M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Master's Thesis

A Flutter Package for Real-Time Mobility Feature Computation

Thomas Nygaard Nilsson, s144470

Kongens Lyngby, Denmark 2020

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

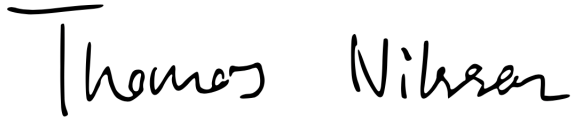
Summary

Mobility features derived from location data can be used to describe behaviour changes in people suffering from depression-related diseases. Existing contributions within the field of computing mobility features in smart-phone environment are cumbersome, if even possible, to reproduce with regards to their source code. Furthermore, the algorithms for computing these features in the literature do not always lend themselves to be computed in real-time. These two research problems were addressed in this thesis for which the solution was a software package implemented in the Flutter framework. Through the package, an application programmer is able to compute mobility features with just 3 lines of code. Furthermore the package allows the application programmer to flexibly choose their own Flutter plugin for tracking location data to avoid dependency issues with other plugins. A field study was conducted where a mobile application running the package was used by 10 participants over 3 weeks. Participants filled out a daily questionnaire pertaining to the features, these answers were then compared to the computed features. When comparing the daily location features with subjective user data we found that the Mobility Features Package predicts the Number of Places visited with an RMSE of 0.5 places, the Home Stay percentage with an RMSE of 14.3% and the Routine Index with an RMSE of 22.5%. However, many non-uniform gaps were observed in the collected location data which impacted the results of the study. For future improvement it will be highly relevant to use an imputation method to mitigate the issue of missing data.

Preface

This Master's thesis was prepared at the department of *Applied Mathematics and Computer Science* at the Technical University of Denmark in fulfillment of the requirements for acquiring a MSc degree in *Human-Centered Artificial Intelligence*.

Kongens Lyngby, Denmark, June 25, 2020



Thomas Nygaard Nilsson, s144470

Acknowledgements

I would like to thank by supervisor Jakob Bardram for providing the initial idea for the project and for guiding me through the project with regular supervision sessions. In addition a big thanks to Jonas Busk for providing Python source code prior to starting the project, this helped the thesis a great deal in succeeding.

It was planned that I should write most of my thesis while psychically sitting at the chair of Information Systems (Krcmar) at the Technical University of Munich. However due to the Coronavirus outbreak in early 2020, I had to travel home to Denmark in March. However, I would like to give a big thanks to Georg Groh and Martin Lurz and Jakob Bardram for making this quasi-exchange semester possible. An additional thank you goes out to Georg Groh for introducing me to Brazilian Jiu Jitsu while I was in Munich.

For conducting the study I would like to thank Jakob Eg for addressing my questions regarding the subjective data collection and which challenges they faced in the study conducted by Cuttone et al. [CLL14] where Jakob was a co-author.

For helping with the thesis writing I would like to thanks Jakob Bardram as well as Darius Rohani and Marie Mørch for reading and correcting my thesis which improved it a great deal.

Contents

CHAPTER 1

Introduction

This chapter will provide an introduction to the thesis by laying out the motivation and background as well as the problem statement and the goals which were set out to be achieved.

All source code used in this thesis is publicly available on Github[1].

1.1 Context and Motivation

It has been shown that changes in a user's location can reveal a lot about their behaviour and mental state. Location data collected from a user's smartphone is transformed into daily metrics called *mobility features* that are highly correlated with the users mental state [Sae+15b; CM15]. Coherence between features derived objectively from a users phone and their mental state - as reported through questionnaires - could be a vital step to support psychotherapy components in the clinic.

Numerous contributions exist within the field of using smartphone data for predicting the various states of the user. It is however not common practice for researchers to publish their source code such that it may be used by other researchers in the future. In addition, most research in the field of mobility data (see [Sae+15b; Sae+15a; CM15; Dor+18]) seems to develop for the Android platform exclusively since this platform is more prevalent in most countries[2]. This means even if the code was released, researchers interested in using the iOS platform would have to rewrite the source code more or less from the ground up. Furthermore, source code tends to get deprecated over time, meaning maintenance by the researcher is required.

Furthermore, it is common practice to gather all of the data first and then perform feature computation and data analysis together afterwards [Sae+15b; Sae+15a; CLL14; Dor+18]. This means that features are not computed when the mobile application is used 'in the field' which is when features can be used for interventions. Some features rely on historical data being available which also complicates real-time computation, since historical data will need to be stored and computed on a resource constrained device. As a consequence, researchers within mobile-health using this package will be able to move their focus from away from software engineering to focus on data

[1] `https://github.com/thomasnilsson/mobility-features-thesis`
[2] `https://gs.statcounter.com/os-market-share/mobile/worldwide`

analysis and study design.

This thesis describes the design and implementation of the *Mobility Features Package*, a software package capable of computing mobility features in a smart-phone application. The research problem is focused on the software engineering aspect of developing such a package and will address the lack of re-usable source code in research and lack of support for on-device, real-time feature computation. By real-time we refer to on-device computation at any time of the day, that is sufficiently quick to not hinder the application.

1.2 Background

Major Depressive Disoder (MDD), commonly known as depression, is a serious illness that is currently treated with medication or through therapy. Dobson et al. [Dim+06] discusses how medication is the current treatment standard for severe cases of depression, with cognitive therapy, a form of psychotherapy, being the most widely investigated treatment for more moderate cases. Dobson et al. also discusses how not all patients wish to be medicated and cognitive therapy has not demonstrated efficacy across trials. Furthermore, traditional psychotherapy is resource-intensive to treat as it requires traditional methods that involves face-to-face consultations face to face consultations. Chartier and Provencher [CP13] describe the need to identify effective, evidence-based treatments that are time and cost-effective in an effort to increase the population's accessibility to treatments. Here, low-intensity interventions, such as guided self-help treatments, hold promise for the dissemination of evidence-based treatments. The treatment form of *behavioral activation* is a component of cognitive behavioral therapy, is receiving increased attention and empirical support as a stand-alone psychotherapy method for treating depression. Dobson et al. found that cognitive therapy was outperformed by behavioral activation which performed similarly to antidepressant medication. Furthermore, the findings were also that behavioral activation is less expensive and longer-lasting alternatives to medication in the treatment of depression. Gravenhorst et al. [Gra+15] back up this claim stating how mobile phones, in particular in the area of mental disorders, can provide better treatment to more users with a lower cost.

Mobile sensing involves the use of the phone's sensors to collect objective user data and may include predicting the user state from this data. Objectivity of the data stems from the data being collected from sensors in contrast to subjective user data such as questionnaires. Mobile sensing has become especially prevalent the last few years due to the rise of ubiquitous computing. Applications of mobile sensing include mental health monitoring also known as *digital phenotyping* or *user context generation*. Mobile sensing within healthcare is often abbreviated *mHealth*. Within mental healthcare, mHealth opens the potential for practical and low-cost solutions for psychological interventions [Ebe+17]. In addition, mHealth also enables mental

healthcare to reach populations that do not have access to traditional psychotherapy [Moh+13], such as developing countries. Within the context of mHealth for behaviour and mental state, location data is commonly used. Rohani et al. [Roh+18] found in a review of 46 studies that several of them showed consistent and statistically significant correlations between objective behavioral features collected via mobile and wearable devices and depressive mood symptoms. In addition they conclude that continuous- and everyday monitoring of behavioral metrics could be a promising supplementary objective measure for estimating depressive mood symptoms. As behavioural activation requires an understanding of the user's mental state and behavior it is an ideal use case for using objective mobility features.

1.3 Research Question

Existing contributions within the field of generating mobility features are cumbersome to reproduce due to a lack of publicly available source code. Furthermore, the algorithms that exist in the literature have not been considered for real-time computation which has implications on algorithm design and resource constraints. How these two things can be achieved will be investigated in this thesis and are addressed in the following research questions:

Question 1: *Which mobility features are relevant to include in a software package?*

Question 2: *How can these features be computed in real-time, on a smartphone device?*

Question 3: *How does the design of such a software package look like?*

1.4 Goals and Methods

The research goals closely match the research questions and will explain how the different sub-questions are answered, and which methods are used to do so.

Goal 1: Implement the Off-line Mobility Feature Algorithms

Saeb et al. [Sae+15b] and Canzian et al. [CM15] describes a list of mobility features, some of which they prove to strongly correlate with depressive behavior. These features need to be implemented in their most simple form, i.e. off-line, where all the location data for a given day is readily available. This will be carried out in Python with a synthetic dataset and later in Dart. This is done in order to demonstrate

the quality of the algorithms. A pre-processing algorithm inspired by Cuttone et al. [CLL14] for finding certain features was used.

Goal 2: Adapt Algorithms to Work in Real-time

The features described above will be implemented such that they can be evaluated on a partial dataset, i.e. an incomplete day's data. Many of the features will have minor changes applied to their calculation and a larger effort is needed to ensure real-time capabilities. This adaptation will be carried out in Python.

Goal 3: Implement Real-time Mobility Features on the Smartphone

An procedure that runs the mobility feature algorithms on a smart-phone has to be written which involves a couple of steps. Location data has to be collected and stored on the smartphone, and then it has to be demonstrated that features can be computed whenever they need to be. However, storing raw data every day would end up taking up too much disk space and is therefore not an ideal solution when working with a smartphone. This means the solution has to be able to compute these history-dependent features without relying on storing raw data of each day. This implementation will be carried out in the Dart programming language.

Goal 4: Design and Release Flutter Software Package

The software package should be designed to perform feature computation in a way that is easy to use for an application programmer. In addition this package should have proper documentation and be released on the Dart package manager [3].

Goal 5: Develop a Demo Mobile Application

To demonstrate how the software package containing the algorithm will be used in practice, a Flutter demo app will be developed in parallel with the software package. By developing an actual application and seeing the algorithms in use, a which is easy to use from an application programmer's perspective will emerge.

Goal 6: Conduct a Field Study

To test the quality of the mobility features, a small scale study of 5-10 participants is planned to run for at least 2 weeks. The participants will use the demo application in which they will have to fill out a small diary in order to evaluate the features computed by the algorithms. Participants should be reminded daily to fill out the diary such that as much user data is collected as possible.

[3] `www.pub.dev`

1.5 Results

The Mobility Features Package was implemented in the Flutter framework and published[4]. Through the package, an application programmer is able to compute mobility features with just 3 lines of code. From the work of Saeb et al. [Sae+15b] and Canzian et al. [CM15] the features *Home Stay, Number of Places, Distance Travelled, Location Variance, Entropy, Normalized Entropy,* and *Routine Index* were chosen to be included in the package. A set of additional location features, e.g. stops, places and moves, were included as well, inspired by the work of Cuttone et al. [CLL14] and Canzian et al. [CM15]. Features are computed in real-time using historical stops and moves stored on the phone which lowers the storage requirements greatly compared to using raw location data for feature computation.

The package does not depend on any specific location plugin due to its design, which allows the programmer to flexibly choose their own plugin for tracking location data. Being independent of any specific location plugin enables easier maintenance and allows the package to be used among other packages dependent on location tracking, without causing dependency issues. Through a field study with 10 participants, the capabilities of the package were demonstrated. For this study a Flutter application collected the participants' location for 3 weeks and used the package to compute the participants' features multiple times daily. Participants also filled out a daily questionnaire pertaining to the features, which were compared to the computed features. In the 3-week study, the following insights concerning the Mobility Features Package were drawn:

- The Mobility Features Packages successfully allowed mobility features to be computed several times a day.
- Non-uniform gaps were observed in the collected location data (as in similar 'in the wild' studies) which reduced the accuracy of the computed features. These gaps are likely due to the operating system on the smart-phones throttling background tracking.
- When comparing the daily location features with subjective user data we found that the Mobility Features Package predicts the Number of Places visited with an RMSE of 0.5 places, the Home Stay percentage with an RMSE of 14.3% and the Routine Index with an RMSE of 22.5%.
- The algorithms tends to undershoot in predicting the number of places and home stay, likely due to gaps in the data. There is no consistent over- or undershooting done when it comes to computing the Routine Index, which likely stemmed from gaps in the location data, as well as variance the validity of the subjective answers regarding the participants routine.

[4] `https://pub.dev/packages/mobility_features`

The thesis lastly discusses different approaches to mitigate the errors in computing features, i.e. how the algorithms can be improved and how gaps in the data can be handled in the future.

1.6 Thesis Overview

The process of developing the Mobility Features Package is documented in the following chapters for which a brief overview is given below.

Chapter 2: Related Work will outline a set of relevant contributions related to behaviour and mental state tracking through mobile sensing.

Chapter 3: Theoretical Background will describe the mathematical theory behind computing the features. This includes algorithm design and definitions of all the features in mathematical terms. This section lays the ground work for the later implementation in any programming language.

Chapter 4: Software Design will discuss the design choices and principles used to model a software system capable of computing the mobility features. Design pertains to both how the internal system of the package works as well as the public facing API which is the interface the application programmer talks to when using the package.

Chapter 5: Flutter Implementation will describe how the domain model and the feature algorithms were implemented in the Dart language. How a user may use the final version of the package is also described.

Chapter 6: Validation will outline the field study that was conducted in which participants were tracked with a mobile application that used the Mobility Features Package to compute daily features based on their location data. Implementation of the mobile application is also discussed in this chapter.

Chapter 7: Results and Discussion will include a data analysis of the study in which the computed features are compared with the subjective answers given by participants. Additionally, further work is discussed including improvements to the package and future integration into other projects.

Chapter 8: Conclusion will report the major findings of the thesis and answer the research questions in detail.

CHAPTER 2
Background and Related Work

Providing a software package for computing mobility features is a novel idea which does not exist in the literature. What does exist however are contributions describing how a user state used for phenotyping can be inferred from location data as well as contributions mobile sensing frameworks for digital phenotyping. The work of this thesis lies somewhere in between these two topics.

2.1 Mobile Sensing and Digital Phenotyping

The work by Insel [Ins18] deals with the topic of collecting and aggregating user data into a so-called *digital phenotype*. It is predicted that by 2050, the biggest impact in psychiatry and mental health will have been the revolution in technology- and information science. Smartphones have become ubiquitous in the past decade and there are over three billion smartphones with a data plan worldwide, each of which has computing power which surpasses supercomputers of the 1990s. In areas around the world without access to resources such as clean water and food, ownership of a smartphone and access to information has become a symbol of modernity.

In the realm of psychiatry, a current data collection problem is the dependence on self-reporting of sleep, appetite, and emotional state, even though it is recognized that depression will impair people's ability to remain objective in assessing their own behavior and thus data is prone to be faulty. Another problem discussed by Insel is how people suffering from mental illness tend to not seek help before it is too late. Depressive relapses are therefore also often reported with considerable delay for patients currently in treatment. The smartphone offers an objective form of mental-state measurement which uses built-in sensors such as geo-location, accelerometer, and human-computer interactions (HCI) to infer the state of the patient. This makes it possible to assess people by using data in a real-time fashion, rather than in retrospect as is currently is done. Digital phenotyping could in theory fill the role of a smoke detector which provides early signs of relapse and recovery, without replacing the face-to-face consultations entirely. In addition, this also allows researchers to track patients in their own environment, rather than in a clinical environment.

2.2 Inferring User State from Location Data

Mobile sensing is the ability to unobtrusively collect sensor data from built-in phone sensors, and within the realm behaviour and mental state inference, location data is commonly used. The work done by Saeb et al. [Sae+15b], Canzian et al. [CM15] and to some extend Cuttone et al. [CLL14] forms the basis of this thesis with regard to mobility features. The work done by Ashbrook et al. [AS02a] is not directly related but is one of the pioneering contributions within the field.

Saeb et al. provides a list of mobility features and relate them to mental state and user behaviour, the featueres are outlined in Table 2.2. Out of these features the authors found that Home Stay, Location Variance, Normalized Entropy and Circadian movement correlate strongly with the PHQ9 score (see Appendix B). Clinical assessment of depressive symptoms is usually done using questionnaires, such as the patient health questionnaire (PHQ-9). These questionnaires evaluate symptoms related to the patients state of mind, which for depressed patients tend to change. Saeb et al. finds a strong correlation between certain mobility features and the PHQ9 score obtained by the patients in a field study. Their findings therefore indicate that it is possible to monitor depression passively using GPS data tracked via the smart-phone. They explain that most people are unwilling to answer questions repeatedly over long periods of time, which is the case with an evaluation system based on the PHQ9 questionnaire. Passive monitoring via the patients smartphone could great improve the management of depression in populations by discovering relapses of high-risk patients earlier, so that they may receive the treatment they need before it is too late.

Canzian et al. provides a contribution similar to Saeb et al. and additionally discusses how existing digital systems for diagnosing depression require the user to interact with the device. These interactions can be inputs such as mood-state provided a few times a day, which is highly subjective and thus error-prone. Canzian et al. addresses this issue via objective patient data, namely the mobility patterns of users. These patterns were derived from location data gathering through smartphone apps, which as Canzian et al. puts it, is unobtrusive monitoring. These patterns are described by the mobility metrics (i.e. features) listed in Table 2.3. The authors found a significant correlation between these metrics and depressive moods similar to [Sae+15b] for which the *PHQ-8* questionnaire is used as reference. This is the same questionnaire as the PHQ9, although only containing the first 8 questions. While Canzian et al. and Saeb et al. describe many of the same features, Canzian provides much more detailed definitions of the features. A good example of this is the *Routine Index* feature, which loosely corresponds to the *Circadian Rhythm* feature in [Sae+15b]. Canzian et al. [CM15] uses the notion of trajectories including *places* and *moves* defined by Spaccapietra et al. [Spa+08] to define the *mobility trace* feature. This feature is used to derive the rest of the features, which is also the approach used for this thesis, although the terminology differs slightly.

Cuttone et al. [CLL14] investigates how accurate significant locations can be inferred from sparsely sampled location data using the GPS tracker in smartphones. For longitudinal studies it can be a problem for the phone battery if the sampling rate is too high. Therefore a more low-energy approach is used where the sampling rate- and the location accuracy are low. A small-scale study was conducted in which location data collected continuously and the users filled out an online diary on a daily basis. The aim was to compute features *Stops* and *Places of Interest (POI)* for each user and match them with the provided answers. These features are described in Table 2.1.

The work by Ashbrook et al. [AS02a] from 2002 is one of the earliest contributions to the field of GPS data based user context. Here, user context refers to where the user will move next, given their current location. While this is not related to mental health necessarily, it is interesting to look into the approach taken given that it is a pioneering contribution. The features defined by Ashbrook are location features only, i.e. features consisting of a latitude and longitude and are shown in Table 2.4.

2.2.1 Data Collection

Canzian et al. built a custom Android app (MoodTraces) which collected data from 46 users over almost 10 months. The app had participants fill out a survey daily between 16:00 and 02:00 the subsequent day, where a phone notification prompted the participant to fill out the survey. Each survey contained a 'trap' question which asked whether the user was current at home or work. The answer to this question could be verified in the data analysis phase, and was used to filter out invalid surveys. The survey asked the user to name all the depressive symptoms they had on that day. PHQ8 score is normally done by asking every 14 days, and have the participant answer how many times they had a symptom over the last 14 days. By using the questionnaires, the PHQ8 scores could be computed by summing the amount of days with the symptoms, which in total add up to a score. It would happen that users forgot to fill out their questionnaire or gave an invalid survey response, in which case an interpolation method was used to cover gaps in the data.

Saeb et al. used a custom Android application, Purple Robot, which collected the

Feature	**Description**
Place of Interest (POI)	*Location of relevance frequented by the user, e.g. home, gym or workplace*
Stop	*A visit to a POI with an arrival- and departure timestamp*

Table 2.1: The Mobility Features defined by Cuttone et al. [CLL14].

Feature	Description
Number of Clusters	*Number of clusters found by the clustering algorithm*
Home Stay	*Percentage of time stayed at the home cluster*
Location Variance	*Variance of the location of stationary points*
Location Entropy	*Entropy of the time distribution wrt. clusters*
Normalized Location Entropy	*Entropy normalized by the maximum possible entropy. Invariant to the number of clusters*
Transition Time	*Percentage of time not stationary*
Total Distance	*Total distance travelled*
Circadian Movement	*The degree to which changes in the location follow a 24 hour rhythm*

Table 2.2: The Mobility Features defined by Saeb et al. [Sae+15b].

users' GPS data with a sampling rate set to once every 5 minutes. Participants had the application manually installed on their existing phones or were given an Android phone with the app pre-installed on it which they were to use as their primary phone. Participants filled out a PHQ9 questionnaire just prior to starting the study, and once again once the study concluded after 14 days.

Ashbrook et al. wrote their contribution in 2002 where old-fashioned GPS receivers were the standard tool for collection location data. Sampling was done once per second which provides very high resolution, however GPS signals had less than ideal coverage in 2002 which meant that the receiver did not work indoors most of the time - which actually worked to their advantage since it made identifying places easier. In addition the GPS receiver also did not capture data if the user was moving at a speed of less than 1 kilometer per hour, to conserve battery. Six users participated in a 2002 study which lasted for seven months.

Cuttone et al. conducted a study with 7 participants including the author, with the author being tracked for four months and the other participants being tracked for two months. Data collection was done with a custom smartphone app developed using the Funf Open Sensing framework [1], where participants were provided an Android phone to use as their main phone. Sampling was carried out with a very low sampling rate of once were 15 minutes which had the upside of not draining the phone battery much. Participants were also instructed to fill out a daily diary of the place

[1] `https://www.funf.org/`

Feature	Description
Place	*A location the user frequently visits*
Significant Place	*The top 10 most visited places out of all places*
Move	*A trajectory between two places*
Mobility Trace	*The stops and moves a given time interval*
Total Distance	*Distance travelled for a given time interval*
Max Distance	*Maximum distance between any two points for a given time interval*
Radius of Gyration	*Radius of the area covered for a given time interval*
Standard deviation of Displacement	*Standard deviation for stop locations for a given time interval*
Max Distance from Home	*Maximum distance from home to any place visited in a given time interval*
Number of Places	*The number of unique places visisted for a given time interval*
Number of Significant Places	*The number of unique significant places visited for a given time interval*
Routine Index	*The average difference between the mobility behaviour of the user for a given interval, on a given day, compared to the same time interval on other days*

Table 2.3: The Mobility Features defined by Canzian et al. [CM15].

Feature	Description
Place	*A geo-location at which the GPS signal could not reach such as a building*
Location	*The centroid of a cluster of places*
Sublocation	*A location within a location, i.e. university building on a uni-campus*

Table 2.4: The Mobility Features defined by Ashbrook et al. [AS02a].

they had been at, and their movements. This was done through an online spreadsheet and participants were reminded via email. The authors comment that using diaries as ground truth is difficult due to participation compliance decreasing over time, since filling out a diary is tedious. In addition the concepts of stops and POIs are somewhat ambiguous and subjective which means that even if participants do fill out their diary, it is subjective and error-prone. This was evident in the study where participants would sometimes fill out sequences of POIs in the wrong temporal

order, and entries were prone to have typos in them which later had to be manually corrected by the researcher. At the end of the study the participants were asked to create a subjective list of their POIs with coordinates obtained through Google Maps.

2.2.2 Computing Features

Canzian et al. uses an algorithm for sampling location data in a way which saves phone battery but also helps identify places. For this, the accelerometer is used by an activity recognition algorithm to determine the user's activity state. This is done by modelling the user as being a state-machine with 3 states:

- Static (S)
- Moving (M)
- Undecided (U)

A user is said to arrive at a place if the transition $M \rightarrow U \rightarrow S$ happens and a departure is noted when $S \rightarrow U \rightarrow M$ happens. By using all the location samples in between the arrival and departure, the centroid of the place can be found using a clustering algorithm. In addition, an iterative clustering algorithm is used to merge places with close proximity to each other, specifically a distance of 200 meters is used as the merging criterion.

Saeb et al. uses a two-stage pipeline for location data processing:

- Each data point is labeled as either being in a stationary or transitional state by using the speed of each data point. Points are labeled using a threshold of 1 km/h; a speed lower than the threshold indicates the state is stationary and higher is a transitional point.
- The K-means clustering algorithm is used to group the *stationary points* into frequently visited places. Since the number of places, i.e. the number of clusters for the K-means algorithm (the parameter K) is not known beforehand, the best parameter value is found using cross-validation.

After this pre-processing procedure was applied, the features described in Table 2.2 were derived.

Ashbrook et al. used the fact than the GPS receiver could not get a signal inside most buildings which was used to identify a user's *places*. The GPS receiver used had a 15 meter accuracy meaning the location coordinates reported may be prone to noise. To remove noise the *K-means* clustering algorithm was applied to the data which resulted in the *locations* features. Inside *locations*, a set of *sub-locations* can be found, which are nodes in a network of small-scale locations. A prime example of

a location with sub-locations is a university campus, where the sub-locations are the different department buildings. These features are listed in Table 2.4.

