.[14] For developers of some of the most popular applications 0.4% might suffice, but for developers of applications with a smaller potential number of users, this might not be adequate as they would only start getting ratings once they reached 250 users (assuming a linear tendency in ratings compared to the number of downloads). By utilising several other ways of measuring application popularity (Facebook likes, Google +1's, etc.), users, as well as developers, will potentially get better estimates of application similarity, popularity, and quality.

9.3.2 Reflections on Utilisation

Based on the results obtained in this project and the tendencies within the area described above, a number of potential uses of the presented approach can be identified.

First of all, the data collected would, as earlier mentioned, be useful to many developers when developing new mobile applications. It would be relevant to consider building a publicly available API for these developers to connect to, which could present statistics on specific applications, specific categories, and application usage within regions or specific time periods. This could be used by developers to gain insight into usage patterns, but might also be used directly as back-end web services to certain kinds of applications.

More advanced applications similar to "Appalyzer" could also be build with even more emphasis on presenting the user with personal statistics and visualisations, and also include the possibility for categorisation and more customisation. This would not necessarily have to recommend new applications (as it seems the Google Play approach will be quite powerful), and could work without ever having to connect to the internet.

As mentioned in Section 1.3, it would be interesting to implement a widget which was able to recommend installed applications with background in the data collected by "Appalyzer". Such a widget might use several measures to calculate the most plausible application which the user would launch, based on time of the day/week, the user's location and the previously opened application(s). This would also allow an exploration of the area of predicting users' behaviour and not only their locations, investigated in [SQBB10].

[14] See: https://play.google.com/store/apps/details?id=com.google.android.gm. As of July 23rd 2012: 378,698 ratings and at least 100,000,000 downloads

Finally, as earlier mentioned, there is a great potential for using this kind of information for advertising in applications, as it would be possible to estimate the most likely transitions to the next application, and therefore present an ad with a similar application.

Chapter 10

Conclusion

10.1 Future Work

As mentioned in Section 1.3 and Section 9.3.2, this project has some interesting aspects for application developers. In order to build a publicly available API as suggested, several things would have to be done; these are listed below.

- A server with more memory should be used
- The stability of the server should be optimised as well as the database performance
- A requirements analysis should be conducted, revealing the actual numbers which developers would need
- More data would have to be collected

In order to collect more data, the Android application would also have to be improved. The following things would be relevant to optimise.

- Remove the recommendation feature

- Add a categorisation features, allowing users to collect applications in custom groups and perhaps to categorise locations as “home”, “work” etc.
- Resolve the issues regarding end times which are not registered correctly
- Base the application on the “funf” framework
- Provide users with even more visualisations and insights in their application usage, for instance by using clustering, and transition analysis on the phone
- Emphasize privacy even more, and encrypt the collected data as well as the connection to the server

Accordingly the application would also have to be promoted in various ways in order to reach a higher number of potential users.

Together, these steps would form an interesting approach which could build a better and bigger data set with more correct data. This would also allow an analysis of the application interaction duration, as the end time registrations would be correct.

10.2 What has been accomplished

A number of significant trends within users’ mobile usage patterns have been obtained. First of all, using location based clustering, it has been possible to verify that application usage patterns are more similar within distinguishable regions than outside of these. Furthermore, the distribution of users’ locations seem to follow a pattern, where more than 50% of all interactions on average are within 500 metres of the location with most registered interactions.

Daily and weekly trends have also been found and validated within specific application categories, and it has been possible to detect frequently occurring interactions between single applications and application categories.

These findings have been made possible through the implementation of a mobile application prototype as well as a server, receiving data. The mobile application was able to collect users’ application usage without impacting each device’s performance or battery consumption significantly.

A thorough analysis of a recommender system for mobile applications has been conducted and, to conclude, the different measures presented in this project are

not well fit for recommending new applications without building a large system similar to Google Play. However, they provide a solid base for recommending already installed applications to users, as it is possible to utilise trends within a user's location, application transitions and application usage history.

Some of the results, especially regarding application usage over time, could be collected on individual applications by developers themselves if they used Google Analytics or similar approaches in their applications. However, when developing a new application it might not be intuitive when different application types are used during a day. This is why a publicly available API, making all of this data available, has been suggested as the next step. This would allow developers to benefit from data on multiple application and application types before implementing a new feature or a completely new application.

Based on the above, the overall goals of this project have been fulfilled, and a better understanding of users' usage pattern has been achieved through various kinds of data analysis.