Cuttone et al. used two main approaches for finding stops and POIs. The first in voles finding stops first and then grouping stops into POIs. The second approach works the other way around and identifies POIs first by clustering the location data, and infers the stops from these POIs afterward. For finding the stops, 2 different algorithms, Distance Grouping and Speed Thresholding, were tried each yielding similar results.

- Distance grouping involves iterating location points temporally and grouping them based on a max-distance measure. Each stop is created with a single location point loc_i and each subsequent location point loc_{i+k} is then added to the stop, with k being incremented by 1 each iteration. This is done until $distance(loc_i, loc_{i+k} > d_{max}$ where d_{max} is the maximum allowed distance. By increasing the max-distance fewer stops of longer duration will be found. For finding the optimal value of this parameter, the f_1 score was calculated for the specific dataset.

- Speed thresholding determines which location points belonged to stops, and which points did not. The max-speed parameter was determined by grouping data samples based on their timestamp, each group having a duration of T. Choosing the max-speed parameter was then done by considering the speed between the median location of each bin. Gaussian Mixture Modelling was also tried which looks at all the location samples independent of their timestamp, and identifies POIs directly from clusters with large densities.

For both these algorithms, the DBSCAN clustering algorithm by Ester et al. [Est+96] was applied afterwards, which groups stops into places and marks noisy stops as not belonging to a place.

The second approach used a *Gaussian Mixture Model* to cluster the raw location data into places, and infer the stops from these clusters. The upside to this method is that it is much simpler to perform on paper, but the downside is that it does not consider the temporal dimension at all which can contain a lot of useful information. This means the GMM uses more compute power than the other two approaches.

For this thesis, a modified version of the distance grouping algorithm was chosen is described in Chapter 3.

2.3 Mobile Sensing Frameworks

The *AWARE Framework* [FKD15] is an open-source toolkit and a reusable platform for collecting data and generating user-context on mobile devices. Phones possess

high-quality sensors but are resource-constrained with regards to their processing speed and battery capacity, which must be considered when computing contexts in real-time. The *AWARE Framework* aims to ensure an easy way of collection contexts for the application developer. More specifically it aims to reduce the software development effort of researchers when building mobile tools for developing context-aware apps. By designing an API that conceals the underlying implementation of sensor data-retrieval, and exposes an abstract representation of a *user context object*, AWARE shifts the focus from software development to data collection and analysis. Currently AWARE is available on both iOS and Android natively, meaning two code bases will have to be maintained if one wishes to write both an Android and iOS application. AWARE supports a number of data channels [2] such as the built-in sensors, as well as more HCI-based data sources such as *Application Usage*, *SMS*, and *Phone Call Logs*. Some of the channels are however not available on iOS due to the iOS developer API being more restrictive than that of its Android counterpart for most data channels.

Inspired by AWARE, the CARP Mobile Sensing (CAMS) Framework by Bardram [Bar20] is a mobile sensing framework for adding digital phenotyping capabilities to a mobile-health app. A number of technological platforms for mobile sensing have been presented over the years and a lot of knowledge on how to facilitate mobile sensing has been accumulated. CAMS is a modern cross-platform software architecture providing a reactive and unified programming model that emphasizes extensibility, maintainability, and adaptability. CAMS is written in Flutter and in contrast to AWARE uses a single code base to compile to both Android and iOS.

CAMS is designed to collect research-quality sensor data from the many smartphone data channels such as sensors and location data, in addition to external sensors that the phone is connected to, such as wearable devices. The main focus of the framework is to allow application programmers to design and implement a custom mobile health app without having to start from scratch, with regards to the sensor integration. This is done by enabling the programmers to easily add mobile sensing data channels to their application. This would include adding support for collecting health-related data channels such as *ECG*, *GPS*, *Sleep*, *Activity*, *Step Count* and many more. Additionally, to format the resulting data according to standardized health data formats (like *Open mHealth* schemas [3]). Last but not least, the collected data should be uploaded to a server, using an API (such as *REST*), and should come in a standardized format such that it may easily be imported for data analysis. To include as many data channels as possible the application should also be able to support different wearable devices for ECG monitoring and activity tracking. Hence, the focus is on software engineering support by providing a high level programming API and a run-time execution environment. In order to simplify the process for the researcher further, the

[2] `https://awareframework.com/sensors/`

[3] `https://www.openmhealth.org/documentation/#/schema-docs/schema-library`

researcher only has to maintain a single code base - in contrast to AWARE. This is because CAMS is written in the cross-platform framework flutter where one common code-base in the Dart programming language is used. Maintenance of the framework will be ongoing, and is required for it to stay relevant as the underlying mobile phone operating systems and APIs are evolving.

An integration for the *Mobility Features Package* into the CAMS Framework is planned but is a future goal beyond this thesis (see Future Work in 7).

CHAPTER 3

Theory

This chapter will describe the theoretical background for the Mobility Features Package including all the features chosen to be included and how they are computed.

3.1 Location Data

The Navbar Global Positioning System (GPS) is a space-based radio-navigation system developed by The US Office of the Department of Defense [Dep20] that provides location data to GPS receivers such as the one found in smart-phones. GPS is capable of delivering information such as latitude and longitude coordinates which indicates where on the Earth's surface the receiver is located, in addition to the altitude, i.e. the distance from the surface. Previously one would have to use a stand-alone GPS tracker as it was done by Ashbrook et al. [AS02b] in order to collect GPS data. Nowadays smart-phones contain GPS receivers which enable users to use a variety of navigation services and applications with their phone alone. For calculating distances between points, a standard Euclidean distance metric cannot be used since the earth is not a plane. Multiple methods of calculating the distance between GPS coordinates exist, one of the fastest to compute being the Haversine formula [Bru13]. The Haversine distance computes the great circle distance btetween two points on a sphere. The Earth is however not exactly spherical, and therefore the Haversine distance is an approximation that works for shorter distances. Given the radius of a sphere r and two points on the sphere, A and B, the *haversine distance* between the two points can be directly computed as

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{lat_B - lat_A}{2}\right) + \cos(lat_A) \cdot \cos(lat_B) \cdot \sin^2\left(\frac{lon_B - lon_A}{2}\right)}\right) \tag{3.1}$$

3.2 Feature Definitions

In this section, an overview of the algorithms used by the Mobility Feature Package will be provided including formal definitions and the considerations behind. The actual implementation will be discussed in chapter 5.

The Mobility Features included in the Mobility Features package are a subset of the features described by Saeb et al. [Sae+15b], Canzian et al. [CM15] and Cuttone et al. [CLL14] and are outlined in Table 3.1.

Feature	**Description**
Stop	*A collection of location samples representing visit at a known place with an arrival and departure timestamp*
Place	*A cluster of stops found the DBSCAN algorithm*
Move	*A travel between two stops with a path of location samples*
Number of Places	*The number of places visited today*
Home Stay	*The percentage of the time spent at the home place today*
Routine Index	*The overlap of the time-place distribution that between today and the max 28 previous days*
Location Variance	*The statistical variance in the latitude and longitude coordinates today*
Distance Travelled	*The total distance travelled today in meters*
Entropy	*The entropy of the time-place distribution today*
Normalized Entropy	*The entropy normalized the by the max possible entropy. Invariant to the number of places visited today*

Table 3.1: The features included in the Mobility Features Package.

3.2.1 Dates and Periods

For collecting data over time, we define today's date as d_t an the yesterday as d_{t-1} in order to define a period as the set of today's date and it's preceding 28 dates defined as $D = \{d_{t-28}, d_{t-27}, ..., d_t\}$. This definition is necessary for computing the routine index feature, since the feature is computed by comparing today with the last 28 days. The reason for choosing 28 days, i.e. 4 weeks is that we wish to capture people's routine at the moment.

Peoples' habits will inevitably change somewhat over time, and if one compares the routine of a certain person now to what their routine looked like a year ago, it is likely somewhat different. But just because a routine changes over time, does not mean it does not exist presently.

3.2.2 Location Sample

A location sample is a timestamped location and is defined by the tuple $x = (T, l)$ where T is the exact timestamp and l is the Location defined as a geographical point on the globe. The distance between two location samples x_a and x_b is defined as $\delta(x_a, x_b) = \delta(l_a, l_b)$ where δ is the *Haversine* distance function.

3.2.3 Stop

Finding stops is done by applying a modified version of the distance grouping procedure described by Cuttone et al. [CLL14]. The modified algorithm traverses the location samples in temporal order, and adds the next location sample if it lies within the max-radius of the current stop. The modification is that when comparing each new location sample, the median location of the current stop is used rather than the distance between the current- and next location sample. The max-distance parameter therefore represents a stop-radius which is set to $r_{stop} = 25$meters. A stop is defined in Equation (3.2).

$$S = \{s_1, s_2, ..., s_{|S|}\} \mid s_i = (T_{arr}, T_{dep}, l) \tag{3.2}$$

The triple (T_{arr}, T_{dep}, l) denotes the arrival timestamp, the departure timestamp and the cluster location for the stop s_i, respectively. Stops are found using Algorithm 3.2.3.

Algorithm 1 Find Stops

Input: set of time ordered data points, X
Output: set of stops, S

$S \leftarrow \{\}$ **while** *at least one point* x' *remains in* X **do**
  $X \leftarrow X - x'$;
  $s \leftarrow \{x'\}$;
  for $x \in X$ **do**
    if $\delta(x,s) \leq r_{stop}$ **then**
      $s \leftarrow s \cup \{x\}$;
      $X \leftarrow X - x$;
    else
      $S \leftarrow S \cup \{s\}$;
      break **for**-loop;
    end
  end
end
return S

Given a stop s, the arrival is denoted $s.arrival$ and departure $s.departure$.

3.2.4 Place

Similar to Cuttone et al. *places* are found by clustering stops using the DBSCAN algorithm by Esther et al. [Est+96]. All the stops today in the period D should used to for finding places, for reasons explained in subsection 3.2.6. That is, given a set of stops S for the period D, the set of places P for the period D is defined by Equation (3.3).

$$P = p_1, p_2, ..., p_N \mid p_i = \{s_1, s_2, ..., s_{|p_i|}\} \tag{3.3}$$

The procedure for finding places defined in Algorithm 3.2.4.

Algorithm 2 Find Places

Input: set of stops, S
Output: set of places, P
$L = DBSCAN(S)$ where s_i has label l_i;
group each stop $s_i \in S$ by l_i;
$S' = \{s_i \mid l_i \geq 0\},\ |S'| = N$;
$p_i = S'_i$ where each stop $s \in S'_i$ has the label l_i;
return $P = \{p_i : i = 0, ..., N\}$;

3.2.5 Move

A move is calculated using the stops- and the location sample by going through each stop and calculating the distance between the current stop and the following stop. The distance is computed by going through all the location samples which were sampled in the time interval between the two stops. These points form the path which was taken between the two stop and the path is used to calculate the exact distance traveled.

A set of moves is defined as

$$M = \{m_1, m_2, ..., m_{|M|}\} \mid m_i = (s_a, s_b, X_i), X_i = \{x_1, x_2, ..., x_{|X_i|}\} \tag{3.4}$$

is a set of time-ordered location samples. Moves are found using the following procedure outlined in Algorithm 3.2.5.

Algorithm 3 Find Moves

Input: set of stops, S, set of time ordered data points, X
Output: set of moves, P
$M = \{\}$ **for** $s_i \in S$ **do**
    $X_i = x$ for which $s_i.departure \leq x.timestamp \leq s_{i+1}.arrival$;
    $d_i = \sum_{x_j \in X_i} \delta(x_j, x_{j+1})$;
    $m_i = (s_i, s_{i+1}, d_i)$;
    $M = M \cup \{m_i\}$;
end
return M;

3.2.6 Hour Matrix

The hour matrix is an auxiliary feature used to compute the home stay and routine index feature. A matrix with 24 rows, each row representing an hour in a day, and columns equal to the number of places. The hour matrix represents the time-place distribution for the user during a day.
This matrix is constructed from all the stops on a given day, each of which belong to certain *place* and has an *arrival* and *departure* timestamp. From this, it can be calculated exactly which hour slot(s) to fill out and the duration to fill that slot with. For simplicity, we define a couple of constraints on the hour matrix:

- The hour matrix has exactly 24 rows, each representing 1 hour in a day.
- The number of columns represents the number of places for the period.
- An entry represents the portion of the given hour-slot that was spent at a given place.
- Each row can maximally sum to 1.

	Place #1	**Place #2**	**...**	**Place #N**
00 - 01				
01 - 02				
...				
16 - 17				
17 - 18				
18 - 19				
...				
23 - 00				

Figure 3.1: An *Hour Matrix*.

Formally, given a period (D) for which the number of places is given as N the hour matrix H for today (d_t) is defined as:

$$\mathsf{H}(d_t) \in [0,1]^{24\times N}, \sum_{j=1}^{N} \mathsf{H}_{i,j}^{d_t} \leq 1 \tag{3.5}$$

Given a stop s, let $i = hour(s.arrival)$, $j = hour(s.departure)$ and $\Delta_T(\cdot)$ is the function for calculating the duration in hours.

$$\mathsf{H}_{i,p} \leftarrow \mathsf{H}_{i,p} + T \tag{3.6}$$

If $i = j$ then $T = \Delta_T(s.departure - s.arrival)$, otherwise the following algorithm is applied:

$$\mathsf{H}_{i,p} = 1 - T \tag{3.7}$$

Where the value of T depends on the variable $k = i$ up to j:

$$T(k) = \begin{cases} \Delta_T(T_{dep} - T_{arr}) & \text{if } i = k = j \\ 1 - (hour(T_{arr}) - T_{arr}) & \text{if } i = k < j \\ 1 & \text{if } i < k < j \\ hour(T_{arr}) - T_{arr} & \text{if } i < k = j \end{cases} \tag{3.8}$$

3.2.7 Home Stay

The home stay feature indicates the percentage of time spent at the Home cluster, out of all the time of the day. In the literature, the home cluster is defined as the cluster which the user spends the most time at between 00:00 and 06:00. In the work by Saeb et al. and Canzian et al. [Sae+15b; Sae+15a; CM15] the home cluster was evaluated over the duration of the whole study, however we define it on a daily basis since the home cluster may change from day to day. Formally, the *home place* $p_h(d_t)$ for the date d_t where the index h is found using Equation (3.10).

$$h = \underset{n}{\operatorname{argmax}} \sum_{m=1}^{6} \sum_{n=1}^{N} \mathsf{H}(d_t)_{m,n} \tag{3.9}$$

In the work by Saeb et al. [Sae+15b] and Canzian et al.[CM15] it is not stated how this would be calculated for an incomplete day. It was chosen to use the time elapsed from midnight until the departure of the last known stop. The home stay feature is computed using (3.10) where the duration of a stop s will be denoted $\Delta T(s)$ and similar the duration spent at a place p is defined as $\Delta T(p)$.

$$\Delta T(p_h(d_t)) = \frac{\sum_i \Delta_T(s_i) \mid s_i \in p_h(d_t)}{T_{now} - T_0} \tag{3.10}$$

Where T_{now} is the departure time of the last known stop, and thus $T_{now} - T_0$ is the time elapsed since 00:00:00 today.

3.2.8 Total Distance Travelled

First we define the distance of a move m_i as the sum of all the distances in the 'chain' of points in X_i, i.e.

$$\delta(m_i) = \sum_{j=1}^{|X_i|-1} \delta(x_j, x_{j+1}) \tag{3.11}$$

The *Total Distance Travelled* for a date d_t is defined as

$$\delta(d_t) = \sum_{i=1}^{|M|} \delta(m_i) \tag{3.12}$$

where M refers to the moves on date d_t.

3.2.9 Number of Places

By using the DBSCAN algorithm to cluster a set of stops, each stop is given a cluster label, either being non-negative if it belongs to a cluster, or the label -1 if it is considered noise. The *Number of Places* is therefore defined as the size of the set of non-negative cluster labels $K = \{k_1, k_2, ..., k_N\}$, i.e.

$$N = |K| \tag{3.13}$$

3.2.10 Location Variance

Defined by Saeb et al. [Sae+15b], the Location Variance is defined as the natural logarithm to the variance of the latitude and longitude coordinates summed together:

$$\sigma^2 = \log(\sigma_{lat}^2 + \sigma_{lon}^2) \tag{3.14}$$

The logarithm is applied in order to compensate for the skewness in the location distribution of location variances across users.

3.2.11 Entropy and Normalized Entropy

Within the field of Information Theory, entropy is described in MacKay [Mac03] as a quantity associated with a random variable, and can be interpreted as the average level of *information* contained within the outcomes of that variable.
The Entropy $E(X)$ of the set of outcomes $X = \{x_1, x_2, ..., x_n\}$ is defined as

$$E(X) = -\sum_{i=1}^{n} p(x) \log p(x) \tag{3.15}$$

The Entropy is maximised if $p(x) = \frac{1}{n}$ for all $x \in X$, i.e. all outcomes are equally likely. If this is the case, the Maximum Entropy becomes

$$E_{max}(X) = -\sum_{i=1}^{n} p(x) \log p(x) = -\sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} = -n\frac{1}{n} \cdot -\log n = \log n \tag{3.16}$$

We define the Normalized Entropy as the Entropy normalized using the maximum possible entropy E_{max}:

$$E_N(X) = \frac{E(X)}{E_{max}(X)} \in [0, 1] \cdot -\log n = \log n \tag{3.17}$$

The Normalized Entropy (NE) makes it easier to compare Entropy values of different distributions since they all reside on the same scale, being a scalar value between 0 and 1. A NE value near 1 indicates that the X follows a uniform distribution where every outcome is equally likely. A small NE value indicates that the distribution is very skewed, with certain outcomes having very high likelihood and some having much lower likelihood. In the context of user mobility, we can view the time user spends at a certain place as the outcome of the place variable, i.e.

$$E(P) = -\sum_{i=1}^{n} Pr(p_i) \log Pr(p_i) \cdot -\log n = \log n \tag{3.18}$$

and by using (3.19) and (3.18) we get the Normalized Entropy for the time-place distribution:

$$E_N(P) = \frac{E(P)}{E_{max}(P)} \in [0, 1] \tag{3.19}$$

where P is the set of places visited today and $Pr(p_i)$ refers to the time spent at place p_i today. It must holds that $Pr(p) > 0$ for $p \in P$, otherwise the term $\log P(p_i)$ cannot be evaluated since $\log(0)$ is undefined. The concept of NE gives us a tool to say something about where the user spends their time; a high NE value indicates they spend their time uniformly among the places, whereas a low value indicates that the user spends most of their time at very few places.

3.2.12 The Routine Index

The routine index describes how similar the place-time distribution of a given day is, compared to previous days for a period D. The place-time distribution for a day is defined by how much time was spent for each of the 24 hours, at different places. For

computing the routine index, a period length of 28 days was chosen, which means the routine index of today describes how similar today was to each day during the last month. The implementation by Canzian et al. [CM15] involved using a different modelling approach and was therefore very complicate to compute. Therefore, a more straightforward definition of the routine index was chosen. To the best of our knowledge this definition defined in Equation (3.23) in conjunction with the concept of an hour matrix is novel, and models the feature as a similarly measure between two Hour Matrices. More exact, the routine index is a similarity measure between the hour matrix today and the 'average' hour matrix for the last 28 days. The result of the similarity measure is a scalar between 0 and 1 - a low value indicating a small degree of overlap and a high value indicating a high degree of overlap.
Given an array of *Hour Matrices* for a period $D' = D - d_t$ (i.e. the historical dates preceding today d_t) the *Routine Matrix* is defined as the average entry for each hour matrix given in Equation (3.20). The days contained in $|D'|$ are indexed with the integer k:

$$\mathsf{R}(D) = \sum_{k=0}^{|D'|} \frac{1}{|D'|} \mathsf{H}(d_k)_{i,j} \tag{3.20}$$

The feature can therefore indirectly be computed from the stops of the last 28 days which means only the stops are necessary to store. In a field study (described in chapter 6), the author was found to have had just around 20 stops per day which amounts to under 600 stops for a 4 week period. Computing the feature requires the places to be found, i.e. to run DBSCAN on the stops, and can result in the feature being expensive to compute. However, 600 data points is a manageable number to work with.
The overlap function produces a matrix O given two matrices A and B of equal dimensions:

$$\mathsf{O}(\mathsf{A}, \mathsf{B}) = \sum_{i=1}^{24} \sum_{j=1}^{N} \min(A_{ij}, B_{ij}) \mid A_{ij} \geq 0, B_{ij} \geq 0 \tag{3.21}$$

The maximum potential overlap is limited by the minimum of the two matrix sums, this is reflected in (3.22).

$$\mathsf{O}_{max}(\mathsf{A}, \mathsf{B}) = \min(\sum \mathsf{A}, \sum \mathsf{B}) \tag{3.22}$$

The routine index for today d_t given the historical dates D, is defined as the overlap normalized by the maximum potential overlap of today's and the Routine Matrix (Equation (3.23)).

$$r(d_t, D) = \frac{\sum(\mathsf{O}(\mathsf{H}(d_t), \mathsf{R}(D)))}{\min\left(\sum \mathsf{H}(d_t), \sum \mathsf{R}(D)\right)} \tag{3.23}$$

Equation (3.23) uses the smallest sum of the two matrices to normalize the overlap, which is done to avoid punishing days with gaps in the data too much. Otherwise one could simple define the maximum potential overlap as 24 hours. However travels between places are not reflected in an Hour Matrix, which will lead to gaps in the data since it takes time to move between places. Ideally, both stops and moves should be used to construct the Hour Matrix, it is not at present clear how this would be done.

Routine Index Examples

In Equation (3.24) we see the Hour Matrix (H) which tells the story of a user doing the following:

- Timeslot 1: *The user visits Place #1 for exactly one hour*
- Timeslot 2: *The user visits Place #2 for exactly one hour*
- Timeslot 3: *The user visits Place #1 again for exactly one hour*

When compared to Routine Matrix (R) it only overlaps with one hour, where the maximum potential overlap is 3 hours, since both matrices sum to 3 hours. The routine index for these two matrices is thus

$$\mathsf{H} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathsf{R} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathsf{O}(\mathsf{H}, \mathsf{R}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.24}$$

For (3.24) the routine index becomes:

$$RI = \frac{1}{1+1+1} = \frac{1}{3} \tag{3.25}$$

In Equation (3.26) we see the Hour Matrix (H) which indicates the user does the following:

- Timeslot 1: *The user visits Place #1 and #3 for exactly half an hour each*
- Timeslot 2: *The user visits Place #2 for exactly one hour*
- Timeslot 3: *The user visits Place #1 and #3 for exactly half an hour each once again*

When compared to Routine Matrix (R) it overlaps with 2 hours in total. The maximum potential overlap is 2 hours, since the minimum matrix sum is $\sum \mathsf{R} = 2$ the routine index becomes 1 (See 3.27).

$$\mathsf{H} = \begin{bmatrix} 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0 & 0.5 \end{bmatrix}, \quad \mathsf{R} = \begin{bmatrix} 0 & 0 & 0.5 \\ 0 & 1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}, \quad \mathsf{O}(\mathsf{H}, \mathsf{R}) = \begin{bmatrix} 0 & 0 & 0.5 \\ 0 & 1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \tag{3.26}$$

$$RI = \frac{1}{1 + 0.5 + 0.5} = \frac{1}{2} = 1 \tag{3.27}$$

CHAPTER 4
Software Design

This chapter deals with the design of the software architecture on a high level and will describe the package in terms of its components. Components may sometimes be referred to as *classes* and vice versa, where *component* refers to the general term and *class* refers to the programming language equivalent of a component.

4.1 System Design

The core idea of the *Mobility Feature Package* is to provide a very simple programming interface such that computing features can be done with very few lines of code. This requires hiding the implementation of the package such that the programmer has very limited ways in which he/she can interface with it. The design of the package API went through two main iterations which consisted of many smaller iterations. Mainly, the difference between the final iteration and the earlier iterations is the amount of code required for managing historical data that the programmer has to write. Early iterations put the responsibility on the application programmer to manage historical data. After developing the field study app discussed in chapter 6 it was decided to move most of this logic inside the package; it became glaringly apparent that the package was too cumbersome to use. The final iteration is the one discussed in this thesis including the choices and lessons learned on the way of designing and developing it. The flowchart in Figure 4.1 displays the task which the software system must be able to carry out, as well as which parts are done by the package, and which are done by the programmer. Namely, the task of collecting location data was chosen to be carried out by the programmer which was a major design choice. Two reasons led to this decision, the first of which being that the Mobility Features Package becomes more loosely coupled and modular. The second reason relates to maintenance; if the package was to implement location data collection then it becomes harder to maintain since any change to the location plugin would imply changes to the Mobility Features Package as well.

4.1.1 Component Overview

Figure 4.2 displays how the final iteration looks like as a software system: The main component is the *Context Generator* component which is the interface that the programmer will use. This component exposes two interfaces to the programmer allowing the user to store their collected *LocationSamples* as well as generate a *Mobility Con-*

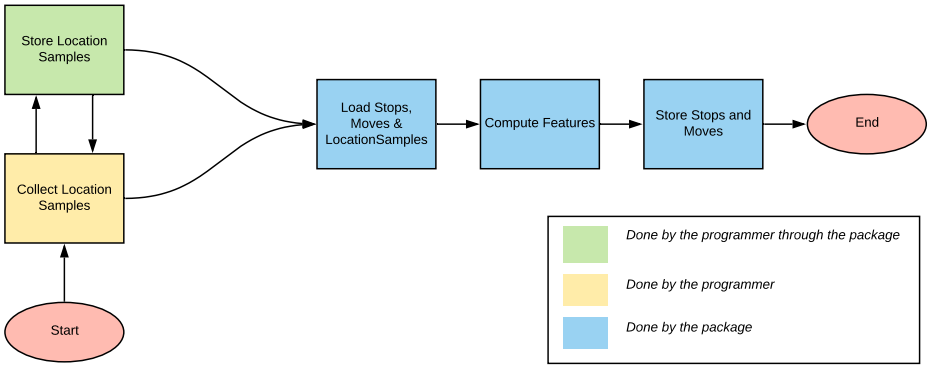


Figure 4.1: Flowchart of the feature computation process.

text that contains the daily features. The two exposed interfaces are provided by the programmer through mobile application. The *Context Generator* is also responsible for storing and loading data via the *Mobility Serializer* component, here the *Serializable* type refers to any data type that needs to be stored as historical data.

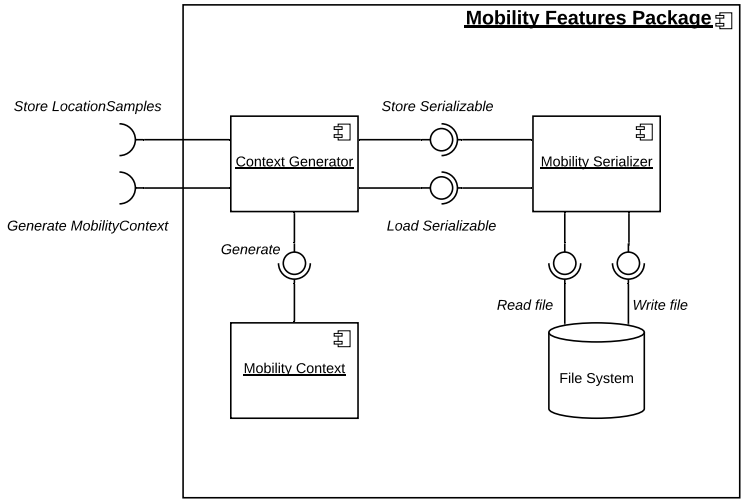


Figure 4.2: Component diagram for the *Mobility Feature Package* from an internal point of view.

Figure 4.3 shows the external software system that includes the mobile application using the package. The design choice of letting the programmer collect location data themselves is also reflected with the usage of an external location plugin. This location plugin will deliver location *Data Transfer Objects* (DTOs) [Fow+02] [p. 401] to the mobile health application. These objects hold location data, i.e. latitude, longitude, and a timestamp and can be converted to *LocationSamples* and saved through the *Mobility Feature Package.*

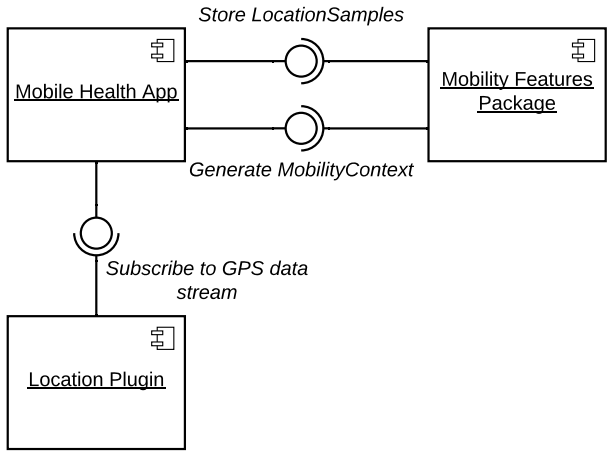


Figure 4.3: Component diagram for the *Mobility Feature Package* from an internal point of view.

4.1.2 Sequence Overview

To display the interactions between the components, sequence diagrams are used. Figure 4.5 shows the system from an external point of view, where the application subscribes to location updates via the location plugin, saves location data via the Mobility Features Package and generates a Mobility Context through it.
From an internal point of view, the *Context Generator* component calls the *Mobility Serializer* for storing and loading historical data, uses the loaded data to generate a *Mobility Context* by calling the MobilityContext component.

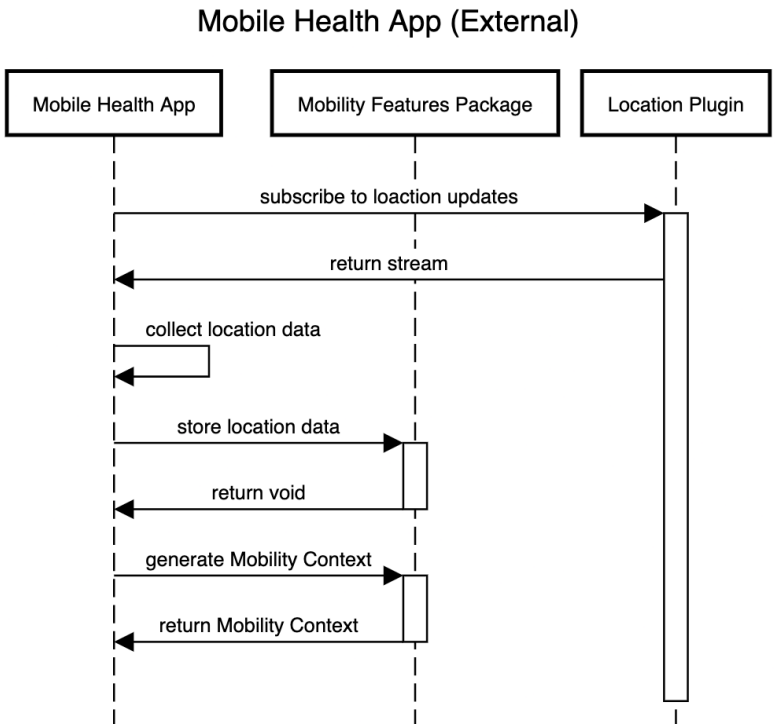


Figure 4.4: Component diagram for the *Mobility Feature Package* from an internal point of view.

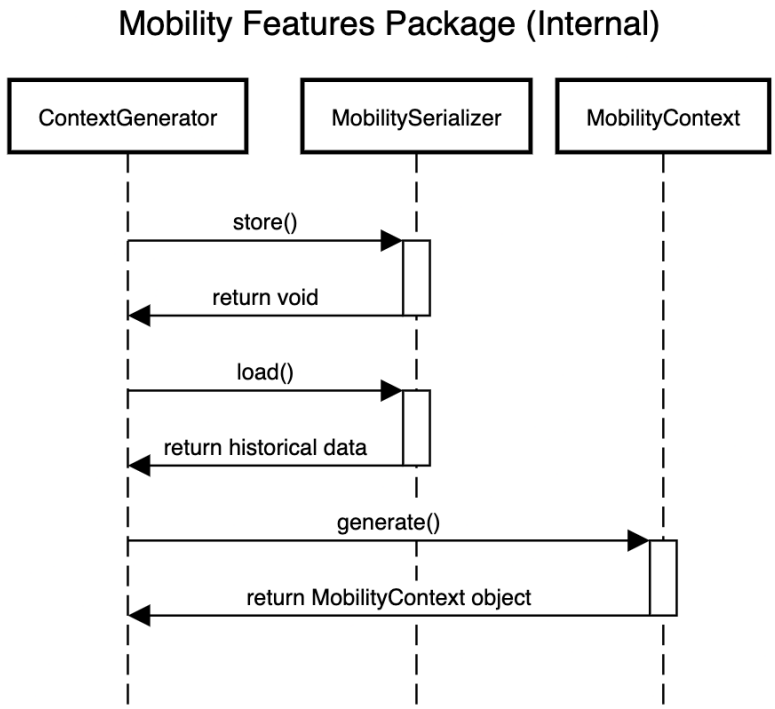


Figure 4.5: Sequence diagram for mobile health application using the Mobility Features Package, i.e. viewed externally from the package.

4.2 Domain Model

The domain model aims to model the theoretical descriptions outlined in Chapter 3 in an object-oriented manner.

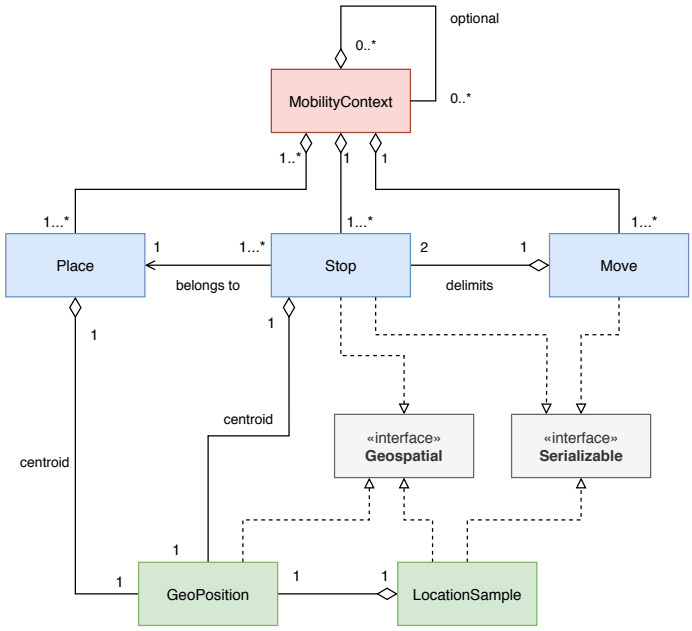


Figure 4.6: UML diagram for the classes used in the *Mobility Features Package*.

In order to capture the data model in an object-oriented programming language, a the UML diagram shown in Figure 4.6 was maintained as the implementation went along in order to keep track of relationships between the classes.
A **GeoPosition** represents a 2D position on the Earth's surface with a geographical *latitude* and *longitude*.

A **Location Sample** is a time-stamped *GeoPosition*. By having a time-stamp, a collection of Location Samples may be ordered and grouped by the time of day. In essence, the class is a Data Transfer Object (DTO) as defined by Fowler [Fow+02] [p. 401] which is used to transfer GPS data from an arbitrary Location plugin to the *Mobility Features Package*.

An **Hour Matrix** class is used to calculate the *Routine Index* feature, as well as to identify the *Home Cluster*, which is the place most visited during 00:00 and 06:00. An Hour Matrix is constructed from a list of *Stops* which all have the same date.

Stop is a centroid of a data point cluster (i.e. a GeoPosition) with an arrival- and departure timestamp, and a place ID indicating which place it belongs to.

A **Place** is defined by a GeoPosition computed from a list of *Stops* and found with the DBSCAN algorithm. Many Stops can belong to the same Place, but a Stop can only belong to one specific place.

A **Move** is constructed from a pair of *Stops* as well as the set of *Location Samples* that was sampled in the time interval between the two *Stops*. This set of Location Samples is the path the user took between the two *Stops*.

A **Mobility Context** is a the component which holds all the mobility features. This includes the location features (i.e. stops, places and moves) as well as the derived features. The component is created from daily stops and moves as well as a list of places for the whole period. In addition, a set of Mobility Contexts from previous days needs to be provided if the Routine Index feature is to be calculated, but is optional.

The **GeoSpatial (Interface)** forces any component implementing it to have a GeoPosition. This allows the Haversine distance to be calculated between components of different types.

The **Serializable (Interface)** forces any component implementing it to be serializable and deserializable. This is the case for GeoPositions, LocationSamples, Stops and Moves.

CHAPTER 5
Flutter Implementation

The final product allows the programmer to compute mobility features with the following 3 lines of code displayed in Figure 5.9. This chapter will describe how the package design described in Chapter 4 was implemented in Flutter such that this succinct code interface was achieved. The Flutter, and Python source code for this thesis can be found at `https://github.com/thomasnilsson/mobility-features-thesis`.

```dart
/// Collect data with a location plugin
List<LocationSample> locationSamples = ///

/// Store data via the package
await ContextGenerator.saveSamples(locationSamples);

/// Compute Features via the package
MobilityContext context = await ContextGenerator.generate();
```

Figure 5.1: All the three lines of code necessary for the application developer to write.

5.1 Flutter, Packages, and Plugins

Flutter is a cross-platform app development framework developed by Google and released in 2018. It allows an application programmer to write a mobile application using a single codebase written in the Dart programming language, and compile this source code to a native Android and iOS application. This has the clear advantage of reducing the amount of labor needed to produce mobile applications which most of the time need to be released on both platforms. Packages and Plugins are the Flutter equivalent of a software library that is hosted on the Dart package manager at `pub.dev`, the Mobility Features Package is hosted at `pub.dev/packages/mobility_features`. The author has written several packages for the Flutter framework released under the CACHET publisher[1]. The Mobility Features Package has a library file of the same name as the package *mobility_features.dart* which is the central point of import statements for all the source code. All import statements are made within this file, and each file belonging to the library are declared using the *part* keyword.

[1] `https://pub.dev/publishers/cachet.dk/packages`

```
library mobility_features;

import 'dart:math';
...

part 'mobility_functions.dart';
...
```

Each file included in the library will have the equivalent *part of* keyword at the top of their file, which allows the file to import all dependencies from the library, and makes the file public to other files within the library and vice versa.

```
part of mobility_features;
...
```

Packages also come with a set of unit tests. Unit testing is discussed in Appendix F where selected tests are shown.

5.2 Package Implementation

The package was implemented in Flutter according to the design in Chapter 4 in which a series of components and the overall data model was outlined. This section will go through selected examples of source code as well as the general principles applied, to achieve the specified design, when implementing in Flutter and Dart. In the Dart programming language, fields, constructors and methods are declared private by using the underscore prefix, i.e. the field `routineIndex` becomes `_routineIndex`. This is used throughout the implementation to prevent the application programmer to access the inner parts of the package. The parts of the implementation related to storage of historical data can be found in Appendix E. The offline algorithms for computing features were developed in Python with a large part being carried out by Jonas Busk[2]. Source code for these algorithms can be found in Appendix D.

5.2.1 Domain Model Implementation

All the components specified in the Domain Model Chapter 4 were implemented with their respective relations to each other. The only class with a public-facing constructor is *LocationSample*, and by transitivity, also *GeoPosition*. This is done to allow the user instantiate a *LocationSample* with data from a given *Location DTO*. The *GeoPosition* class a field for the latitude and one for the longitude and a fundamental class used by the *GeoSpatial* interface. The interface is a private abstract class which means it is only visible internally in the package library.

[2] `https://www.researchgate.net/scientific-contributions/2129480702_Jonas_Busk`

```
abstract class _Geospatial {
  GeoPosition get geoPosition;
}
```

This interface allows other classes to promise the Dart compiler that it has a *GeoPosition* field which allows it to be compared to other classes that implement the same interface. In Dart interfaces and abstract classes are the same thing, and the *abstract class* keyword is used for implementing them. The *GeoPosition* class even implements this interface since a *GeoPosition* object itself has a *GeoPosition*. This may seem superfluous but will come in handy when finding Stops (see Subsection 5.2.2).

```
class GeoPosition implements _Serializable, _Geospatial {
  double _latitude;
  double _longitude;

  GeoPosition(this._latitude, this._longitude);

  GeoPosition get geoPosition => this;
  double get latitude => _latitude;
  double get longitude => _longitude;
}
```

5.2.2 Computing Features

Finding the location features Stops, Moves, and Places were done according to the algorithms described in Chapter 3.

Stops

The Stop class has two constructors: A factory constructor which takes a set of LocationSamples from which the centroid of the set is computed, as well as the earliest timestamp, which will be the arrival time, and the latest timestamp which will be the departure time. After these attributes are found, the normal constructor is used.

```
factory Stop._fromLocationSamples(List<LocationSample> locationSamples,
  {int placeId = -1}) {

  GeoPosition center = _computeCentroid(locationSamples);
  return Stop._(center, locationSamples.first.datetime,
    locationSamples.last.datetime, placeId: placeId);
}
```

The normal constructor uses a *GeoPosition*, in addition to an arrival and departure time. A *place ID* may also be specified at construction, but often it is not yet known at construction time hence it is optional.

```
Stop._(this._geoPosition, this._arrival,
    this._departure, {this.placeId = -1});
```

The Stop algorithm takes a List of *LocationSamples* as input and uses two while-loops, and two pointers (*start* and *end*) to delimit a subset of the input data. Every time the outer loop iterates, the *start* pointer is moved past the end pointer, to skip already seen data. The inner loop is responsible for moving the *end* pointer: With each iteration of the inner loop, the centroid of the current subset is computed. If the distance from this centroid to the latest added sample is within the given `stopRadius` parameter, then the subset is expanded by incrementing the *end* pointer, and the process is continued. Otherwise, the inner loop terminates and a Stop is created from the subset. The Stop is created without a *place ID* since Places have not yet been identified. Also, Stops with a duration shorter than the duration specified by the `stopDuration` parameter are removed since they are noisy. This is an addition to the algorithms previously described and is mostly used due to the very high sampling frequency which was not accounted for the in original implementation by Cuttone et al. [CLL14].

```
int start = 0;
while (start < n) {
  int end = start + 1;
  List<LocationSample> subset = data.sublist(start, end);
  GeoPosition centroid = _computeCentroid(subset);

  while (end < n &&
    Distance.fromGeospatial(centroid, data[end]) <= stopRadius) {
    end += 1;
    subset = data.sublist(start, end);
    centroid = _computeCentroid(subset);
  }

  Stop s = Stop._fromLocationSamples(subset);
  stops.add(s);

  start = end;
}
```

The distance calculation is carried out using the `GeoSpatial` interface previously mentioned. The distance function `fromGeoSpatial` takes two objects which implement the interface and unpacks the latitude and longitude from these objects. The haversine distance can then be computed afterward.

```
class Distance {
  static double fromGeospatial(_Geospatial a, _Geospatial b) {
    return fromList(
        [a.geoPosition._latitude, a.geoPosition._longitude],
        [b.geoPosition._latitude, b.geoPosition._longitude]);
  }

  static double fromList(List<double> p1, List<double> p2) {
    /// Haversine implementation
  }
}
```

Finding Moves

The Move class has two constructors which are both private. Common for both constructors is that they take two Stops as arguments, with argument being either a path of *LocationSamples* or a distance (a double). The factory constructor called `_fromPath` calculates the distance of the path and then uses the normal constructor to create a Move.

```
factory Move._fromPath(Stop a, Stop b, List<LocationSample> path) {
  double d = _computePathDistance(path);
  return Move._(a, b, d);
}

Move._(this._stopFrom, this._stopTo, this._distance);
```

The normal constructor is used for de-serialization whereas the factory constructor is used to create a Move given two Stops and the path of samples between them.
The algorithm for finding Moves takes a List of *LocationSamples* and the Stops found from the samples as input. The algorithm first checks if the set of Stops is empty, and if so returns an empty set of Moves. If, the set of Stops is not empty, then two 'fake' Stops are created and added to the set of Stops. These two additional stops are created from the first and last element in the set of Location Samples. For each Stop in the set of Stops, it is calculated which samples lie in between the current and next Stop. A Move is then created using the current Stop, the next Stop, and the path between.

```
Stop first = Stop._fromLocationSamples([data.first]);
List<Stop> allStops = [first] + stops;

if (data.first != data.last) {
  Stop last = Stop._fromLocationSamples([data.last]);
  allStops.add(last);
}
```

```
for (int i = 0; i < allStops.length - 1; i++) {
  Stop cur = allStops[i];
  Stop next = allStops[i + 1];
  List<LocationSample> samplesInBetween = data
      .where((d) =>
          cur.departure.leq(d.datetime) && d.datetime.leq(next.arrival))
      .toList();

  moves.add(Move._fromPath(cur, next, samplesInBetween));
}
```

The notion of fake Stops is an addition to the definition in Chapter 3, and are created to avoid edge cases where tracking was started while moving. In such a case no Moves will be created before the user is stationary for some time at least one Stop is found. This situation will likely not be very common, but was found during self-study and therefore deemed worthy of covering. The extra Stops are only used for finding moves and will not be used for finding Places.

Finding Places

The Place class only has one normal constructor which takes an ID (an integer) and a List of Stops.

```
Place._(this._id, this._stops);
```

The Place algorithm takes a set of Stops for a given period, i.e. Stops over multiple days. The DBSCAN algorithm by Ester et al. [Est+96] is used to find clusters in the Stops and label each Stop with a cluster-ID, this is the *place ID* previously discussed. Once the labels are computed, the Stops are grouped by their *place ID*. For each group, a Place object is created with the group label and all the Stops with that label. Lastly, the `placeId` attribute for each Stop in the group is set to the group label.

```
DBSCAN dbscan = DBSCAN(
    epsilon: placeRadius, minPoints: 1,
    distanceMeasure: Distance.fromList);

List<List<double>> stopCoordinates =
    stops.map((s) => ([s.geoPosition.latitude,
      s.geoPosition.longitude])).toList();

dbscan.run(stopCoordinates);

Set<int> clusterLabels = dbscan.label.where((l) => (l != -1)).toSet();
```

```
for (int label in clusterLabels) {
  List<int> indices =
      stops.asMap().keys.where((i) => (dbscan.label[i] == label)).toList();

  List<Stop> stopsForPlace = indices.map((i) => (stops[i])).toList();

  Place p = Place._(label, stopsForPlace);
  places.add(p);

  stopsForPlace.forEach((s) => s.placeId = p._id);
}
```

Hour Matrix Computation

The Hour Matrix is an auxiliary feature used for internal computation and is therefore private. The class is implemented using a 2D double array as a field, representing the matrix of 24 rows, equal to the number of hours in a day, and columns equal to the Number of Places visited on the day. The construction of the Hour Matrix is done with a factory constructor that takes a List of Stops and the number of places visited. From this, the matrix is created and filled out. Each Stop can be converted into an array of doubles that tells which place and how much was visited.
Next, the `routineMatrix()` factory constructor is discussed. This is a method for

```
factory _HourMatrix.fromStops(List<Stop> stops, int numPlaces) {
  List<List<double>> matrix = new List.generate(
      HOURS_IN_A_DAY, (_) => new List<double>.filled(numPlaces, 0.0));

  for (int j = 0; j < numPlaces; j++) {
    List<Stop> stopsAtPlace = stops
        .where((s) => (s.placeId) == j).toList();

    for (Stop s in stopsAtPlace) {
      for (int i = 0; i < HOURS_IN_A_DAY; i++) {
        matrix[i][j] += s.hourSlots[i];
      }
    }
  }
  return _HourMatrix(matrix);
}
```

Figure 5.2: Construction of the Hour Matrix.

creating the RoutineroutineMatrix Matrix of the average day, given a list of other Hour Matrices. The method is quite simple since it uses two for loops to fill out an empty zero-matrix with the average value of each position indexed by `i` and `j`, for each matrix.
Lastly, the `computeOverlap` method is discussed: This method computes the overlap similarity function discussed in Equation (3.21). Another Hour Matrix is provided as parameter referred to as `other` and the current Hour Matrix is referred to as `this` since the method is called on a specific object.
The maximum possible overlap is computed as the minimum of the two matrix sums, since if one matrix is very sparse, then the overlap is severely limited. If either of the sums is zero then -1 is returned, due to either matrix being empty, which is valid. For computing total overlap a sum is used, and the matrix positions are iteration. For each position, the overlap for two scalars is computed and added to the total overlap. The overlap for two scalar values we defined in Equation (3.21) as the minimum value of the two, given that both values are non-negative.