Appendix A

Mobile Application Interface

Detailed description of the implementation of the interface on the Android application "Appalyzer".

A.1 Menu Layout and Interaction

As described in Section 4.1.1, an experimental layout was chosen for the menu navigation in the application. A pie-chart-looking button with three "pies", each representing a single button, was chosen. This decision however, led to one big issue: *Standard Android buttons are rectangular* (see Figure A.1). This both meant that the bounding box of each part of the pie-chart would not correspond to the visual interpretation, and that the three buttons would overlap when placed in a `View` component. The overlap itself was not a big issue, visually, since transparent backgrounds of the three parts would make it look just fine; the bounding box issue was more difficult to handle, as a press on on button might mean that another button had been pressed.

The solution was to build a custom `View`[1] component which extended the

[1] http://developer.android.com/reference/android/view/View.html

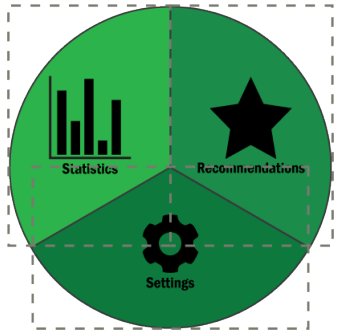


Figure A.1: Illustrating difficulties with non-rectangular buttons in Android

`Button`[2] class, and took care of touch events for all three buttons in the pie-chart. One could also have used one custom `Button`-object for each button, but as earlier described, this would mean that one button would have to invoke an event on another button. The implemented class (`MyButton`) handles multiple tasks:

- It changes the visible `Drawable` to reflect a press
- It recognises which button was pressed
- It fires the corresponding event when a button is pressed
- It can have its set of drawables changed to a new set of drawables, to show a new menu

To identify which (part of the) button is pressed, the class overrides the `onTouchEvent()`-method, and, if the pressed point is within the circle's radius, calculates the angle between the vector from the centre of the circle to the pressed point and a unit vector $\begin{pmatrix} 0 \\ -1 \end{pmatrix}$. As the circle is divided into three parts (each filling 120°), it is easy to calculate in which part the pressed point was intercepted.

If the `MotionEvent`[3] is of the type `ACTION_DOWN`, the `Drawable` of the button must be changed to a drawable reflecting that the corresponding part of the button is pressed. If the type is `ACTION_UP` and the up-event happened within

[2] http://developer.android.com/reference/android/widget/Button.html
[3] http://developer.android.com/reference/android/view/MotionEvent.html

the same button-part as the down-event, the event is recognised as a button press.

If an event is recognised as a button press, the class will fire a custom `Event`. The implementation of custom events and `EventListener`s (the observer pattern[4]), is a powerful way of communicating between view components in an Android application, and it maintains a good relationship between observer(s) and an `Object`, where the object (`MyButton`) does not need any information or other relationship to the observer (`MenuActivity`)[5].

The specific implementation is quite simple, and must only handle a single observer, which will then check the `Tag` of the `View` at hand to see which state it is in. Therefore, the implementation does not need handling of listener removals or thread safety which might otherwise be of concern for the specific pattern [Goe].

Finally, a button press might mean that the custom button should display a new set of drawables. This is done by passing it the four new drawables, after which it will update itself with an animation to indicate a change in the menu state.

A.1.1 Asynchronous load of data

Everywhere when data has to be loaded from the database, it must be done asynchronously in order to give the user the best possible experience. Therefore, when either of the three main visualisations (in the activities: `StatsActivity`, `PieActivity` and `MyCustomMapActivity`) have to access the relevant data, they make use of an `AsyncTask`[6] to load the data in the background while displaying a `ProgressDialog`[7] to let the user know that the application is actually busy. This is usually a good idea, in order to keep the user well informed and avoid impatient users [Tid05, Ch. 5, pp. 149–150].

[4] See: http://www.dofactory.com/Patterns/PatternObserver.aspx

[5] "When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled." [GHJV94, p. 327]

[6] http://developer.android.com/reference/android/os/AsyncTask.html

[7] http://developer.android.com/reference/android/app/ProgressDialog.html

A.2 Data Visualisations

All of the three visualisations support the filtering mechanism presented in Section 4.1.1, which allows an intuitive way of showing only a subset of applications for each visualisation. The filtering in one visualisation is reflected in the other two whenever a change is made. Below is a brief presentation of some of the most important details from each of the three visualisations.