Home Stay Computation

The derived features are computed according to their definitions in Chapter 3, using the lazy evaluation template outlined in Figure E.1. The home stay feature is no exception. The algorithm for computing home stay uses the stops of today: First, the total time elapsed today is calculated using the departure timestamp of the last known Stop of today. Next, the Stops are used to identify the home place by constructing an Hour Matrix and then extracting the `homePlaceId` from the Hour Matrix. Then, the total duration spent at the home place is calculated by summing the duration of

```
factory _HourMatrix.routineMatrix(List<_HourMatrix> matrices) {
  int nDays = matrices.length;
  int nPlaces = matrices.first.matrix.first.length;
  List<List<double>> avg = zeroMatrix(HOURS_IN_A_DAY, nPlaces);

  for (_HourMatrix m in matrices) {
    for (int i = 0; i < HOURS_IN_A_DAY; i++) {
      for (int j = 0; j < nPlaces; j++) {
        avg[i][j] += m.matrix[i][j] / nDays;
      }
    }
  }
  return _HourMatrix(avg);
}
```

Figure 5.3: Computation of the Routine Matrix (i.e. average Hour Matrix).

```
double computeOverlap(_HourMatrix other) {
  assert(other.matrix.length == HOURS_IN_A_DAY &&
      other.matrix.first.length == _matrix.first.length);

  double maxOverlap = min(this.sum, other.sum);
  if (maxOverlap == 0.0) return -1.0;

  double overlap = 0.0;
  for (int i = 0; i < HOURS_IN_A_DAY; i++) {
    for (int j = 0; j < _numberOfPlaces; j++) {
      if (this.matrix[i][j] > 0.0 && other.matrix[i][j] > 0.0) {
        overlap += min(this.matrix[i][j], other.matrix[i][j]);
      }
    }
  }
  return overlap / maxOverlap;
}
```

Figure 5.4: Construction of the Hour Matrix.

the Stops which belong to the home place. The home stay is then calculated as the time at home divided by the total time elapsed.

Routine Index

The routine index is the most difficult to compute by far. The method for computing this feature inside the Mobility Context class is however quite short, but this is due to all the matrix computations being done in the Hour Matrix class, i.e. the averaging and overlapping of matrices. The algorithm first checks if any contexts are provided if not then the routine index should be -1.0. Next, the Hour Matrices for each historic date is computed, and from these, the Routine Matrix is computed. Lastly, the routine index is found by computing the overlap between the two Hour Matrix of today, and the Routine Matrix, using the `a.computeOverlap(b)` method of the Hour Matrix class.

5.2.3 Mobility Context

A *MobilityContext* object represents features for a given date. The class has a private constructor that takes a List of Stops and Moves from today, and a List of Places from the current period, i.e. the last 28 days including today. When the class is instantiated the date of today is automatically inferred, if not provided through the date parameter, which is an optional parameter. This parameter can be overridden

```
double _calculateHomeStay() {
  DateTime latestTime = _stops.last.departure;

  int totalTime = latestTime.millisecondsSinceEpoch -
      latestTime.midnight.millisecondsSinceEpoch;

  _HourMatrix hm = this.hourMatrix;
  if (hm.homePlaceId == -1) {
    return -1.0;
  }

  int homeTime = stops
      .where((s) => s.placeId == hm.homePlaceId)
      .map((s) => s.duration.inMilliseconds)
      .fold(0, (a, b) => a + b);

  return homeTime.toDouble() / totalTime.toDouble();
}
```

Figure 5.5: The method for computing the home stay feature.

```
  double _calculateRoutineIndex() {
    if (contexts == null) {
      return -1.0;
    } else if (contexts.isEmpty) {
      return -1.0;
    }

    List<_HourMatrix> matrices = contexts
        .where((c) => c.date.isBefore(this.date))
        .map((c) => c.hourMatrix)
        .toList();

    _HourMatrix routine = _HourMatrix.average(matrices);

    return this.hourMatrix.computeOverlap(routine);
  }
```

Figure 5.6: The method for computing the routine index feature.

in the case of unit testing for specific dates or if the programmer wishes to compute a Mobility Context for a date in the past.

```
MobilityContext._(this._stops, this._allPlaces, this._moves,
  {this.contexts, this.date}) {
  _timestamp = DateTime.now();
  date = date ?? _timestamp.midnight;
}
```

The other optional parameter is a List of Mobility Contexts is used for computing the routine index - how this is achieved will be explained in Subsection 5.2.4. The 'derived' features are implemented as doubles (except for Number of Places which is an integer) and are fields in the Mobility Context class.

Lazy Evaluation

A *getter* method for a given feature should reflect that the feature that is to be retrieved requires minimal computation (i.e. 'getting'), and therefore the feature computation should not take place in the getter method. However, there is a middle way, since the given feature needs to be computed only once to be evaluated, which allows us to keep the getter syntax. This middle way is *lazy evaluation* described by Fowler [Fow+02] [p. 200], which applied to this example is the idea of postponing computation of a given field until the first time it is needed. After being computed, the field is stored and can be retrieved henceforth without any computational cost. In practice, this is done by letting the field be initialized to *null*, and checking for *null* in the getter method. If the value is *null* then the feature is calculated and the field's value is updated after the computation and the getter can return the field's value. If the field is not *null* then the feature has already been computed and can be returned immediately.

```
class MobilityContext {
    // Field
    double _routineIndex;

     // Getter
    double get routineIndex {
        return _routineIndex;
    }
}
```

Figure 5.7: The getter method for a feature field.

```
class MobilityContext {
    // Field
    double _routineIndex;

    // Getter
    double get routineIndex {
        if (_routineIndex == null) {
            _routineIndex = _calculateRoutineIndex();
        }
        return _routineIndex;
    }
}
```

Figure 5.8: Lazy evaluation of a feature.

5.2.4 Context Generation

The instantiation of Mobility Contexts is done through the Context Generator class, which is the interface between the programmer and the core of the package. All computation and storing and loading of data is done through this class. The class is a static class and thus does not have a mutable state which means that all methods of this class are also static and cannot rely on any internal, non-static values. The central part of the ContextGenerator class is the `generate()` method, which is where a MobilityContext is computed. The method is asynchronous since it requires loading data from the file system before computation can take place. It does not require any parameters to call, but has two optional parameters: `usePriorContexts` is a boolean option to compute the MobilityContext using prior contexts which is false by default. The other parameter is a date parameter, `today`, which similar to the MobilityContext constructor allows the user to override today's date, which is automatically computed if not specified.

```
static Future<MobilityContext> generate(
      {bool usePriorContexts: false, DateTime today}) {...}
```

First, the file system is queried by initializing the three different MobilitySerializers, i.e. one for Location Samples, another for Stops and a third one for Moves. Next, Location Samples are loaded and filtered; any samples with a date different from today are thrown away since they have already been used on a previous day and are no longer relevant to keep. After this the Stops and Moves are loaded from disk and filtered; any Stops and Moves that are either from today or older than 28 days are thrown away.

```
List<LocationSample> samplesToday = await sampleSerializer.load();
List<Stop> stopsHist = await stopSerializer.load();
```

```
List<Move> movesHist = await moveSerializer.load();

samplesToday = _filterSamples(samplesToday, today);
stopsHist = _stopsHistoric(stopsHist, today);
movesHist = _movesHistoric(movesHist, today);
```

The reason for throwing away elements from today is that they need to be recomputing using all the recent Location Samples which can alter the old results. After recomputing today's Stops and Moves, the historical and recent Stops are merged to represent the whole period, and likewise for the Moves. Places are then computed using all the Stops of the period.

```
List<Stop> stopsToday = _findStops(samplesToday, today);
List<Move> movesToday = _findMoves(samplesToday, stopsToday);

List<Stop> stopsAll = stopsHist + stopsToday;
List<Move> movesAll = movesHist + movesToday;

List<Place> placesAll = _findPlaces(stopsAll);
```

Next, the Stops and Moves for the period are stored to disk, but before they are stored, the flush method is used for the serializers in order to delete the old content permanently.

```
stopSerializer.flush();
moveSerializer.flush();
stopSerializer.save(stopsAll);
moveSerializer.save(movesAll);
```

Lastly, if prior contexts are to be used then the historical dates are extracted from the historical stops, and for each date, the Stops and Moves are extracted and used to construct a Mobility Context, with each context being added to a List of prior contexts.

```
List<MobilityContext> priorContexts = [];

if (usePriorContexts) {
  Set<DateTime> dates = stopsHist.map((s) => s.arrival.midnight).toSet();
  for (DateTime date in dates) {
    List<Stop> stopsOnDate = _stopsForDate(stopsHist, date);
    List<Move> movesOnDate = _movesForDate(movesHist, date);
    MobilityContext mc =
        MobilityContext._(stopsOnDate, placesAll, movesOnDate, date: date);
    priorContexts.add(mc);
  }
}
```

The method returns a MobilityContext object using the Stops and Moves of today and the Places for the period. Also, the date of today is chosen to be overridden and the computed contexts are also provided. If no contexts were computed, then `priorContexts` will be an empty List.

```
return MobilityContext._(stopsToday, placesAll, movesToday,
        contexts: priorContexts, date: today);
```

5.3 Using the Package

This section will be a mirror of the official instructions on how to use the package, as of version *1.1.5*. For getting started, the programmer has to download the package by adding it as a dependency in their `pubspec.yaml` file of the Flutter project and running the following command:

```
flutter packages get
```

Once the dependency has been downloaded it can be imported as follows:

```
import 'package:mobility_features/mobility_features.dart';
```

Features are computed using the 3 lines of code displayed in Figure 5.9:

```
/// Collect data with a location plugin
List<LocationSample> locationSamples = ///

/// Store data via the package
await ContextGenerator.saveSamples(locationSamples);

/// Compute Features via the package
MobilityContext context = await ContextGenerator.generate();
```

Figure 5.9: All the three lines of code necessary for the application developer to write.

The exact method for arriving at this stage is outlined in the following 4 steps.

Step 1: Collect location data

Location data collection is, as mentioned, not part of the Mobility Features package, and the programmer will, therefore, have to a location plugin such as `https://pub.dev/packages/geolocator`. From here, each incoming location object has to be converted to a `LocationSample` via the constructor

Below is shown an example where incoming `Position` objects are coming in from the `GeoLocator` plugin and are being handled after saved to a list, in the `_onData()` call-back method.

```
List<LocationSample> locationSamples = [];
...

void _onData(Position d) async {
    GeoPosition geoPos = GeoPosition(d.latitude, d.longitude);
    LocationSample sample = LocationSample(geoPos, d.timestamp);
    locationSamples.add(sample);
}
```

Step 2: Save location data

The location data must be saved on the device such that it can be used in the future. Saving the data to persistent storage prevents it from being lost should the RAM reset. Given that the programmer has a collected the samples in a list, the samples can be serialized using the `save()` method of the MobilitySerializer.

```
await ContextGenerator.saveSamples(locationSamples)
```

Ideally, saving the data is done with a certain interval, such as every time 100 samples are collected.

Step 3: Compute features

The Features can be computed using the static class `ContextGenerator` which uses the stored location samples, as well as historic features to compute the features for today.
There most basic computation is done as follows:

```
MobilityContext context = await ContextGenerator.generate();
```

Note: it is not possible to instantiate a ‘MobilityContext‘ object directly. It must be instantiated through the ‘ContextGenerator.generate()‘ method.

Step 3.1: Compute features with prior contexts

Should the programmer wish to compute the routine index feature as well, then prior contexts are needed. Concretely, the application needs to have tracked data for at least 2 days, to compute this feature. The generation of a Mobility Context using prior contexts is done using the `usePriorContexts` argument and setting it to `true`.

```
MobilityContext context =
    await ContextGenerator.generate(usePriorContexts: true);
```

Step 3.2: Compute features for a specific date

By default, a *MobilityContext* object uses the current date as a reference to filter and group data, however, should one wish to compute the features for a specific date, then it is possible to do so using the `today` argument and providing the desired date.

```
DateTime myDate = DateTime(01, 01, 2020);
MobilityContext context =
    await ContextGenerator.generate(today: myDate);
```

Step 4: Get features

All features are implemented as *getters* for the MobilityContext class and can, therefore, be retrieved using the dot-notation.

```
MobilityContext context = // Generate context

List<Place> places = context.places;
List<Stop> stops = context.stops;
List<Move> moves = context.moves;

int numberOfPlaces = context.numberOfPlaces;
double homeStay = context.homeStay;
double entropy = context.entropy;
double normalizedEntropy = context.normalizedEntropy;
double distanceTravelled = context.distanceTravelled;
double routineIndex = context.routineIndex;
```

CHAPTER 6

Validation

To validate the Mobility Features Package in a real world setting, a small-scale field study was conducted in which 10 participants. For this study a mobile application was developed in Flutter that used the package to compute various features. These features were compared to subjective user-data also collected through the application in the form of a small questionnaire. This chapter will go through the considerations made during developing the app and conducting the study.

6.1 Field Study

A small-scale study similar to that of Cuttone et al. [CLL14] was conducted which ran for 3 weeks and included 10 participants (including the author). The goal of the study was to validate the accuracy of the features produced. In the study the participants used the application discussed in Chapter 5 to collect their location data and computed mobility features daily. In addition, participants answered a questionnaire daily to get subjective user data.

6.1.1 Self-study

Before the main study was conducted a self-study was conducted in three different cities, in order to select appropriate parameters for the algorithms. This parameter tweaking happened while the author was in Munich where he sat in a large university building and visited different offices. In Figure 6.1 the Places and the Stops at the university are displayed and as can be seen which are very close. Had the radius parameter for finding places been higher, then some of the found places would have been merged into a single place. Here, the parameter was set to 25 meters, and in the final study, it was chosen to set it to 50 meters.

6.1.2 Main Study

In addition to this, the application also had a diary consisting of 4 questions that the participant had to fill out each day. In order to make it easy for the user to remember filling out the diary, a reminder was sent to the participants at 8 PM. The time 8 PM was chosen due to being relatively late while still being early enough in the day that people would still be checking their phone. Some people go to bed at 9-10 PM which had to be taken into account. The diary questions were related to 3 of the Mobility

Figure 6.1: A map of TUM Garching, displaying the Places (red clusters) as well as the Stops which make up these places (blue markers) visited by the author.

Features which were *Number of Clusters*, *Home Stay* and *Routine Index*. The purpose of the questions was to get subjective estimates of the values of these features. It is important to stress that answers are very rounded off, since the participants cannot be expected to give very precise answers. Answers are also highly subjective since the definition of things such as a 'place' may vary a lot from person to person.

6.1.3 Evaluating Features

Answers were collected through a diary in order to evaluate the accuracy of the computed features. The features inquired about had to be easy to formulate as a question such that participants could provide reliable answers. Features such as entropy and location variance were ill-suited for this, whereas Home Stay, Number of Places, and Routine Index were chosen instead. The questions were the following:

#1 How many unique places (including home) did you stay at today?

#2 How many hours did you spend away from home today? (Rounded-up)

#3 Did you spend time at places today that you don't normally visit?

#4 On a scale of 0-5, how much did today look like the previous, recent days? (Where 0 means 'not at all' little and 5 means 'Exactly the same')

Collecting the subjective *Number of Places* visited, was done by asking the participant how many places they had visited today, including their home. Technically it is possible that a user visits no places at all, but highly unlikely.

For collecting the subjective *Home Stay* percentage, the participant was asked the inverse question, i.e. how many hours they were *away* from their home today (from which Home Stay can then be calculated later). This question is much easier for the participant to answer, and there is no need to explain to the participant that time spent during the night counts towards the Home Stay, as an example.

The Routine Index was more difficult to formulate as a question since there is no succinct way of putting it. It was decided upon rating today scale from 0 to 5, where 0 indicates that today looks nothing like previous days and 5 indicating that today looks identical to the previous days. Ideally, the scale should be more fine-grained such as a 0 to 10 scale, however, this would put too much on the participant. The information we wished to know was, on a high level, whether or not today looked like the previous days which is still possible with a more coarse grained scale. Question #3 also related to the Routine Index feature but was not used for later data analysis since it was hard to compare directly to the Routine Index and would require looking at the Hour Matrix instead.

6.1.4 Hindrances

The Coronavirus pandemic leads to countries closing borders and urging people to stay at home as much as possible. This included workplaces shutting down and people had to work from home, as well as places of recreational character such as gyms and restaurants. This had some implications for the study in that people would spend most of their time at home, and they would likely not visit very many places. In addition, it was probably also not common for most people to go visit new places during the pandemic. However, all in all, the pandemic only shaped the results of the field study and did not prevent the study from taking place at all. In addition the Android platform also closed down the access needed for tracking location data in the background without interruption as of February 2020 [1,2]. It was therefore chosen to focus on releasing and testing the application for iOS only, due to time constraints.

[1] `https://nakedsecurity.sophos.com/2020/02/25/android-11-to-clamp-down-on-background-location-access/`

[2] `https://developer.android.com/training/location/background`

6.2 Study Application

This section will describe the study application in terms of components and will illustrate how the application looked.

6.2.1 Installing the Application

The application was only released on iOS as previously mentioned and therefore was distributed exclusively via the Apple App Store. When beta testing iOS applications one can use the TestFlight service provided by Apple which essentially functions a separate App Store for applications in development, and require an invitation to install. Once the user had such an invitation the installation process was the following: The user installs the TestFlight application via the App Store, then opens the TestFlight application to find the Mobility Study app ready for installation. Once the Mobility Study app is installed, it will ask for permission to track the user's location as well as sending the user notifications. The location tracking is obviously necessary for the collection of location data. The notifications are not necessary, but do help the user be reminded to fill out a daily questionnaire. An installation manual (see Appendix G) was provided to the participants to ensure the applications were set up correctly. The installation process is shown in Figure 6.2.

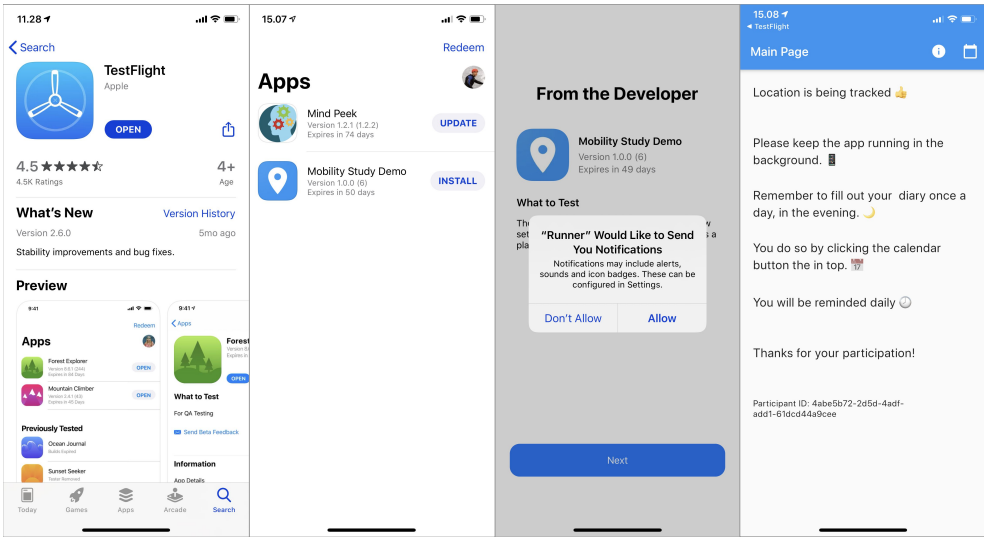


Figure 6.2: The initial installation- and setup process for the user.

6.2.2 App Screens

The Main Page of the application displays a list of instructions and does have any interesting functionality. From the main screen it is possible to navigate to the Info Page and the Diary Page, from the navigation bar in the top.

The Info Page is made to inform the user of how the data will be used, and an email to contact the researcher in case of any questions.

The Diary Page contains the questionnaire the user has to answer daily. The user can either navigate to this page themself or tap the notification they receive each evening. The four questions can be answered by pressing the Answer button which will show a wheel of possible values to pick an answer from. Once all questions have been answered, the submit button will be enabled. Submitting the answers will upload the answers to a Firebase server, and store them on the device as well. When storing is done, the last screen will appear which informs the user the answers have been saved and thanks them for their contribution.

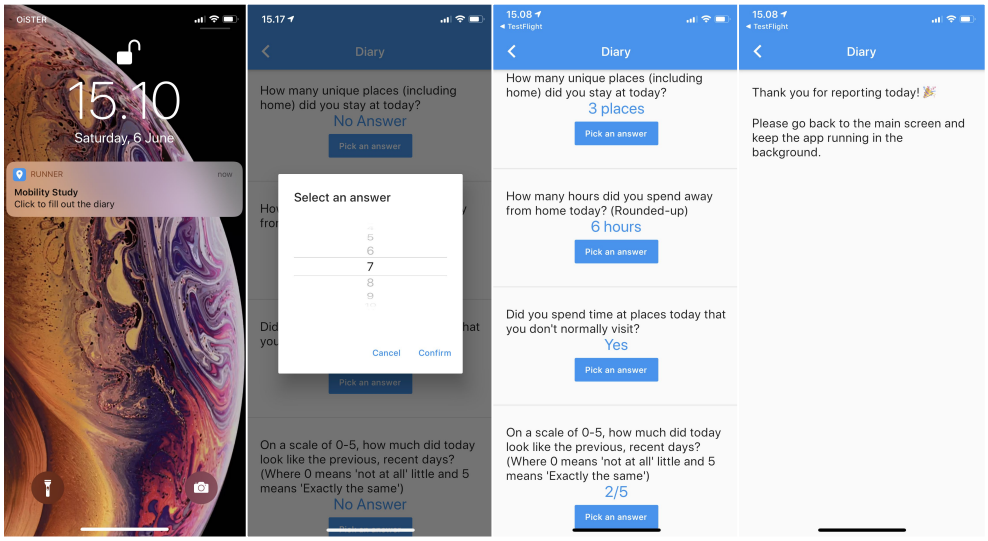


Figure 6.3: The different screens which the user is taken through for submitting answers.

6.2.3 Displaying Features

An initial version of the application included a display of the real-time calculated features, which were recalculated each time a button was pressed. It was decided to not display the features in the final version for the study since they would influence the answers given by the users. For example, if the application would state that the user has visited a certain number of places today, it is highly unlikely that a participant would report something else if they are the least in doubt themselves. This display of features may be relevant (at least for some of the more features which are more easily interpret-able such as Home Stay) for a real-world application where it makes sense to inform the user exactly what goes on behind the curtain.

6.2.4 Data Storage

To store the data from the study online, such that it could later be extracted for data analysis, a Firebase file storage server was used for uploading files multiple times daily. Concretely, the LocationSamples, Stops, and Moves were stored locally on the device. Whenever a Feature calculation was triggered, the calculated MobilityContext was serialized and uploaded as a file, in addition to the data points for the day and all Stops and Moves on the phone for last 28 days (see Chapter 3). This process was very wasteful in the sense that only 'new' data points needed to be uploaded, but instead, the whole file was just overwritten. This was mainly done to ensure little data loss and avoid inconsistencies between the online file and the local file.

6.3 Application Implementation

This section will describe the implementation of the study application, mostly regarding how data was collected, how features were computed as well as how the data was sent to a Firebase instance. The study app used the package while it was in an earlier iteration. In this iteration, almost none of the logic related to storing and loading data was part of the package, and as such all of this had to be written in the application instead. This section will provide source code examples of how the application should be implemented with the new API since the old version is deprecated. Largely, the data flow of the study app has not changed but the concrete implementation has, in the sense that much fewer lines of code are needed.

6.3.1 Architecture Overview

To provide a high-level overview of the different components which make up the study application, a high level component diagram is displayed in Figure 6.5. The MobilityStudy component in blue is the component responsible for managing the application state but does not do much outside of this since the application state management required is minimal. Had it been a more complex application with many different

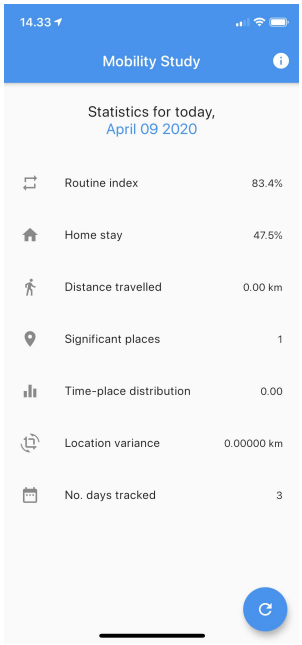


Figure 6.4: An early iteration of the study app in which the feature values were shown.

screens and a state which had to be maintained across these screens (for example a shopping cart in a shopping app) then more logic would lie inside the MobilityStudy component. Instead, the Main Screen is spawned from the MobilityStudy component which in turn creates an AppProcessor instance. The AppProcessor instance is responsible for a multitude of tasks, such as asking for permissions, collection location data, and computing features. Storing and loading from the disk is done through the FileManager component which includes location data, Stops, Moves, MobilityContexts, and diary answers. This component is also responsible for uploading the stored data to Firebase.

6.3.2 Custom Location Plugin

For collecting Location Samples, a custom version of the *Geolocator*[3] plugin was developed for the purpose of this package to achieve reliable background location streaming. The current implementation of *Geolocator* was missing a flag in the *Objective-C* implementation for iOS, which allows the app to continue streaming location data while

[3] `https://pub.dev/packages/geolocator`

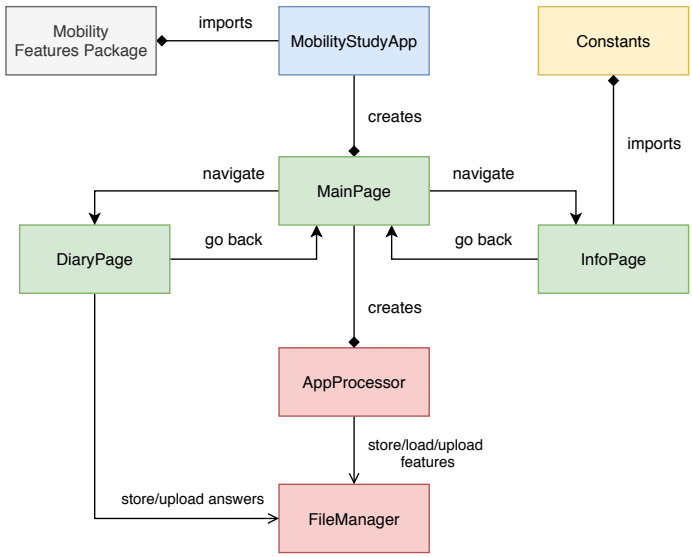


Figure 6.5: Component diagram for the study application displaying the different building blocks and the interactions between them.

the app is minimized. The flag for background updates has to be set for an instance of a Location Manager which is the access to the Location API:

```
_locationManager.allowsBackgroundLocationUpdates = YES;
```

If this flag is not set to 'YES' (i.e. True), the location stream will die shortly after the application is minimized. It is important to note that this plugin is not part of the Mobility Features Package, but is likely needed to make use of the package. A Github issue was created[4] and the features were merged to a development branch for the GeoLocator plugin. The functionality is however not public as of yet, and the custom plugin was, therefore, necessary to use when developing the application.

6.3.3 Collecting Location Samples

The custom Geolocator plugin was used to set up a stream of location data. The DTO of the plugin called Position, and contains latitude, longitude, and timestamp, in addition to other GPS information. The stream is set up with a subscription using a call-back method that is invoked every time a Position object is generated by the stream.

[4] `https://github.com/Baseflow/flutter-geolocator/issues/390`

```
await _geoLocator.isLocationServiceEnabled().then((response) {
  if (response) {
    streamingLocation = true;
    _subscription = _geoLocator.getPositionStream(options).listen(_onData);
  } else {
    print('Location service not enabled');
  }
});
```

This call-back method is the `_onData` method which is responsible for saving the collected data. It does so by first converting the Position DTO object into a LocationSample DTO object, and then adding it to a buffer. This buffer is implemented as a List of LocationSamples and when the number of samples in this buffer exceeds 100, the content of the buffer will be stored to disk via the `MobilitySerializer`. Afterward, the buffer is emptied, and the process starts anew.

6.3.4 Firebase Services

Firebase file storage was used to host all the data generated by the study application. File storage hosted on a centralized server made it easy to oversee the study and check in on users to see if they remembered to provide answers and track their location.

Additionally, the Firebase Cloud Messaging (FCM) service was set up such that push notifications could be delivered to the participants' phones via the application, to remind them to fill out the diary. Push notifications are another alternative to local notifications, the latter of which is scheduled using alarm-based triggers. Push notifications are a great tool as a developer since notifications are sent out from a centralized server and can be edited at any time without access to the physical phones. In was sometimes useful to send out notifications to specific users if they forgot to fill out many days in a row.

6.3.5 Computing Features and Async Calls

Every time a buffer has been stored to the disk 5 times, features are computed. This was done simply to ensure features were computed regularly in real-time (i.e. with an incomplete dataset) during the study. This was an arbitrary trigger set up to ensure that features were computed multiple times per day. In addition, whenever the user navigates to the diary page, features are computed such that they are generated close to answers being given. One concept not discussed much so far is the need for asynchronous computation and the use of multi-threading. Flutter applications support multi-threading, which means the main thread runs the user interface, and background threads can be spawned in order to perform heavy computation which would otherwise block the main thread, which means the user would experience a

```
void _onData(Position x) async {
  GeoPosition gp = GeoPosition(x.latitude, x.longitude);
  LocationSample sample = LocationSample(gp, x.timestamp);
  _buffer.add(sample);

  if (_buffer.length >= BUFFER_SIZE) {
    /// Save buffer locally, empty it,  then upload data
    await _sampleSerializer.save(_buffer);
    _buffer = [];
    await FileManager().uploadSamples(uuid);

    /// If enough data has been collected, evaluate features
    numberOfBuffers++;
    if (numberOfBuffers >= 5) {
      numberOfBuffers = 0;

      /// Offload computation to background, do not await
      _computeFeaturesAsync();
    }
  }
}
```

Figure 6.6: The *_onData* method responsible for handling incoming Location DTOs.

frozen UI. In the study app, there was no real user interface so to speak, but in a real-world application, there will be a user interface that cannot be allowed to freeze due to the feature calculation. In Flutter threads are referred to as Isolates which communicate using a *SendPort* and a *ReceivePort*. These two objects can be used to transfer other objects between threads, such that the main thread can request a background thread to calculate the features, and the background thread will then send back a *MobilityContext* object once finished.

```
Future<MobilityContext> _computeFeaturesAsync() async {
  ReceivePort receivePort = ReceivePort();
  await Isolate.spawn(_asyncComputation, receivePort.sendPort);
  SendPort sendPort = await receivePort.first;

  MobilityContext mobilityContext = await _relay(sendPort);
  return mobilityContext;
}
```

The `_relay` method works as an interface between the `_computeFeaturesAsync`

method which runs on the UI thread and the static method `_asyncComputation` which runs on the background thread and simply relays messages between the two threads.

```
Future _relay(SendPort sendPort) {
  ReceivePort receivePort = ReceivePort();
  sendPort.send([receivePort.sendPort]);
  return receivePort.first;
}
```

Lastly, the `_asyncComputation` method is static which is due to the computation being done in a separate thread than the main thread. If the objects contained within the *AppProcessor* instance (i.e. in the main thread) were to change their state while computation was ongoing in the background thread, then the resulting computation would produce an outdated result. The *ContextGenerator* is also a static class without a mutable state and can, therefore, be used to compute the Mobility Context without the need for mutable state anywhere in the chain, once the message reaches the background thread.

```
static void _asyncComputation(SendPort sendPort) async {
  ReceivePort receivePort = ReceivePort();
  sendPort.send(receivePort.sendPort);
  List args = await receivePort.first;
  SendPort replyPort = args[0];

  MobilityContext context =
      await ContextGenerator.generate(usePriorContexts: true);

  replyPort.send(context);
}
```

CHAPTER 7
Results and Discussion

This chapter will cover the result of the analyses from the study described in Chapter 5. Lastly, a discussion of these results, as well as different aspects of the Mobility Features Package and general limitations will be presented.

7.1 Data Analysis of Study

As discussed, the participants had to fill out a small questionnaire every day in the evening and these answers were then matched against the computed results. The study resulted in a dataset with 205 days of data, corresponding to 2.51M timestamped location data points, spread over 10 participants. Table 7.1 shows the overview of the data collected for each participant, including the number of points, number of days and the storage requirements for the collected data.

Similar to Cuttone et al. [CLL14], the participants have been anonymized as *P1* through *P9*, and the author has been marked as *R* (researcher).

The answers were of a different format than the computed features and therefore had to be converted into scalars such that they could be compared directly to the features. For the Home Stay feature, the answer the users gave was the number of hours away from home, at the time of registering. It was assumed that most people be registering

P	Days	Samples	MB	Samples/day	MB/day
P1	23	181	17.3	7878.8	0.8
P2	25	142	13.6	5684.4	0.5
P3	21	101	9.7	4850.7	0.5
P4	22	98	9.4	4494.9	0.4
P5	14	209	20.0	14977.4	1.4
P6	26	101	9.7	3922.8	0.4
P7	23	111	106.4	48525.0	4.6
P8	15	141	13.5	9417.3	0.9
P9	12	51	4.9	4311.7	0.4
R	24	365	34.9	15233.6	1.5

Table 7.1: The overview of collected Location Samples for each participant in the study.

at home since the diary was filled out in the evening. Therefore for calculating the Home Stay value from a given answer a, equation 7.1 was applied:

$$h = \frac{t - a}{t} \tag{7.1}$$

Here, t refers to the timestamp at which the diary was filled out.
For the *Routine Index*, the answer (a number of 0 to 5) was transformed to scalar between 0 and 1, i.e. let the scale value be $s \in \{0, 1, 2, 3, 4, 5\}$, then the corresponding *Routine Index* is calculated using equation 7.2.

$$r = \frac{s}{s_{max}}, \quad s_{max} = 5 \tag{7.2}$$

Regarding data collection, figure 7.2 displays how much data was collected per participant. From this figure, it is clear that some participants did not collect nearly as much data as others, and because of this their computed features are expected to be more inaccurate. It was also discovered that the number of stops found is not necessarily correlated with the number of location samples found, as can be seen for participant P7. This can be due to reasons such as moving around much less, which results in fewer but stops with a longer duration being found.

The average size of a serialized *LocationSample* was compted based on file size to be around 100 bytes, with a stop is being 140 bytes. With this information, it is possible to calculate the compression rate for each participant, i.e. how much their dataset was reduced by converting location samples into stops, and this is shown in Figure 7.2. The number of valid days for a specific feature is the number of days for which the user gave answers, and the feature could be calculated. If the user for example turned off their phone during the night, the Home Stay feature could not be calculated, but the routine index feature and the number of places will likely still have been calculated.

7.1.1 Subjectivity of Answers and Ground Truth

A thing to keep in mind is that the participants' answers are not ground truth, and there are multiple reasons why a participant's answer may be inaccurate. Firstly, people's subjective recollection is not necessarily as accurate as they think it might me. Secondly, it was not communicated explicitly to the participants what exactly counts as a place, and how their 'routine' is calculated - this means that participants may have filled out the questionnaire differently, given the same ground truth data - for example, whether or not being in the garden counts as being home. Thirdly, some users misunderstood the routine scale, and users whose data looked strange were contacted afterward to enquire about whether they had misunderstood the question, and if so the answers were corrected as much as could be done. Lastly, the hour away from home and routine index answers were very course-grained, and therefore the

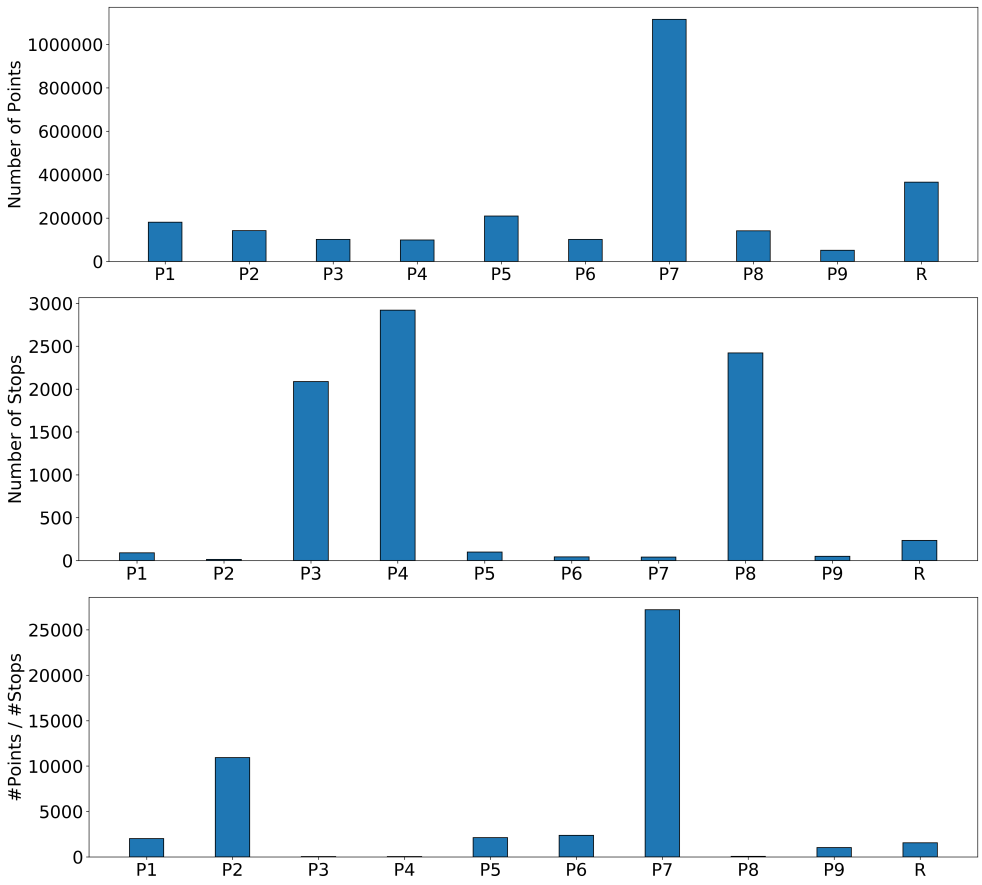


Figure 7.1: The number of raw location samples (top) and the number of stops (middle) collected, and the ratio between the two, for each participant.

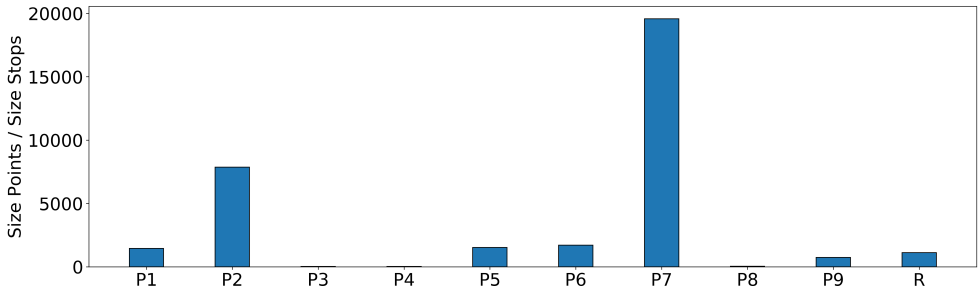


Figure 7.2: The ratio between location samples and stops expressed in terms of storage, in MB.

participants were probably rarely able to give exact answers, according to their own recollection.

7.1.2 Missing Location Data and Answers

The sampling was set to once per second, yet data was not sampled uniformly. In fact, there were many gaps in the collected data. Figure 7.3 shows the data of the author collected during the study over 24 days. The author was very diligent in tracking his location and yet substantial gaps in the data can be observed. These gaps are consistent in the morning where the phone is not moving around, but after 6 AM it seems there is no pattern for the gaps. This is a problem for computing the features, mostly the Routine Index feature due to relying on computing the overlap between days. The current implementation of the feature will ignore time-slots for which data is missing, which leads to a biased Routine Index.

Regarding the subjective user data, many users missed filling out the diary on several days. Figure 7.4 shows the number of total days where a participant participated in the study against the days on which they provided an answer. It was only possible to calculate the error between computed features and the answers on day, if the participant gave an answer. This means that for some participants, a lot of data was thrown away for the data analysis. As an example, it was considered whether or not to entirely get rid of P9 since he/she has very little data both terms of total days collected and even fewer in terms of days with answers.
Another problem that was encountered was when participants would fill out the questionnaire during the night, or the very next morning due to forgetting the previous evening. This meant the answer for day n would be listed as date $n+1$, i.e. the wrong date. This was rectified by changing all answers given between 00:00 and 10:00 to the previous date at 23:59:59. In addition, some users entirely forgot certain days

but remembered it the following day, and reported it manually to the researcher, and these answers were then added manually.

7.1.3 Feature Evaluation

The Home Stay feature required the participant to track their location during the night, as previously mentioned. This meant that if they did not, the Home Stay feature could not be evaluated for that particular day, however other features might still be available for computation, such as the Number of Places. This meant that the number of days for which a given feature can be compared to the answers is not necessarily the same for all features.
The Home Stay feature is calculated by using the *tracked* time at home, divided by the total time elapsed since midnight, and therefore a small gap in the data will make the feature undershoot. It is therefore to be expected that this feature will lie somewhat lower than the answer given by the user since there will be gaps in the data.
The day-by-day results for the author are displayed in figure 7.5 and from this plot, the computed features have a high correspondence with the provided answers. However, the feature computation is based on the author's own definitions which mean the computed features and answers of the author are also expected to be highly correlated. It is therefore relevant to show another participant, namely participant P8, who was very diligent in answering and tracking their location data. For this participant, very promising results were also produced which can be seen in Figure 7.7.

7.1.4 Measuring Errors

The RMSE was computed for all three features across participants and are shown in Table 7.2. We observe the following:

- The Number of Places predicted deviates with almost 1 place daily.
- The Home Stay percentage deviates with 14% daily.
- The Routine Index deviates with 22.5% daily. Here it is important to note than the Routine Index answer was given on scale with 20 percent increments which means that the 22.5% RMSE is equivalent to one 'step' one the scale.

To say something about whether or not the features undershoot or overshoot, the mean error was calculated for each participant as as $ME = \frac{\sum(F)-\sum(A)}{N}$ (where N

	Number of Places	**Home Stay**	**Routine Index**
Mean RMSE	0.99	14.27 (%)	22.5 (%)

Table 7.2: The mean RMSE computed for all participants.

is the total number of days for the specific feature, for the participant). If the mean error is positive, the feature overshoots compared to the answered value, and vice versa if the error is negative. Since the mean error has a sign, i.e. either positive or negative, the 'mean' mean error cannot be computed. Instead, the individual mean error for each participant is plotted in Figure 7.8. From this figure we observe the following:

- The Number of Places feature undershoots by around 0.6 places or lower for most participants, with P7 being an outlier with -1.5 places. It is likely that the undershooting stems from gaps in the data, which can lead to places not being detected due to missing signal.
- The Home Stay feature undershoots for 8/10 participants, most of these having an error under 10%. For those two participants where the feature overshoots, the error is also within 10%. It is highly likely that the undershooting stems from gaps in the data which will lead to an underestimation of the time spent at home.
- The Routine Index feature is mixed with an equal number of participants for which it overshoots and undershoots. It is likely that both gaps in the data as well as subjectivity on a user basis leads to this varying result.

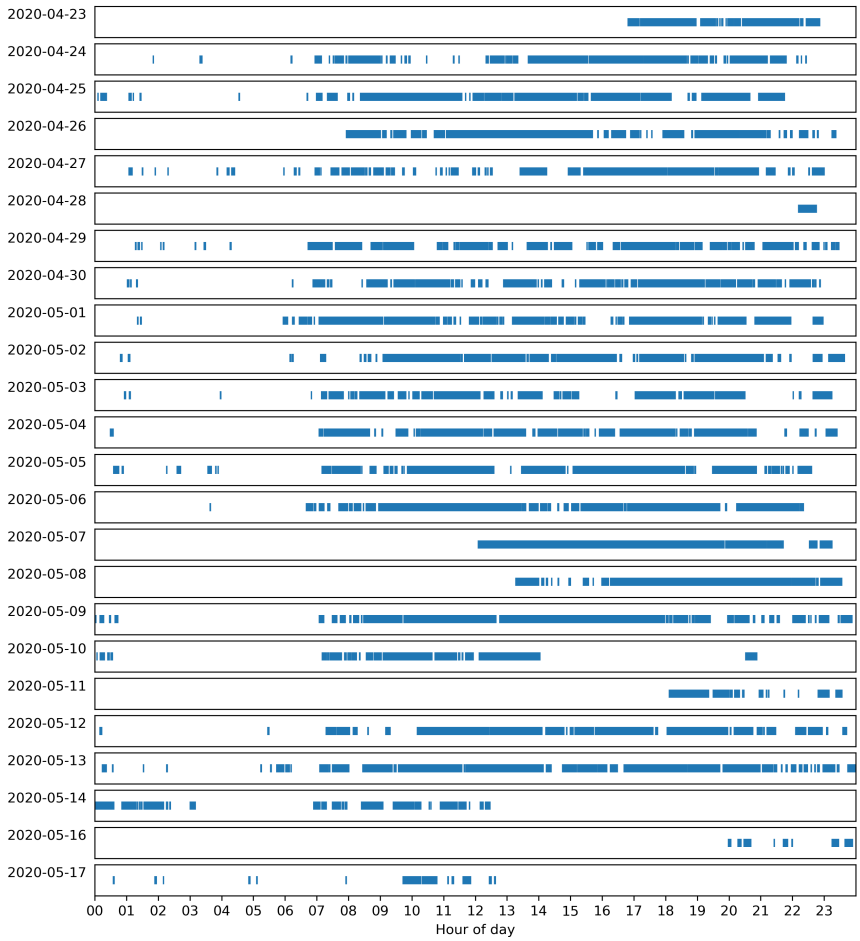


Figure 7.3: An event plot for the author's sampled location data, each day of the study. As can be seen there are quite a few gaps in the data which will make the computes features less accurate.

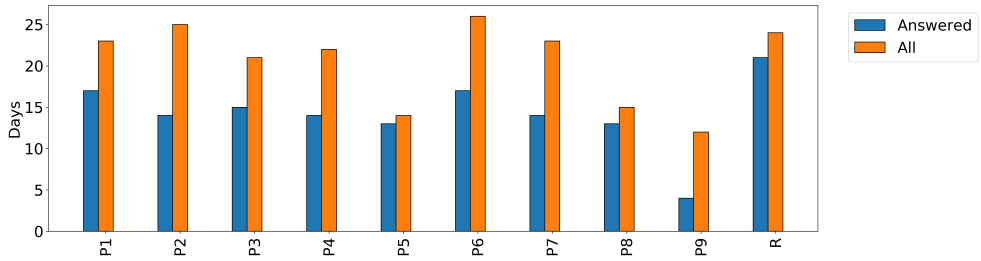


Figure 7.4: The total number of days of participation vs the days for which the diary was filled out by the participant. As can be seen, some users often forgot to give an answer.

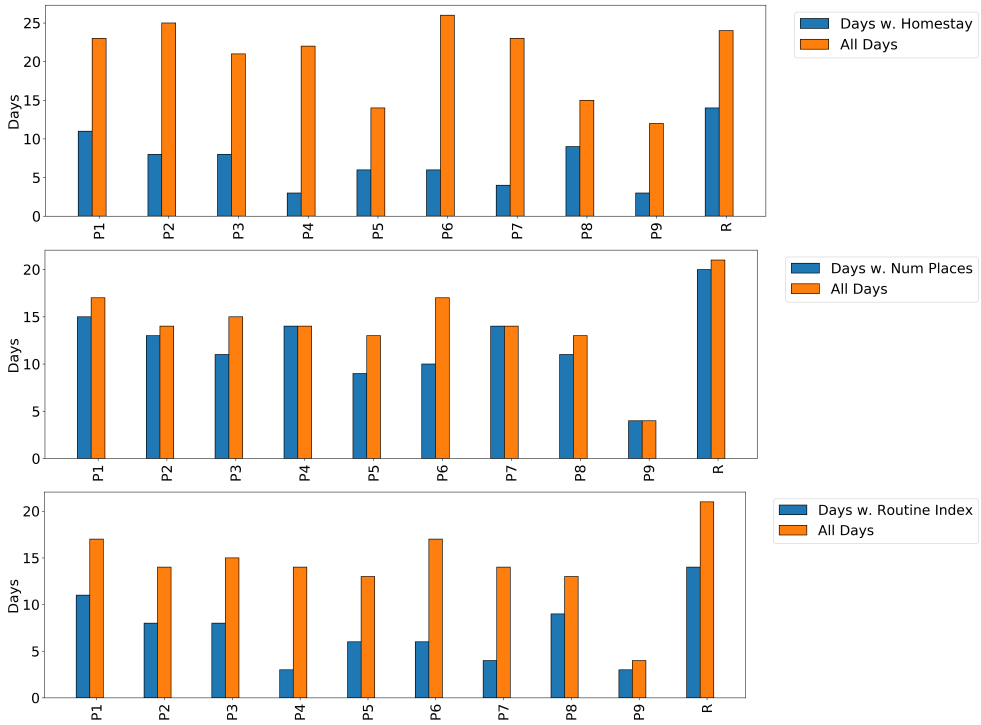


Figure 7.5: The days for which a given feature could be evaluated, out of the total days for which an answer was given and features were computed. As can be seen from the top plot, participant P4 had very few days where the home stay could be computed in comparison to the total number of days tracked which are above 20.