A.2.1 "Stats"

The `Activity` handling this visualisation, presents the user with a list of apps, and the time consumption of each application. This is done in a `ListView` with a custom adapter extending the `ArrayAdapter`. The data is fetched (asynchronously) from the database.

A.2.2 The Pie Chart

The pie chart has been implemented, using the `AchartEngine`[8], which provides an easy way of building various charts. Technically, a `.jar`-file from achartengine had to be included in the Android project, after which it was possible to use a `ChartFactory` from this library to build the chart. The data for the chart is extracted from the database, and all applications with a total rate below 0.7 % are put into one group of applications called "Other" to make the visualisation free of too much noisy information.

A.2.3 The Map

The map activity has been implemented following the basic rules for a `MapActivity`[9], and the data displayed (as coloured dots) is extracted from the database and put on the map in an `ItemizedOverlay`[10].

[8] A charting software library for Android applications, capable of building various visualisations of data, including: pie charts, graphs, and bar charts developed by "The 4ViewSoft Company". See: http://www.achartengine.org/

[9] https://developers.google.com/maps/documentation/android/reference/com/google/android/maps/MapActivity

[10] https://developers.google.com/maps/documentation/android/reference/com/google/android/maps/ItemizedOverlay

Appendix B

Tests

Below, the background for the conducted tests as well as these tests is described.

B.1 Mobile Application

A full-blown test of an Android application requires several steps. Not all of them have been conducted, considering the scope and focus of the project, but here follows a general overview of testing Android applications, and a summary of how "Appalyzer" was tested before release.

B.1.1 Test Background

There are different ways of testing an Android application, and the ultimate test is most often to have end-users test the application. The general rules for testing an application have been given by the Android Development Team, and describe

both what to test[1] and how to test this[2]. But several other testing techniques exist. Several frameworks for black-box testing exist (e.g. "Robolectric"[3] and "Robotium"[4]), and among these, some even make it easy to test an application on multiple different devices at once, for instance "Lesspainful"[5].

Since Android applications are written in Java, unit testing[6] is also easy to do, using the standard "JUnit"[7] tool. The most thorough and intuitive test of an application however, is to operate it manually on a real device to test different things such as:

- Screen resolution – Does the UI look as expected on the specific device?
- Screen orientation – How does the application correspond to orientation changes, and where they handled properly in the application Manifest/-code?
- Touch screen operation – Does the interaction on a touch screen feel natural/intuitive?
- CPU and network performance – When the application is not run in an emulator, how well does it perform on a smaller CPU, and how (well) does it handle network changes?

[RLMM09, Ch. 7.1]

For deeper analysis on the application behaviour, it is relevant to use a debugging tool such as DDMS[8], which allows advanced debugging, memory allocation analysis, and network traffic investigation. This is especially good for testing for memory leaks in an application.

[1] Short list of general rules for what to test in an Android application. See: http://developer.android.com/guide/topics/testing/what_to_test.html

[2] Testing fundamentals for an Android application, describing best practises and how to use JUnit and monkeyrunner in Eclipse. See: http://developer.android.com/guide/topics/testing/testing_android.html

[3] Black-box testing framework, allowing test of application features without having to deploy the application to an actual device/emulator. See: http://pivotal.github.com/robolectric/

[4] Black-box testing framework, allowing test of application features on device, see: http://code.google.com/p/robotium/

[5] Automated application testing for Android and iOS on multiple real devices. See: https://www.lesspainful.com/

[6] IEEE Standards Board: http://aulas.carlosserrao.net/lib/exe/fetch.php?media=0910:1008-1987_ieee_standard_for_software_unit_testing.pdf

[7] http://junit.sourceforge.net/

[8] http://developer.android.com/guide/developing/debugging/ddms.html

B.1.2 Conducted Tests

In "Appalyzer", only a subset of the mentioned tests were conducted. As launching the application as soon as possible was the number one priority it was not tested thoroughly on all devices, screen resolutions etc. However, it was tested on 4 different devices. On-hand tests were carried out by multiple users, and the feedback was used to improve the design and performance of the application even more. For deeper debugging, the built-in debugging tool in Eclipse[9] was used extensively, and the previously mentioned DDMS-tool was also used to profile the applications memory consumption.