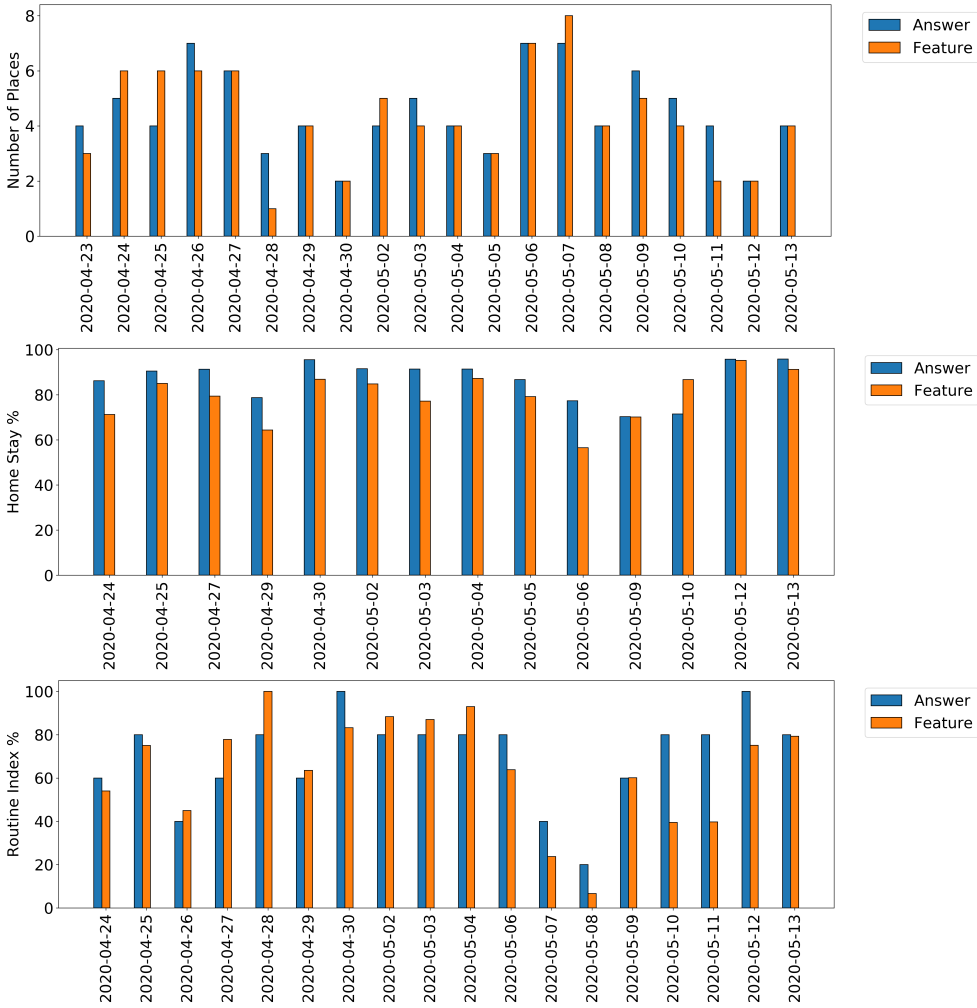


Figure 7.6: The answered and calculated data for each day, for the author.

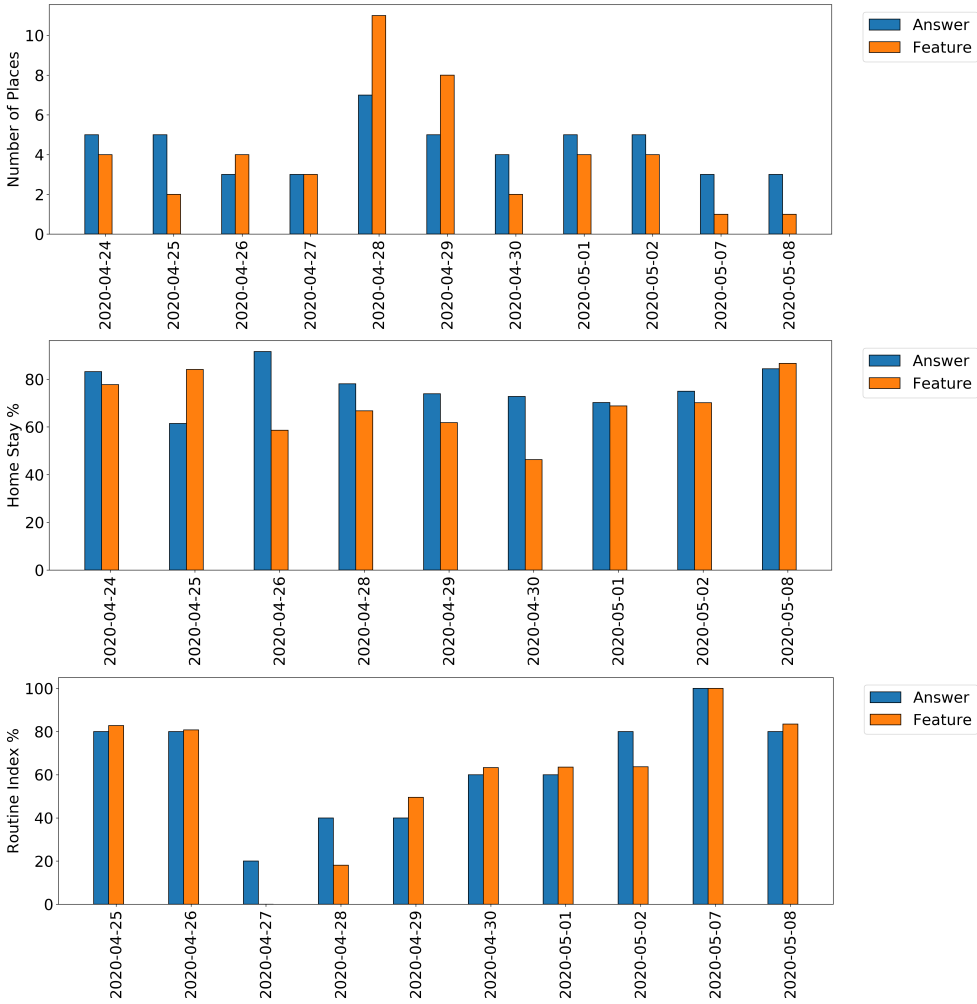


Figure 7.7: The answered and calculated data for each day for participant P8.

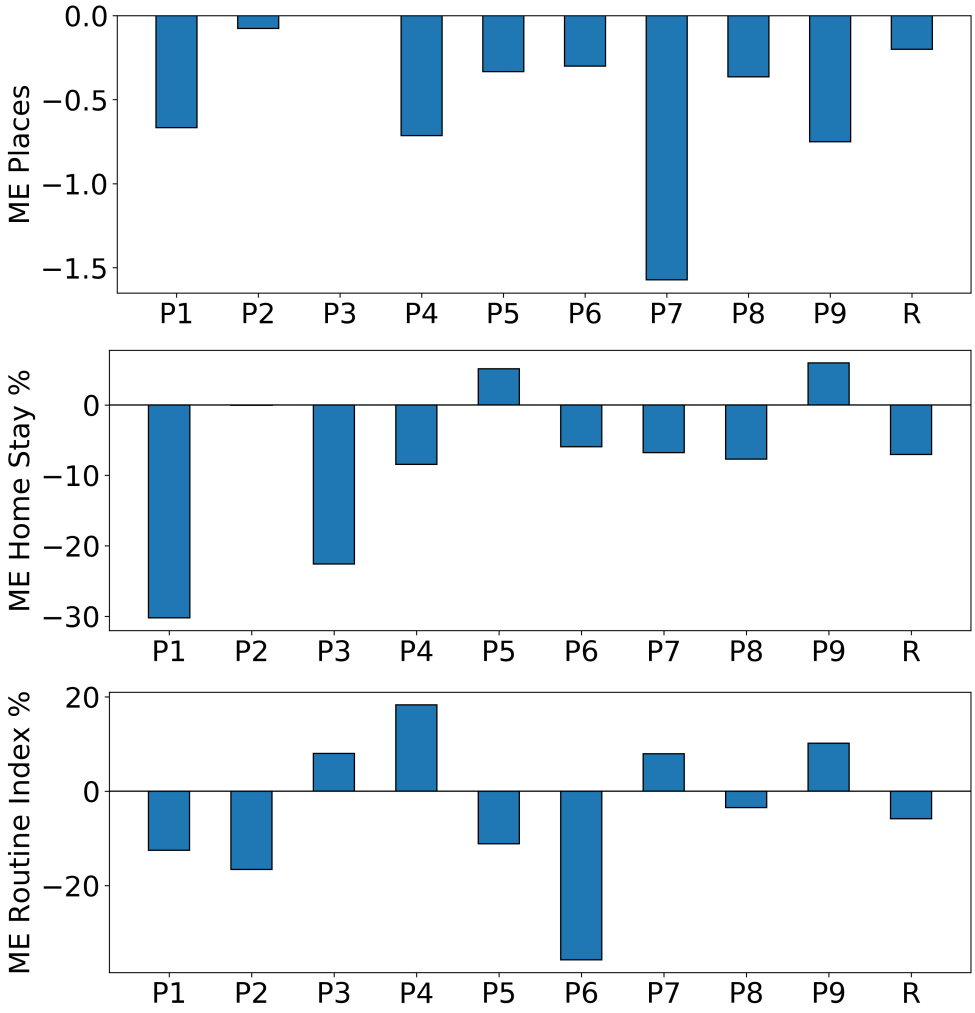


Figure 7.8: The mean error for each participant for each of the three features. A negative mean error means the computed feature undershoots, and a positive error means it overshoots.

7.2 Future Work

The application development process led to many of the improvements of the package API reflected in Chapter 4 and 5 however, a couple of issues remain both in terms of the API and the feature accuracy. In addition it is the plan to integrate the package into existing projects, such as CAMS [Bar20] and MUBS [Roh+20].

7.2.1 Missing Data

As discussed prior in this chapter, this are substantial non-uniform gaps in the collected location data, even for the author. This is a common occurrence for 'in the wild' studies [Pal+17]. Many variables can influence the background location tracking in smartphones, such as the availability of GPS signal, battery, as well as available memory on the phone. The author had a relatively new phone (iPhone XS) and was diligent in making sure the phone was tracking the location as often as possible, yet still has significant gaps in his data. This means other participants that were less diligent or had older phones have even more gaps is still missing in certain periods.

Feature values can be influenced to a high degree by missing data, especially the Routine Index. Currently the Routine Index feature ignores periods of missing data when computing the overlap between the routine matrix and today's hour matrix, however a lot of gaps in the data will result in the feature overshooting. The solution for this missing data problem is either to use a dedicated GPS receiver, or to use an imputation strategy for missing data. Imputation is most likely the best solution, and can be carried out by filling out gaps using the saved stops and moves as historical data. This may likely bias the features however it will be a promising alternative to the current method.

7.2.2 Data Collection and Parameter Tuning

The field study resulted in a dataset of 2.5M data points which means there would have been an opportunity for tuning the parameters if time allowed. Parameter tuning would increase the accuracy of the features produced by the algorithms and thereby increase its usefulness. Some features vary greatly in their accuracy from participant to participant and it is unknown whether that is down to the commitment of the participants and how diligent they were in tracking their location, as well as their answering, or whether the algorithms simply do not consider certain commonly occurring edge cases.

Many days are without a subjective answer meaning they have been thrown away in the data analysis. Rather than relying on participant answers, a researcher could in theory manually label each participants data sets with appropriate feature values. Certain features such as the number of places visited will be easier to manually detect,

compared to to other features such as entropy. If this was done, then finding optimal parameters for the algorithms would be a machine learning task.

Regarding the study, to improve the validity of the answers given by participants in a future study, a short introduction should be given to each participant prior to starting the study. The introduction should define what counts as a place, what counts as their home, and how a routine is defined. This was not done for the field study in this thesis which is likely a cause for a lowered quality of the subjective answers. If the participants are instructed beforehand, one can hope that answers become more consistent and the results of the data analysis will be more informative.

7.2.3 Routine Index

An issue that was not considered for this thesis is the fact that the routine on weekdays differs a lot from the routine during the weekend. This is especially true for people who spent 8 or more hours at work during the weekdays and spent those 8 hours somewhere else during Saturday and Sunday. For such a person the *Routine Index* cannot exceed $\frac{2}{3}$ due to a third of the hours in a day being spent at a different place than their usual place. Even weekdays may look slightly different from one another, especially for those who are part of sports clubs which meet during certain days of the week. In future work, it would be relevant to examine whether comparing Mondays to Mondays, Tuesdays to Tuesdays, etc. would yield a more accurate Routine Index.

In addition, the Routine Index should not rely on a full day of data if it is to be calculated in real-time. To make the feature work best on an incomplete day of data, it should reflect the routine of the user up until the current time of the day. This means if it is calculated at 14:00 then it should only take into consideration the data from the first 14 hours from previous days as well. This would result in the feature likely being very high early in the day, since people usually sleep the same place, but then may get lower as the day progresses. This varying routine index can be useful in real-time context, where triggers can be set up based on the *Routine Index*, ex to alert the user when the value exceeds a certain threshold a certain threshold.

As mentioned in section it would also be highly relevant to incorporate moves into the Routine Index, such that commutes will be taken into account when computing the feature. It not very clear how this could be represented with the existing definition of the Hour Matrix, however one possibility is making an Hour Matrix for moves only, showing which travels were taken at which time. The routine index could then be computed from both the *stop hour matrix* and the *move hour matrix*.

7.2.4 Forced Daily Computation

Currently, the implementation throws away Location Samples from previous days when computing the MobilityContext for today. This approach assumes that any data left from a previous date has been transformed into stops and moves, and therefore no longer is needed. However does not consider the case where a large part of the stored Location Samples have not been used, due to no computation having taken place. In some cases, whole days of Location Samples may end up being thrown away without stops and moves being computed from this data. The current way to avoid this is to compute features every day, making sure at least one computation takes place late in the evening such that minimal data is lost. Another way around it is to override the date, it is known that computation did not take place for a given date. This 'latest date of computation' can be kept track of by the programmer, but goes back to the problem of managing complexity.

Ideally, this is done by the package itself, and can be solved by first grouping location samples by date when loaded. Next, stops and moves are computed for each of the dates. Lastly, the stops and moves are saved to the disk and all the location samples from prior days can be thrown away.

7.2.5 Asynchronous Computation

The asynchronous computation is cumbersome to set up and takes over 30 lines of code to perform. This should ideally be moved inside the package in the next iteration. Another improvement to make is not relying on lazy evaluation, as discussed in Chapter 5. In the current version, only the stops, moves and places are be computed in the background thread. All remaining features are computed inside the MobilityContext object which have not yet been evaluated at the time the object is constructed, due to lazy evaluation. This means the remaining features are computed synchronously upon request, which will likely happen in the main thread. This could lead to freezing the UI-thread, and the fix for this is to compute all features in the constructor of the Mobility Context class. The trade-off will be that it takes longer to compute the features in the asynchronous call, but there will no need to compute the features in the main thread.

7.2.6 Example Application

The Dart package manager, Pub, requires packages to have an example application to demonstrate its usage. Since the study application used an old version of the package API and does not display data, it should probably not be used further. Instead, the old version of the study app displayed in 6.4 is a good candidate for an example app since it presents the calculated features to the user and can be implemented dynamically were features are constantly recomputed and updated.

7.2.7 Integration and Maintenance

The package fits into the *CARP Mobile Sensing Framework* developed by CACHET, as previously mentioned and will, therefore, continue to exist beyond this thesis. An integration into CAMS was not made as part of this thesis due to time constraints and the scope of the thesis. The package will be maintained by the author, who will be employed at CACHET as a research assistant. In additional, the MUBS recommender system by Rohani et al. [Roh+20] is a smart-phone application used for treatment of bi-polar patients through recommendation of pleasant activities. The system does so by tracking patients' prior engaged activities and which the patients rate through the app manually. By using the mobility features we aim to add mental state and behaviour prediction to improve the recommendation algorithm, with these features being automatically generated.

CHAPTER 8

Conclusion

For the conclusion we shall address the original three research questions which made up the hypothesis:

Which mobility features are relevant to include in a software package?

The features Number of Places, Home Stay, Entropy/Normalized Entropy, Location Variance, and Routine Index were chosen based on the work by Saeb. et al [Sae+15b] and Canzian et. al [CM15]. The most important features pertaining to depression were *Home Stay*, *Entropy/Normalized Entropy*, and the *Routine Index*. All features except for the *Routine Index* can be evaluated daily without the need for historical data. In addition, a set of intermediate features, namely *Stops*, *Places*, and *Moves* were also implemented as part of the package. These intermediate features aided in reducing the amount of data processed by the feature-extraction algorithms and proved useful as features themselves.

How can these features be computed in real-time, on a smartphone device?

A combination of mathematical definitions and clever data storage and loading was necessary to achieve real-time feature computation: All features are computed using Location Samples collected from the current day. Certain mathematical re-definitions had to be made for the features such that they could be computed given a dataset from an incomplete day. A novel definition for the *Routine Index* feature was made which required *Stops* from multiple days, stored on the device, to compute the feature. These *Stops* must be saved on the device and loaded whenever the feature computation takes place. *Stops* effectively work as a compressed form of Location Samples and thus reduces the storage and allowed feature computation to be possible in real-time.

How does the design of such a software package look like?

It was decided upon a design that provides an API with a high abstraction level that hides most of the feature computation implementation from the user. This design allows the programmer to compute the features with just 3 lines of code, excluding data collection. The implementation details hidden away from the user included storing and loading historical data as well as computing features using the historical- and daily data. The package does not depend on any specific location plugin due to its design, which allows the programmer to flexibly choose their own plugin for tracking location data. Being independent of any specific location plugin

enables easier maintenance and allows the package to be used among other packages dependent on location tracking, without causing dependency issues.

Validation of the Package

Through a field study with 10 participants, the capabilities of the package were demonstrated. For this study a Flutter application collected the participants' location for 3 weeks and used the package to compute the participants' features multiple times daily. Participants also filled out a daily questionnaire pertaining to the features, which were compared to the computed features. In the 3-week study, the following insights concerning the Mobility Features Package were drawn:

- The Mobility Features Packages successfully allowed mobility features to be computed several times a day.
- Non-uniform gaps were observed in the collected location data (as in similar 'in the wild' studies such as in [Pal+17]) which reduced the accuracy of the computed features. These gaps are likely due to the operating system on the smart-phones throttling background tracking.
- When comparing the daily location features with subjective user data we found that the Mobility Features Package predicts the Number of Places visited with an RMSE of 0.5 places, the Home Stay percentage with an RMSE of 14.3% and the Routine Index with an RMSE of 22.5%.
- The algorithms tends to undershoot in predicting the number of places and home stay, likely due to gaps in the data. There is no consistent over- or undershooting done when it comes to computing the Routine Index, which likely stemmed from gaps in the location data, as well as variance the validity of the subjective answers regarding the participants routine.

Different approaches to mitigate the errors in computing features, i.e. how the feature algorithms can be improved as well as the procedure which saves and loads historical data. In addition it is also discussed how an imputation method using historical data is likely need to cover gaps in the location data which were observed in the data analysis.

APPENDIX A
Nomenclature

API Application Programming Interface

BA Behavioural Activation

CACHET Copenhagen Center for Health Technology

CAMS CARP Mobile Sensing

CARP CACHET Research Platform

MDD Major Depressive Disorder

DBSCAN Density-based spatial clustering of applications with noise

APPENDIX B
Questionnaires

B.1 PHQ-9 (Patient Health Questionnaire)

The PHQ-9 questionnaire, patented by Pfizer, contains 9 questions pertaining to the mental state of the patient [1]. Each question asks 'Over the last two weeks, how often have you been bothered by any of the following problems?' with the questions being the following:

Q1: Little interest or pleasure in doing things?

Q2: Feeling down, depressed, or hopeless?

Q3: Trouble falling or staying asleep, or sleeping too much?

Q4: Feeling tired or having little energy?

Q5: Poor appetite or overeating?

Q6: Feeling bad about yourself, or that you are a failure, or have let yourself or your family down?

Q7: Trouble concentrating on things, such as reading the newspaper or watching television?

Q8: Moving or speaking so slowly that other people could have noticed. Or the opposite – being so fidgety or restless that you have been moving around a lot more than usual?

Q9: Thoughts that you would be better off dead, or of hurting yourself in some way?

Each question can be answered with the following 4 possibilities, each giving a number of points indicated in brackets:

- Not at all (0 points)
- Several days (1 point)

[1] `https://patient.info/doctor/patient-health-questionnaire-phq-9`

- More than half the days (2 points)
- Nearly every day (3 points)

At the end of the survey, the points are summed up and the patient is categorized into one of 5 categories based on the number of points acquired:

- Less than 5 (no depression)
- 5-9 (mild depression)
- 10-14 (moderate depression)
- 15-10 (moderate/severe depression)
- Greater than 20 (severe depression)

APPENDIX C
Package Documentation

C.1 Structure

The package contains two main directories and three metadata files as depicted in Figure C.1. The first directory is the source code directory, *lib*, containing the domain model, and algorithms for computing MobilityContexts. The second directory is the *test* directory containing unit tests which aid in the process of validating the algorithms. The metadata files are the *CHANGELOG.md* which contains a list of changes made to the package such that an application programmer can keep track of changes to the API.

The *pubspec.yaml* contains the package specification including the package name, a description, version, homepage, and a list of dependencies. The dependencies are other packages on which the package depends, as in this case, the Mobility Features Package depends on the `simple_cluster`, `stats` and `path_provider` packages each with a specific version number. The package itself also has such a version number that allows an application developer to import a specific version of the package, for

```
mobility_features
   lib/
       mobility_context.dart
       mobility_domain.dart
       mobility_features.dart
       mobility_functions.dart
       mobility_intermediate.dart
       mobility_serializer.dart
   test/
       data/
       mobility_features_test.dart
       test_utils.dart
   CHANGELOG.md
   pubspec.yaml
   README.md
```

Figure C.1: The file structure of the Mobility Features Flutter Package.

example, if they built their application around a previous release, they may wish to continue depending on that specific release rather than upgrading to the newest version.
Lastly, the README.md file contains instructions for using the package including code snippets and use case examples.

C.2 Publishing

Distributing a Flutter package is done via the Dart Package Manager, Pub. Pub is essentially a git repository of a package including all versions of that package. When publishing a package the contents of the README file are converted to HTML and are what the user is initially presented with. The README should, therefore, give a brief overview and description of the package, in addition to instructions. Figure C.3 shows the latest version of the package hosted at `https://pub.dev/packages/mobility_features`.
Publishing automatically generate API documentation by using comments in the code. Normally, comments are made with 2 forward slashes (//), but comments made with three forward slashes (///) mark the code-block following it with API documentation, i.e. the contents of the comment.

```
name: mobility_features
description: Real-time mobility feature calculation
version: 1.1.5
homepage: https://github.com/cph-cachet/flutter-plugins/

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  simple_cluster: ^0.2.0
  stats: ^0.2.0+3
  path_provider: ^1.6.10

dev_dependencies:
  flutter_test:
    sdk: flutter
```

Figure C.2: The pubspec.yaml file for the Mobility Features Package.

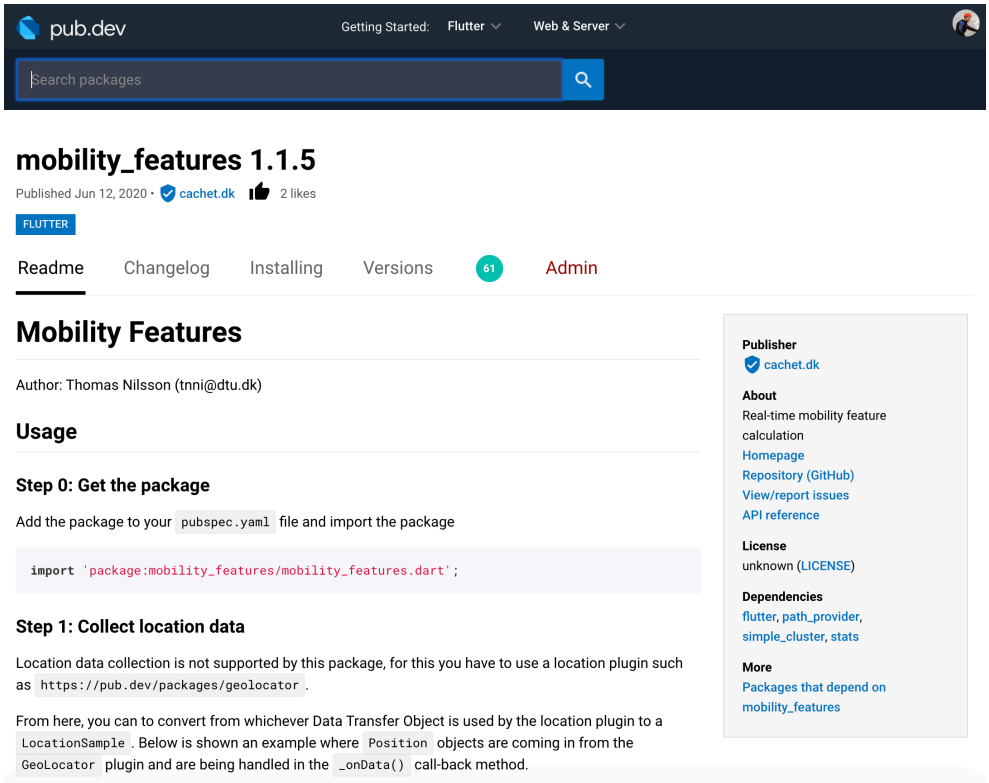


Figure C.3: The page hosting the Mobility Features Package on www.pub.dev.

```
/// A [LocationSample] holds a 2D [GeoPosition] spatial data point
/// as well as a [DateTime] value s.t. it may be temporally ordered
class LocationSample implements _Serializable, _Geospatial {...}
```

Figure C.4: The API comments for the source code of the Location Sample class.

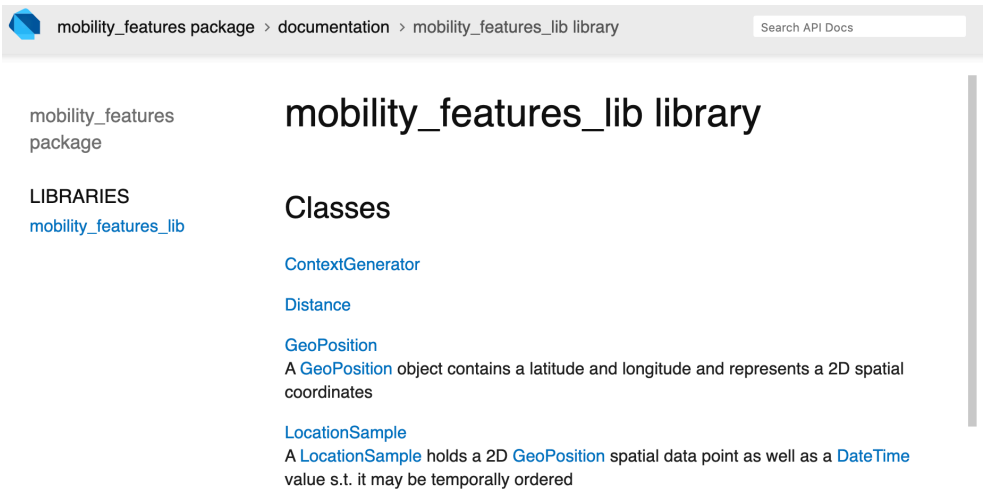


Figure C.5: The auto generated documentation for the package, hosted on pub.dev, including the code snippet in Figure C.4 for the Location Sample class.

APPENDIX D

Python Demo

The following pages contain the Python implementation of the offline feature algorithms which is run on a synthetic dataset. This was used in the development process since prototyping in Python is much faster than in Dart.

FEATURES-DEMO

June 22, 2020

1 Stops, places and moves location analysis

Definitions: - **Location data** is collected as a sequence of location samples with varying sample frequency and accuracy. - **Places** are locations of relevance to the user, such as home or workplace and are described by their coordinates and an ID. - **Stops** are specific visits to one of those places, described by their coordinates along with arrival and departure time. A stop is always associated with exactly one place while a place can be associated with many stops. Stops are always non-overlapping in time. - **Moves** are sequences of location points between stops and are described by departure and arrival time, origin and destination place and the distance of the move.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from geopy.distance import geodesic
        from sklearn.cluster import DBSCAN
        import gmaps

        import sys
        sys.path.append("../")
        from location import *

        # Keep data consistent, load from disk.
        LOAD_DATA_FROM_DISK = False
```

```
In [2]: def get_date(row):
            return row.date()
```

1.1 Generate example data

```
In [3]: if LOAD_DATA_FROM_DISK:
            df = pd.read_json('multi_date_data.json').T
            df.datetime = pd.to_datetime(df.datetime, unit='ms')
            df.date = df.datetime.dt.date.astype('datetime64[ns]')
        else:
            # create data simulating 3 places (a,b,c)
            a = (55.686381, 12.557155) # Blaagaards Plads
            b = (55.666919, 12.536792) # Spaces
            c = (55.688305, 12.561862) # Hulen
```

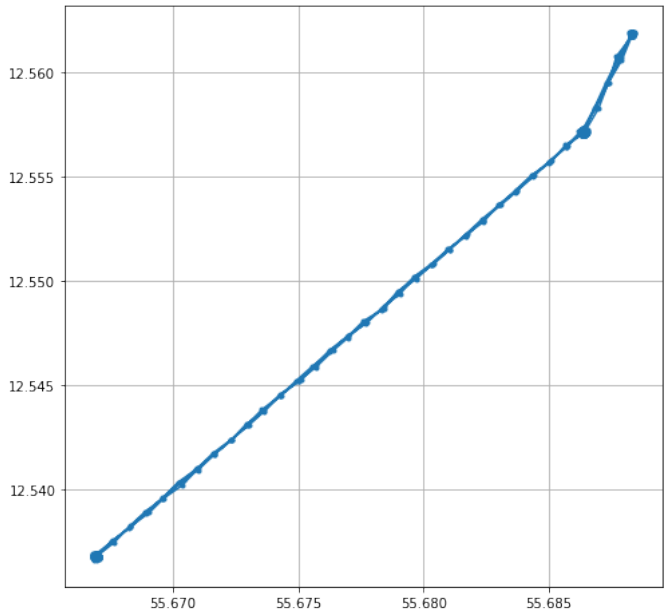
```
            X = np.vstack([
                # day 1: home, work, home, workout, home
                np.array([a]*(60*8+30)),
                np.array([np.linspace(a[0], b[0], 30), np.linspace(a[1], b[1], 30)]).T,
                np.array([b]*(60*7+30)),
                np.array([np.linspace(b[0], a[0], 30), np.linspace(b[1], a[1], 30)]).T,
                np.array([a]*55),
                np.array([np.linspace(a[0], c[0], 5), np.linspace(a[1], c[1], 5)]).T,
                np.array([c]*55),
                np.array([np.linspace(c[0], a[0], 5), np.linspace(c[1], a[1], 5)]).T,
                np.array([a]*60*5),
                # day 2: home, work, home
                np.array([a]*(60*8+30)),
                np.array([np.linspace(a[0], b[0], 30), np.linspace(a[1], b[1], 30)]).T,
                np.array([b]*(60*7+30)),
                np.array([np.linspace(b[0], a[0], 30), np.linspace(b[1], a[1], 30)]).T,
                np.array([a]*60*7),
                # day 3: home, workout, home
                np.array([a]*(60*10+55)),
                np.array([np.linspace(a[0], c[0], 5), np.linspace(a[1], c[1], 5)]).T,
                np.array([c]*55),
                np.array([np.linspace(c[0], a[0], 5), np.linspace(c[1], a[1], 5)]).T,
                np.array([a]*60*12),
                ])

            X += np.random.normal(loc=0, scale=0.00005, size=X.shape)

            df = pd.DataFrame(X, columns=['latitude', 'longitude'])
            df.insert(0, 'user_id', 0)
            df.insert(1, 'timestamp', np.arange(df.shape[0]) * 60000 + 1573430400000.0)
            df.insert(2, 'datetime', pd.to_datetime(df.timestamp, unit='ms'))
            df.insert(3, 'date', df.datetime.dt.date.astype('datetime64[ns]'))

            # Write to file
            df.T.to_json('multi_date_data.json')

        df.head()

        df.date = df.date.apply(get_date)
```

```
In [57]: dates = np.unique(df.date.values)
         dates
```

```
Out[57]: array([datetime.date(2019, 11, 11), datetime.date(2019, 11, 12),
                datetime.date(2019, 11, 13)], dtype=object)
```

1.2 Quick visualization of the 3 places visited

```
In [58]: plt.figure(figsize=(8,8))
         plt.plot(df.latitude.values, df.longitude.values, marker='.', alpha=1)
         plt.grid()
         plt.show()
```



1.3 Preprocessing (stops, places and moves)

- A `stop` is a collection of stationary points
- A `place` is a cluster of `stops` found using DBSCAN
- A `move` is a transition from one `stop` to another.

```
In [6]: stops, places, moves = get_stops_places_and_moves(df)
        stops['date'] = stops.arrival.apply(get_date)
        moves['date'] = moves.arrival.apply(get_date)
```

```
In [7]: stops
```

Out[7]:

	user_id	latitude	longitude	samples	arrival	departure	duration	place	date
0	0	55.686381	12.557161	511	2019-11-11 00:00:00	2019-11-11 08:30:00	510.0	0	2019-11-11
1	0	55.666914	12.536788	452	2019-11-11 08:59:00	2019-11-11 16:30:00	451.0	1	2019-11-11
2	0	55.686383	12.557170	57	2019-11-11 16:59:00	2019-11-11 17:55:00	56.0	0	2019-11-11
3	0	55.688303	12.561877	57	2019-11-11 17:59:00	2019-11-11 18:55:00	56.0	2	2019-11-11
4	0	55.686376	12.557154	812	2019-11-11 18:59:00	2019-11-12 08:30:00	811.0	0	2019-11-11
5	0	55.666921	12.536791	452	2019-11-12 08:59:00	2019-11-12 16:30:00	451.0	1	2019-11-12
6	0	55.686382	12.557154	1077	2019-11-12 16:59:00	2019-11-13 10:55:00	1076.0	0	2019-11-12
7	0	55.688310	12.561865	57	2019-11-13 10:59:00	2019-11-13 11:55:00	56.0	2	2019-11-13
8	0	55.686377	12.557154	721	2019-11-13 11:59:00	2019-11-13 23:59:00	720.0	0	2019-11-13

```
In [8]: places
```

Out[8]:

	user_id	place	latitude	longitude	duration	stops
0	0	0	55.686381	12.557154	3173.0	5
1	0	1	55.666918	12.536790	902.0	2
2	0	2	55.688307	12.561871	112.0	2

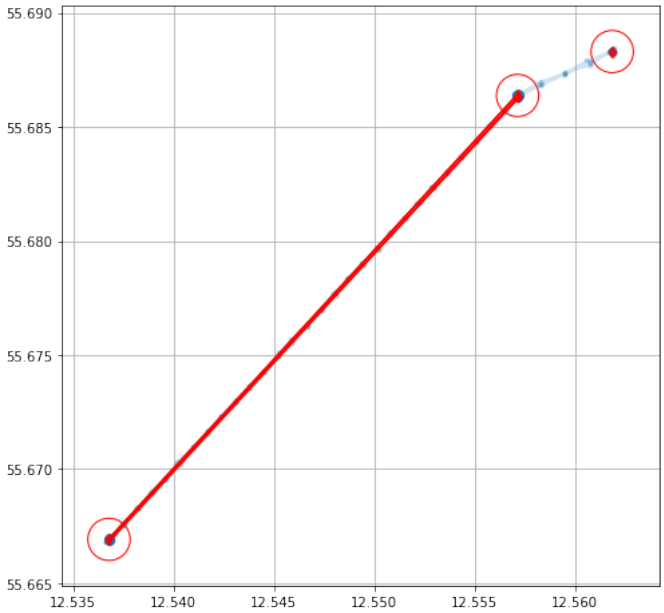
```
In [9]: moves
```

Out[9]:

	user_id	from_latitude	from_longitude	to_latitude	to_longitude	samples	departure	arrival	from_place	to_place	distance	duration	mean_speed	date
0	0	55.686350	12.557288	55.666931	12.536812	30	2019-11-11 08:30:00	2019-11-11 08:59:00	0.0	1.0	2520.430627	29.0	1.448523	2019-11-11
1	0	55.666994	12.536835	55.686413	12.557147	30	2019-11-11 16:30:00	2019-11-11 16:59:00	1.0	0.0	2516.562350	29.0	1.446300	2019-11-11
2	0	55.686326	12.557210	55.666843	12.536827	30	2019-11-12 08:30:00	2019-11-12 08:59:00	0.0	1.0	2523.909200	29.0	1.450523	2019-11-12
3	0	55.666941	12.536796	55.686385	12.557099	30	2019-11-12 16:30:00	2019-11-12 16:59:00	1.0	0.0	2518.612156	29.0	1.447478	2019-11-12

1.3.1 Visualizing the clusters and moves

```
In [59]: plt.figure(figsize=(8,8))
         plt.plot(df.longitude.values, df.latitude.values, marker='.', alpha=.2)
         plt.scatter(stops.longitude.values, stops.latitude.values, marker='d', color='r', zor
         plt.scatter(places.longitude.values, places.latitude.values, s=1000, facecolors='none
         for index, move in moves.iterrows():
             plt.plot([move.from_longitude, move.to_longitude], [move.from_latitude, move.to_la
         plt.grid()
         plt.show()
```



2 Features

2.1 Number of clusters

This feature represents the total number of clusters found by the clustering algorithm.

```
In [60]: def number_of_clusters(places):
             return len(places)

In [61]: number_of_clusters(places)

Out[61]: 3
```

2.2 Location Variance:

This feature measures the variability of a participant's location data from stationary states. LV was computed as the natural logarithm of the sum of the statistical variances of the latitude and the longitude components of the location data.

```
In [62]: def location_variance(df):
             # If fewer than 2 observations, we can't compute the variance
             if len(df) < 2:
                 return 0.0
             return np.log(df.latitude.var() + df.longitude.var() + 1)

In [63]: location_variance(df)

Out[63]: 0.00013597465949774686
```

2.3 Location Entropy (LE):

A measure of points of interest. High entropy indicates that the participant spent time more uniformly across different location clusters, while lower entropy indicates the participant spent most of the time at some specific clusters. Concretely it is calculated as:

$$Entropy = -\sum_{i=1}^{N} p_i \cdot \log p_i$$

where each i represents a location cluster, N denotes the total number of location clusters, and pi is the percentage of time the participant spent at the location cluster i. High cluster entropy indicates that the participant spent time more uniformly across different location clusters, while lower cluster entropy indicates the participant spent most of the time at some specific clusters.

Here, we use the duration spent at each place, found in the `duration` column in the `places` dataframe.

```
In [64]: def _entropy(durations):
             p = durations / np.sum(durations)
             return -np.sum(p * np.log(p))

In [65]: _entropy(places.duration)

Out[65]: 0.6377255748619863

In [66]: # NumPy for reference:

         from scipy.stats import entropy
         entropy(places.duration)

Out[66]: 0.6377255748619863
```

2.4 Normalized LE:

Normalized entropy is calculated by dividing the cluster entropy by its maximum value, which is the logarithm of the total number of clusters. Normalized entropy is invariant to the number of clusters and thus solely depends on their visiting distribution. The value of normalized entropy ranges from 0 to 1, where 0 indicates the participant has spent their time at only one location, and 1 indicates that the participant has spent an equal amount of time to visit each location cluster.

Here we just divide by the log to the number of places.

```
In [67]: def normalized_entropy(durations):
             return entropy(durations) / np.log(len(durations))

In [68]: normalized_entropy(places.duration)

Out[68]: 0.5804828340625297
```

2.5 Transition Time:

Transition Time measures the percentage of time the participant has been in the transition state.

A few ways of doing this, but one is using the `moves` dataframe and simply summing the `duration` column, and dividing by 24 hours.

```
In [69]: def transition_time(moves):
             move_time = moves.duration.sum()
             return move_time / (24 * 60)

In [70]: transition_time(moves)

Out[70]: 0.08055555555555556
```

3 Total Distance:

This feature measures the total distance the participant has traveled in the transition state.

Here we simply sum the `distance` column in the `moves` dataframe.

```
In [71]: def total_distance(moves):
             return moves.distance.sum()

In [72]: total_distance(moves)

Out[72]: 10079.514332342535
```

3.1 Routine Index

```
In [73]: HOURS_IN_A_DAY = 24

         def print_hour_matrix(M):
             for i, row in enumerate(M):
                 line = "[{:0>2} - {:0>2}] ".format(i, i+1)
                 for e in row:
```

```
                line += '%0.2f ' % e
            print(line)

    def make_hour_matrix(stops, num_places):
        h = np.zeros((HOURS_IN_A_DAY, num_places))

        for index, row in stops.iterrows():
            pid = row.place
            start_hour = row.arrival.hour
            end_hour   = row.departure.hour

            # If user arrived and departed within the same hour
            # Then the time stayed is the diff between departure and arrival
            if start_hour == end_hour:
                h[start_hour, pid] = row.departure.minute - row.arrival.minute

            else:
                # Arrival hour
                h[start_hour, pid] = 60 - row.arrival.minute

                # In between
                for hour in range(start_hour+1, end_hour):
                    h[hour, pid] = 60

                # Departure hour
                h[end_hour, pid] = row.departure.minute

        return h / 60 # Normalize by 60 mins
```

In [74]:
```
# Plot a matrix as a color map
def matrix_plot(m):
    plt.figure(figsize=(10,10))
    plt.imshow(m, cmap='bone')
    plt.title('Hour matrix')
    plt.xlabel('Place ID')
    plt.ylabel('Timeslot')
    plt.yticks(range(HOURS_IN_A_DAY), ["[{:0>2} - {:0>2}] ".format(i, i+1) for i in r
    plt.xticks(range(m.shape[1]))
    plt.show()
```

In [75]:
```
s1 = stops[stops.date == dates[0]]
s1
```

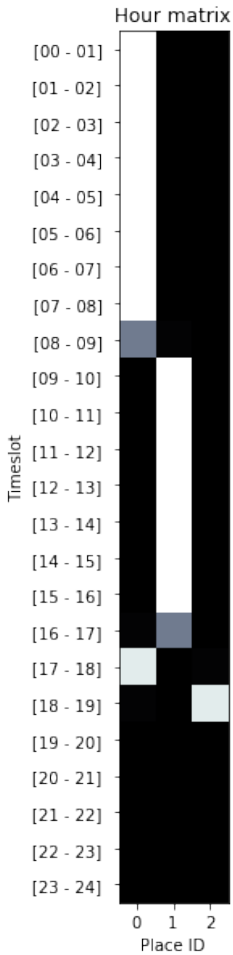
Out[75]:

	user_id	latitude	longitude	samples	arrival	\
0	0	55.686381	12.557161	511	2019-11-11 00:00:00	
1	0	55.666914	12.536788	452	2019-11-11 08:59:00	
2	0	55.686383	12.557170	57	2019-11-11 16:59:00	
3	0	55.688303	12.561877	57	2019-11-11 17:59:00	

```
4        0  55.686376  12.557154      812 2019-11-11 18:59:00
```

	departure	duration	place	date
0	2019-11-11 08:30:00	510.0	0	2019-11-11
1	2019-11-11 16:30:00	451.0	1	2019-11-11
2	2019-11-11 17:55:00	56.0	0	2019-11-11
3	2019-11-11 18:55:00	56.0	2	2019-11-11
4	2019-11-12 08:30:00	811.0	0	2019-11-11

```
In [76]: h1 = make_hour_matrix(s1, len(places))
         matrix_plot(h1)
```


Hour matrix
[00 - 01]
[01 - 02]
[02 - 03]
[03 - 04]
[04 - 05]
[05 - 06]
[06 - 07]
[07 - 08]
[08 - 09]
[09 - 10]
[10 - 11]
[11 - 12]
[12 - 13]
[13 - 14]
[14 - 15]
[15 - 16]
[16 - 17]
[17 - 18]
[18 - 19]
[19 - 20]
[20 - 21]
[21 - 22]
[22 - 23]
[23 - 24]
Timeslot
0
1
2
Place ID

```
In [77]: def RI(h_mean, h, end_hour=24):
             '''
             input:
                 h_mean (2d matrix): Historical Mean Matrix
                 h (2d matrix): Hour Matrix for a day

             output:
                 routine_index: -1 (could not be calculated) or [0 to 1].
             '''
             if h_mean.sum() == 0:
                 return -1.0 # no routine index could be calculated

             assert(h_mean.shape == h.shape)

             m,n = h.shape

             overlap = 0.0
             for i in range(m):
                 for j in range(n):
                     overlap += min(h_mean[i,j], h[i,j])

             max_overlap = min(h_mean.sum(), h.sum())

             return overlap / max_overlap
```

3.2 Using todays stops and historical stops to calculate routine index

I.e. no updating of routine matrix, always recalculate it.

```
In [78]: STOPS = {}
         for date in dates:
             print('Date:', date)
             # Select data by date
             data = df[df.date == date]

             # Find stops, moves, places
             S, P, M = get_stops_places_and_moves_daily(data, merge=False, move_duration=3)

             # Store them
             STOPS[date] = S
```

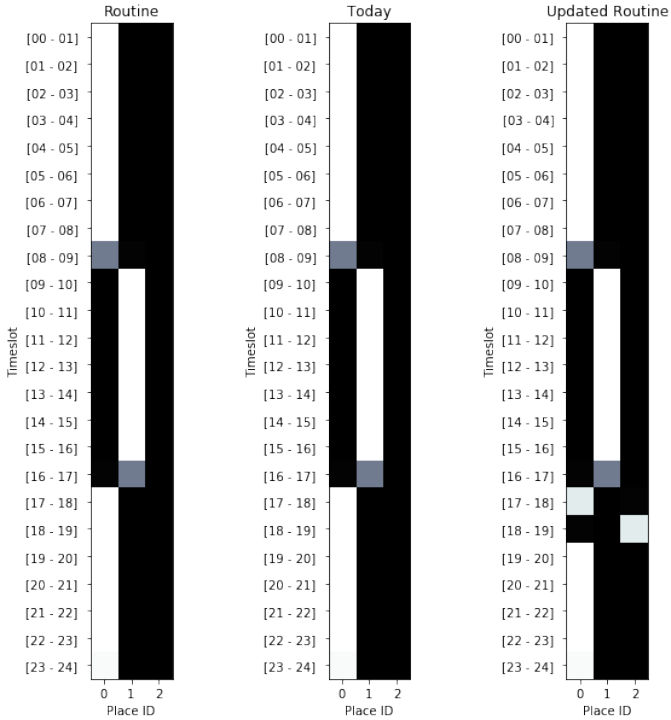
```
Date: 2019-11-11
Date: 2019-11-12
Date: 2019-11-13
```

```
In [79]: def plot_today_and_routine(today, routine, routine_after, save=False):
             interval_strings = ["[{:0>2} - {:0>2}] ".format(i, i+1) for i in range(HOURS_IN_A
```

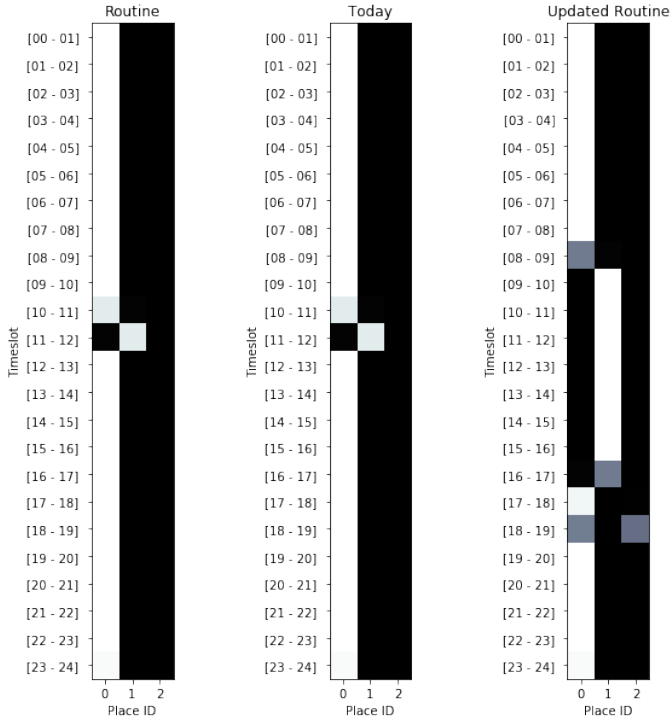
```
    f, (ax1, ax2, ax3) = plt.subplots(1, 3)
    f.set_size_inches((10,10))


    ax1.imshow(routine, cmap='bone')
    ax1.set_title('Routine')
    ax1.set_xlabel('Place ID')
    ax1.set_ylabel('Timeslot')
    ax1.set_yticks(range(HOURS_IN_A_DAY))
    ax1.set_yticklabels(interval_strings)
    ax1.set_xticks(range(routine.shape[1]))

    ax2.imshow(today, cmap='bone')
    ax2.set_title('Today')
    ax2.set_xlabel('Place ID')
    ax2.set_ylabel('Timeslot')
    ax2.set_yticks(range(HOURS_IN_A_DAY))
    ax2.set_yticklabels(interval_strings)
    ax2.set_xticks(range(today.shape[1]))

    ax3.imshow(routine_after, cmap='bone')
    ax3.set_title('Updated Routine')
    ax3.set_xlabel('Place ID')
    ax3.set_ylabel('Timeslot')
    ax3.set_yticks(range(HOURS_IN_A_DAY))
    ax3.set_yticklabels(interval_strings)
    ax3.set_xticks(range(routine_after.shape[1]))

    if save:
        plt.savefig('routine.png')
    plt.show()
```

```
In [80]: DISTF = lambda a, b: geodesic(a, b).meters

         def get_places_2(stops, dist=25, distf=DISTF):
             if stops.empty:
                 stops['place'] = []
                 places = pd.DataFrame(columns=['user_id', 'place', 'latitude', 'longitude', '
             else:
                 points = stops[['latitude', 'longitude']].values
                 dbs = DBSCAN(dist, min_samples=1, metric=distf).fit(points)
                 stops['place'] = dbs.labels_
                 places = stops.groupby('place').agg({
                     'latitude': np.median,
                     'longitude': np.median,
                     'duration': np.sum,
                     'samples': len,
                 }).reset_index()
```