B.2 Server

Testing a web server is often about testing the performance of the server, once the number of connections increases [Kil02, Ch. 3]. Testing the server performance has not really been a priority, as it was assumed that the server used would only have to serve a relatively small number of users (less than 100).

The correctness of the server however, has been tested thoroughly to ensure data integrity and meaningful application recommendations for clients. The recommendation part was tested with dummy data from a completely different context, namely movie ratings. The recommendations are based on the users' "ratings" (usage counts) of applications, and this feature is comparable to a movie rating.

B.3 Data Analysis and Visualisation

The data visualisations have been tested in Chrome on a 2.4GHz Macbook Pro running Mac OS X 10.7.4 and there has been no testing of the resulting appearance on different devices or screen sizes. The primary debugging tool has been outputting to the console.

The validity of the data used for the visualisations has been checked through database investigation as well as thorough testing of the database lookups.

[9] http://eclipse.org/

Bibliography

[Abl] Frank Ableson. http://www.ibm.com/developerworks/web/library/x-andbene1/index.html?ca=drs-. Introduction to, and analysis of XML- and JSON-parsing in Android.

[AEK00] Asim Ansari, Skander Essegaier, and Rajeev Kohli. Internet recommendation systems. *Journal of Marketing Research*, 37(3):363–375, 2012/06/17 2000.

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. Towards the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17, no. 6, 2005.

[BHS+11] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47–56, New York, NY, USA, 2011. ACM.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[EpKSX96] Martin Ester, Hans peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994.

[Goe] Brian Goetz. http://www.ibm.com/developerworks/java/library/j-jtp07265/index.html. Discussion on the observer pattern and discussion of different thread safety issues.

[HBV01] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17:107–145, 2001. 10.1023/A:1012801612483.

[Kil02] P. Killelea. *Web Performance Tuning*. O'Reilly Series. O'Reilly, 2002. A book on optimising web performance, both regarding web servers, web content, and browsers.

[KKSZ11] Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3), 2011.

[Lec07] Julien Lecomte. http://www.julienlecomte.net/blog/2007/10/28/, 2007. Description and discussion of pattern from javascript-development, ensuring browser responsiveness while performing intense computations.

[LKKK11] Yujin Lim, Hak-Man Kim, Sanggil Kang, and Tai-hoon Kim. Recommendation algorithm of the app store by using semantic relations between apps. In Tai-hoon Kim, Hojjat Adeli, Rosslin John Robles, and Maricel Balitanas, editors, *Ubiquitous Computing and Multimedia Applications*, volume 151 of *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-20998-7_18.

[LM08] Daniel Lemire and Anna Maclachlan. Slope One Predictors for Online Rating-Based Collaborative Filtering. September 2008.

[Nok] Nokia. http://research.nokia.com/page/12000. The nokia data challenge, which was a project where a lot of students got access to data collected over more than a year by more than 200 devices.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[PB07] Michael Pazzani and Daniel Billsus. Content-based recommendation systems. In Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *The Adaptive Web*, volume 4321 of *Lecture Notes*

in Computer Science, pages 325–341. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-72079-9_10.

[PYC06] Han-Saem Park, Ji-Oh Yoo, and Sung-Bae Cho. A context-aware music recommendation system using fuzzy bayesian networks with utility theory. In Lipo Wang, Licheng Jiao, Guanming Shi, Xue Li, and Jing Liu, editors, *Fuzzy Systems and Knowledge Discovery*, volume 4223 of *Lecture Notes in Computer Science*, pages 970–979. Springer Berlin / Heidelberg, 2006. 10.1007/11881599_121.

[RLMM09] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., 1st edition, 2009.

[RRS11] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. *Recommender Systems Handbook, Springer*, 2011.

[Seg07] T. Segaran. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Series. O'Reilly, 2007.

[SKKR01] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *World Wide Web*, pages 285–295, 2001.

[SQBB10] Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of Predictability in Human Mobility. *Science*, 327(5968):1018–1021, February 2010.

[Tid05] J. Tidwell. *Designing Interfaces*. O'Reilly Series. O'Reilly, 2005.

[TV10] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 2010.

[XZL10] G. Xu, Y. Zhang, and L. Li. *Web Mining and Social Networking: Techniques and Applications*. Web Information Systems Engineering and Internet Technologies Book Series. Springer, 2010.

[ZHLP10] Nima Zandi, Rasmus Handler, Jakob Eg Larsen, and Michael Kai Petersen. *People, places and playlists: modeling soundscapes in a mobile context*. 2010.