```
                places.rename(columns={'samples': 'stops'}, inplace=True)
                places.insert(0, 'user_id', stops.user_id.values[0])
            return stops, places

In [96]: for date in dates:
            # Calculate todays matrix
            stops_today = STOPS[date]
            stops_so_far = [STOPS[d] for d in dates[dates <= date]]
            stops_so_far = pd.concat(stops_so_far)
            stops_so_far = stops_so_far.sort_values(['arrival'])
            stops_so_far = stops_so_far.reset_index()
            stops_so_far, places_so_far = get_places_2(stops_so_far)

            number_of_places = len(places_so_far)

            hour_matrix_today = make_hour_matrix(stops_today, number_of_places)

            dates_hist = dates[dates < date]

            routine_matrix = hour_matrix_today

            if len(dates_hist) > 0:
                print(date)
                hour_matrices_hist = [make_hour_matrix(STOPS[date_hist], number_of_places) fo
                new_routine_matrix = np.mean(hour_matrices_hist, axis=0)
                ri = RI(new_routine_matrix, hour_matrix_today)
                plot_today_and_routine(hour_matrix_today, routine_matrix, new_routine_matrix)
                routine_matrix = new_routine_matrix

            else:
                ri = -1
            print('Routine index: %0.2f' % ri)

            hms[date] = hour_matrix_today
            rms[date] = routine_matrix

            print('-'*35)

Routine index: -1.00
-----------------------------------
2019-11-12
```




```
Routine index: 0.96
-----------------------------------
2019-11-13
```




```
Routine index: 0.69
-----------------------------------
```

3.3 Home Stay:

The percentage of time the participant has been at the cluster that represents home. We define the home cluster as the cluster, which is mostly visited during the period between 12 am and 6 am.

Implementation steps: * Identify home: Use the `hours` dataframe to determine the most visited cluster between 00 and 06 am. * Count percentage of time at home: Use the `places` dataframe to calculate the time distribution.

However - we need to fill out the `hours` dataframe with data between 00 and 06 first

```
In [33]: date = dates[0]
         # Calculate todays matrix
         stops_today = STOPS[date]
         stops_so_far = [STOPS[d] for d in dates[dates <= date]]
         stops_so_far = pd.concat(stops_so_far)
         stops_so_far = stops_so_far.sort_values(['arrival'])
         stops_so_far = stops_so_far.reset_index()
         stops_so_far, places_so_far = get_places_2(stops_so_far)

         num_places = len(places_so_far)

In [34]: H = make_hour_matrix(stops_today, num_places)

In [104]: def get_home_place(hour_matrix):
              start, end = 0, 6
              place_dist = hour_matrix[start:end].sum()

              # Check that there was actually data between 00 and 06
              assert not np.all(hour_matrix[start:end].sum() == 0)

              return hour_matrix[start:end].sum().argmax()

          def home_stay(places, hour_matrix):
              distr = places.duration / places.duration.sum()
              home_id = get_home_place(hour_matrix)
              return distr[home_id]

In [106]: get_home_place(H)

Out[106]: 0

In [109]: home_stay(places_so_far, H)

Out[109]: 0.6307356154406409
```

APPENDIX E

Source Code

E.0.1 Storing and Loading Data

The storing and loading of data, which includes Location Samples, Stops, and Moves happen through the *MobilitySerializer* class. This class allows classes that implement the Serializable interface to be serialized and de-serialized. Just like the GeoSpatial interface, the Serializable interface is also implemented as a private abstract class only used internally in the package library. The interface contains a method for serializing a class to JSON, named `toJson()` which takes no parameters and produces a HashMap of Strings to the dynamic, the dynamic type meaning any type. This is the Dart equivalent of a JSON object. Another method the interface forces other classes to implement is the deserialization method `fromJson(json)` which takes a JSON object as a parameter and creates a runtime object of the given type, from the JSON object. The implementation of this method is left to the individual classes implementing the interface which is done by extracting data from the JSON object.

```
abstract class _Serializable {
  Map<String, dynamic> _toJson();

  _Serializable._fromJson(Map<String, dynamic> json);
}
```

The MobilitySerializer class is a generic which allows the type `E` to be specified later, with `E` referring to either a Location Sample, Stop or Move which all implement the Serializable interface. The MobilitySerializer is constructed using a reference to a File object. The File object is used for storing the data of the given type i.e. Location Samples are stored one file, Stops in another, and Moves in a third.

```
class MobilitySerializer<E> {
  File file;

  MobilitySerializer._(this.file) {
    bool exists = file.existsSync();
    if (!exists) {
      flush();
    }
  }
```

```
  Future<void> flush() async =>
      await file.writeAsString('', mode: FileMode.write);
...
}
```

When initialized, it is checked whether or not the specified file exists, and if not the `flush` method is called, which simply writes an empty string to the file, overriding any content, which has the effect of creating the file, should it not already exist. A concrete example of instantiated the MobilitySerializer for Stops is shown below, where *stops.json* refers to the file in which Stops should be stored.

```
MobilitySerializer<Stop> stopSerializer =
        MobilitySerializer<Stop>._(await _file('stops.json'));
```

For storing data the `save` method is used which takes in a list of objects which all implement the Serializable interface. Each element in the list is serialized via its `toJson` method and concatenated into one big string separated by a delimiter token, and this string is then written to the specific file of the MobilitySerializer object.

```
Future<void> save(List<_Serializable> elements) async {
  String jsonString = "";
  for (_Serializable e in elements) {
    jsonString += json.encode(e._toJson()) + delimiter;
  }
  await file.writeAsString(jsonString,
    mode: FileMode.writeOnlyAppend);
}
```

Loading works in the reverse order, where the contents of the specified file are loaded into a string, the string is then split into elements using the delimiter token and each of these elements is de-serialized using the `fromJson` method. For deciding which type to de-serialize the elements into, a switch statement is used that checks the type of `E` which is specified when the MobilitySerializer object is instantiated.

```
Future<List<_Serializable>> load() async {
    String content = await file.readAsString();

    List<String> lines = content.split(delimiter);

    Iterable<Map<String, dynamic>> jsonObjs = lines
        .sublist(0, lines.length - 1)
        .map((e) => json.decode(e))
        .map((e) => Map<String, dynamic>.from(e));

    switch (E) {
```

```
          case Move:
            return jsonObjs.map((x) =>
                Move._fromJson(x)).toList();

          case Stop:
            return jsonObjs.map((x) =>
                Stop._fromJson(x)).toList();

          default:
            return jsonObjs.map((x) =>
                LocationSample._fromJson(x)).toList();
        }
    }
```

Ideally, the switch statement could have been replaced by the following one-liner:

```
    return jsonObjs.map((x) => E.fromJson(x)).toList();
```

However, this relies on the language feature called reflection [1] which allows the compiler to infer the type of `E` at compile-time. However, Dart does not support *reflection* which makes this impossible.

E.0.1.1 Accessing the File System

For storing collected location data the MobilitySerializer for Location Samples can be retrieved through this class, with a getter method.);

```
static Future<MobilitySerializer<LocationSample>>
      get locationSampleSerializer async =>
          MobilitySerializer<LocationSample>._(await _file(LOCATION_SAMPLES)
```

Internally this class has a method for creating a file system reference, which relies on the platform the application is running on. For mobile apps, the file system must be accessed through the `path_provider` package with the `getApplicationDocumentsDirectory()` method. If the application is run on the desktop, such as when unit testing the file system can be accessed by specifying a file name directly. This is a textbook example of hiding complexity from the application programmer.

[1] `https://www.javaworld.com/article/2075801/reflection-vs--code-generation.html`

```
static Future<File> _file(String type) async {
  bool isMobile = Platform.isAndroid || Platform.isIOS;

  String path;
  if (isMobile) {
    path = (await getApplicationDocumentsDirectory()).path;
  } else {
    path = 'test/data';
  }
  return new File('$path/$type.json');
}
```

Figure E.1: Lazy evaluation of a feature.

APPENDIX F

Unit Testing

Unit testing [1], in which small parts of the source code are tested played a significant role in the latter part of the development process. It was prioritized to make a working demo application in order to conduct a study and while unit tests can speed up certain parts of the debugging process, it was still faster to 'hack something together'. The only testing done prior to the study was regarding serialization and making sure the feature computation producing meaningful results, i.e. they were manually verified. The traditional way of using unit testing is through Test-Driven Development developed by Kent Beck [Bec02] in which the tests are written first, and the corresponding source code which should pass the test is written afterward. The package went through many smaller iterations, in which the data flow was moved around, and unit testing made discovering bugs much easier by enabling one to constantly verify that the source code produced the desired results every time changes were made. As the package went through multiple iterations, each iteration either added or removed functionality or changed the existing functionality slightly which meant new unit tests were often written to cover the functionality. In the end, some the functionality was pruned and therefore some of the tests were also superfluous or had a large amount of overlap between them and were therefore also consolidated.

F.0.1 Limitations of Unit Testing

There were a few shortcomings of unit testing encountered, which largely came down to the inability to compare objects before and after serialization. This stems from objects having a hash code, i.e. a unique fingerprint which is not stored when serializing. The fingerprint is used to compare objects while in memory, and since the fingerprint is lost the problem arises. For this to be resolved, a better testing method needs to be implemented in terms of comparing objects. This can be done by implementing a function that breaks down each object, be it a Stop, or MobilityContext, into the most atomic values, i.e. latitude, longitude, time-stamp, etc, which can be compared without a hash code. For testing the algorithms, small synthetic datasets were created in order to test very rudimentary cases. It was harder to construct very large synthetic datasets in order to test more realistic, noisy datasets and discover edge cases. In the future, more elaborate unit tests should be written, especially for the clustering algorithms, since the ground truth, i.e. cluster centroids and points belonging to clusters can be calculated by hand. Another possibility that was partly

[1] `https://martinfowler.com/bliki/UnitTest.html`

explored was using a real-world dataset that the author gathered tracking himself however to verify the algorithms on this dataset it would need to manually label which was not done. The large real-world dataset was however used to verify that the algorithms produced meaningful results and that the computation did not throw any errors. Lastly, the current state of the API gives the public access to certain methods which are not supposed to have public access. The reason for this is that they need to be part of the unit tests, concretely it is the MobilitySerializer methods *flush* and *save*. These methods are not intended to be used by the user since the flush method deletes all contents of the corresponding file. The load method does not pose a threat to the usability, but is unnecessary clutter, and should only be used internally by the package.

F.0.2 Example Unit Tests

In this subsection selected unit tests will be exemplified.

Location Sample Serialization

This test is the simplest unit test in the collection in which the storing- and loading functionality of the MobilitySerializer is displayed. A small, synthetic is created consisting of three Location Samples, which is first stored via the save() method, and next the load() method is called. To check whether or not the store and load were successful, the lengths of the original dataset and the loaded dataset are compared.

```
test('Serialize and load three location samples', () async {
  MobilitySerializer<LocationSample> serializer =
      await ContextGenerator.locationSampleSerializer;

  LocationSample x =
      LocationSample(GeoPosition(123.456, 123.456), DateTime(2020, 01, 01));


  List<LocationSample> dataset = [x, x, x];

  await serializer.flush();
  await serializer.save(dataset);
  List loaded = await serializer.load();
  expect(loaded.length, dataset.length);
});
```

Figure F.1: A unit testing demonstrating storing and loading a small, synthetic dataset.

Test: Single Stop

This test is a step up in complexity in terms of what is tested. A dataset is constructed that simulates a user staying at a single location from 00:00 to 17:00. This should result in a single stop and place being found, no moves, and a homestay value of 1.0 (i.e. 100 percent). The data is first serialized and a Mobility Context is computed afterward from which the features are extracted.

Test: Multiple Days with Routine Index

This test works similarly to the previous one but has the dataset spread over two different locations. The same dataset is repeated for 5 days, where the number of Stops, Moves, and Places is evaluated each day, in addition to the Home Stay and Routine Index feature. Concretely, the places visited are the same each day, at the same hours of the day meaning the Routine Index is 1.0 except for the first day since the Routine Index requires at least one historical day for comparison. The user stays at one place from 00:00 to 06:00 making it the home cluster, and another place from 08:00 to 09:00. This means the Home Stay should be equal to $\frac{6}{9}$, or 66.67 percent.

```
test('Features: Single Stop', () async {
  Duration timeTracked = Duration(hours: 17);

  List<LocationSample> dataset = [
    // home from 00 to 17
    LocationSample(loc0, jan01),
    LocationSample(loc0, jan01.add(timeTracked)),
  ];

  MobilitySerializer<LocationSample> serializer =
      await ContextGenerator.locationSampleSerializer;

  serializer.flush();
  await serializer.save(dataset);

  MobilityContext context =
    await ContextGenerator.generate(today: jan01);
  expect(context.homeStay, 1.0);
  expect(context.stops.length, 1);
  expect(context.moves.length, 0);
  expect(context.places.length, 1);
});
```

Figure F.2: Unit test for a single Stop.

```
test('Features: Multiple days, multiple locations', () async {
  MobilitySerializer<LocationSample> serializer =
      await ContextGenerator.locationSampleSerializer;

  /// Clean file every time test is run
  serializer.flush();

  for (int i = 0; i < 5; i++) {
    DateTime date = jan01.add(Duration(days: i));

    /// Todays data
    List<LocationSample> locationSamples = [
      // 5 hours spent at home
      LocationSample(loc0, date.add(Duration(hours: 0, minutes: 0))),
      LocationSample(loc0, date.add(Duration(hours: 6, minutes: 0))),

      LocationSample(loc1, date.add(Duration(hours: 8, minutes: 0))),
      LocationSample(loc1, date.add(Duration(hours: 9, minutes: 0))),
    ];

    await serializer.save(locationSamples);

    /// Calculate and save context
    MobilityContext context = await ContextGenerator.generate(
        usePriorContexts: true, today: date);

    double routineIndex = context.routineIndex;
    double homeStay = context.homeStay;

    expect(context.stops.length, 2);
    expect(context.places.length, 2);
    expect(context.moves.length, 1);

    expect(homeStay, 6 / 9);

    // The first day the routine index should be -1,
    // otherwise 1 since the days are exactly the same
    if (i == 0) {
      expect(routineIndex, -1);
    } else {
      expect(routineIndex, 1);
    }
  }
});
```

Figure F.3: Unit test for a single Stop.

APPENDIX G
Installation Manual

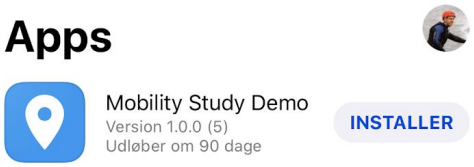
The following pages contains the instruction manual the participants were sent out, in order for them to install the application. The application was distributed via TestFlight where each participant received an invitation via their Apple ID.

Mobility Study - How to Install

Thomas Nilsson, Technical University of Denmark
tnni@dtu.dk

Step 1
Download Apple's TestFlight app from the App store.

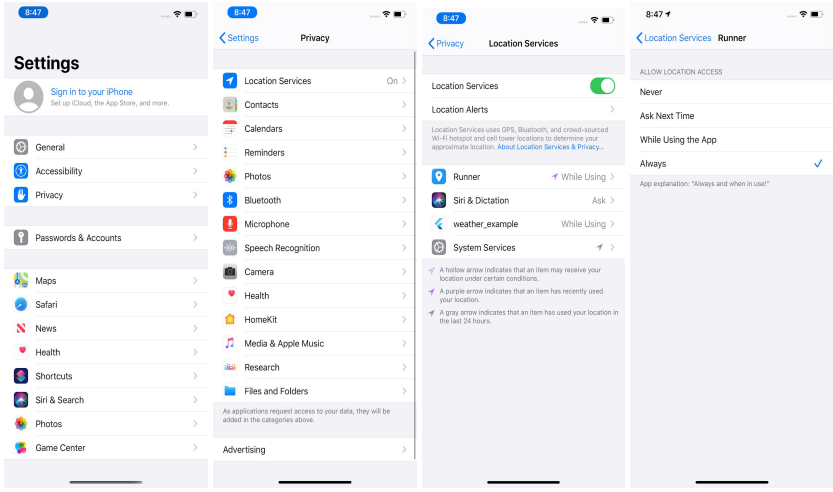
Step 2
Open the TestFlight app, and install the available app. This will install an app called *Runner*, on your device.



Step 3
Open the *Runner* app and give it the permissions it asks for.

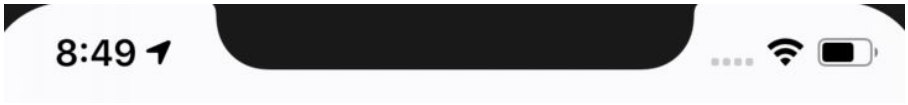
In addition, you need to minimize the app by pressing the home button and go into your settings app.

Here, navigate to *Privacy* > *Location Services* > *Runner* and choose 'Always' in order to allow the app to monitor your location in the background.



Step 4

Keep the application running the background, you should see the compass indicator in the top bar of your phone, which indicates that location is being tracked.



Step 5

Once a day you will be asked to fill out four questions, we call this a diary. You can fill out the diary as many times as you want per day (for example if you fill it out wrongly). However, only the latest diary on a given day will be used.

Bibliography

[AS02a] D. Ashbrook and T. Starner. "Learning significant locations and predicting user movement with GPS". In: *Proceedings. Sixth International Symposium on Wearable Computers,* 2002, pages 101–108.

[AS02b] D. Ashbrook and T. Starner. "Learning significant locations and predicting user movement with GPS". In: *Proceedings. Sixth International Symposium on Wearable Computers,* 2002, pages 101–108.

[Bar20] Jakob E. Bardram. *The CARP Mobile Sensing Framework – A Cross-platform, Reactive, Programming Framework and Runtime Environment for Digital Phenotyping.* 2020. arXiv: 2006.11904 [cs.HC].

[Bec02] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature).* Addison-Wesley Longman, Amsterdam, 2002. ISBN: 0321146530.

[Bru13] Glen Robert van Brummelen. *eavenly Mathematics: The Forgotten Art of Spherical Trigonometry.* Princeton University Press, 2013. ISBN: ISBN 9780691148922.

[CLL14] Andrea Cuttone, Sune Lehmann, and Jakob Eg Larsen. "Inferring Human Mobility from Sparse Low Accuracy Mobile Sensing Data". In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication.* UbiComp '14 Adjunct. Seattle, Washington: Association for Computing Machinery, 2014, pages 995–1004. ISBN: 9781450330473. DOI: 10.1145/2638728.2641283. URL: https://doi.org/10.1145/2638728.2641283.

[CM15] Luca Canzian and Mirco Musolesi. "Trajectories of Depression: Unobtrusive Monitoring of Depressive States by Means of Smartphone Mobility Traces Analysis". In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing.* UbiComp '15. Osaka, Japan: Association for Computing Machinery, 2015, pages 1293–1304. ISBN: 9781450335744. DOI: 10.1145/2750858.2805845. URL: https://doi.org/10.1145/2750858.2805845.

[CP13] Isabelle [Soucy Chartier] and Martin D. Provencher. "Behavioural activation for depression: Efficacy, effectiveness and dissemination". In: *Journal of Affective Disorders* 145.3 (2013), pages 292–299. ISSN: 0165-0327. DOI: https://doi.org/10.1016/j.jad.2012.07.023. URL: http://www.sciencedirect.com/science/article/pii/S0165032712005423.

[Dep20] Office of the Department of Defense. "GLOBAL POSITIONING SYSTEM STANDARD POSITIONING SERVICE PERFORMANCE STANDARD". In: 5th Edition (2020). URL: https://www.gps.gov/technical/ps/.

[Dim+06] Sona Dimidjian et al. "Randomized trial of behavioral activation, cognitive therapy, and antidepressant medication in the acute treatment of adults with major depression." In: *Journal of Consulting and Clinical Psychology* 74.4 (2006), pages 658–670. DOI: 10.1037/0022-006X.74.4.658. URL: https://doi.org/10.1037/0022-006X.74.4.658.

[Dor+18] Afsaneh Doryab et al. "Extraction of Behavioral Features from Smartphone and Wearable Data". In: *CoRR* abs/1812.10394 (2018). arXiv: 1812.10394. URL: http://arxiv.org/abs/1812.10394.

[Ebe+17] David Daniel Ebert et al. "Prevention of Mental Health Disorders Using Internet- and Mobile-Based Interventions: A Narrative Review and Recommendations for Future Research". In: *Frontiers in Psychiatry* 8 (2017), page 116. ISSN: 1664-0640. DOI: 10.3389/fpsyt.2017.00116. URL: https://www.frontiersin.org/article/10.3389/fpsyt.2017.00116.

[Est+96] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters a Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pages 226–231.

[FKD15] Denzil Ferreira, Vassilis Kostakos, and Anind K. Dey. "AWARE: Mobile Context Instrumentation Framework". In: *Frontiers in ICT* 2 (2015), page 6. ISSN: 2297-198X. DOI: 10.3389/fict.2015.00006. URL: https://www.frontiersin.org/article/10.3389/fict.2015.00006.

[Fow+02] Martin Fowler et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[Gra+15] Franz Gravenhorst et al. "Mobile phones as medical devices in mental disorder treatment: an overview". In: *Personal and Ubiquitous Computing* 19.2 (February 2015), pages 335–353. ISSN: 1617-4917. DOI: 10.1007/s00779-014-0829-5. URL: https://doi.org/10.1007/s00779-014-0829-5.

[Ins18] Thomas R. Insel. "Digital phenotyping: a global tool for psychiatry". eng. In: *World psychiatry : official journal of the World Psychiatric Association (WPA)* 17.3 (October 2018). PMC6127813[pmcid], pages 276–277. ISSN: 1723-8617. DOI: 10.1002/wps.20550. URL: https://doi.org/10.1002/wps.20550.

[Mac03] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge: Cambridge University Press, 2003. ISBN: ISBN 0-521-64298-1.

[Moh+13] David C. Mohr et al. "Behavioral Intervention Technologies: Evidence review and recommendations for future research in mental health". In: *General Hospital Psychiatry* 35.4 (2013), pages 332–338. ISSN: 0163-8343. DOI: https://doi.org/10.1016/j.genhosppsych.2013.03.008. URL: http://www.sciencedirect.com/science/article/pii/S0163834313000698.

[Pal+17] N. Palmius et al. "Detecting Bipolar Depression From Geographic Location Data". In: *IEEE Transactions on Biomedical Engineering* 64.8 (2017), pages 1761–1771.

[Roh+18] Darius A Rohani et al. "Correlations Between Objective Behavioral Features Collected From Mobile and Wearable Devices and Depressive Mood Symptoms in Patients With Affective Disorders: Systematic Review". In: *JMIR Mhealth Uhealth* 6.8 (August 2018), e165. ISSN: 2291-5222. DOI: 10.2196/mhealth.9691. URL: http://www.ncbi.nlm.nih.gov/pubmed/30104184.

[Roh+20] Darius A. Rohani et al. "MUBS: A Personalized Recommender System for Behavioral Activation in Mental Health". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pages 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376879. URL: https://doi.org/10.1145/3313831.3376879.

[Sae+15a] Sohrab Saeb et al. "Mobile Phone Sensor Correlates of Depressive Symptom Severity in Daily-Life Behavior: An Exploratory Study". In: *Journal of Medical Internet Research* 17 (July 2015). DOI: 10.2196/jmir.4273.

[Sae+15b] Sohrab Saeb et al. "The Relationship between Clinical, Momentary, and Sensor-based Assessment of Depression". In: *International Conference on Pervasive Computing Technologies for Healthcare : [proceedings]. International Conference on Pervasive Computing Technologies for Healthcare* 2015 (August 2015). PMC4667797[pmcid], 10.4108/icst.pervasivehealth.2015.259034. ISSN: 2153-1633. DOI: 10.4108/icst.pervasivehealth.2015.259034. URL: https://www.ncbi.nlm.nih.gov/pubmed/26640739.

[Spa+08] Stefano Spaccapietra et al. "A conceptual view on trajectories". In: *Data Knowledge Engineering* 65.1 (2008). Including Special Section: Privacy Aspects of Data Mining Workshop (2006) - Five invited and extended papers, pages 126–146. ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2007.10.008. URL: http://www.sciencedirect.com/science/article/pii/S0169023X07002078.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