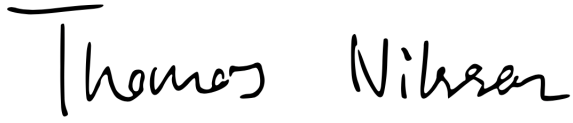
DTU Compute
Department of Applied Mathematics and Computer Science

{2.7182818284}ϑφερτυθιοπσδφγηξκλ

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 library implemented in the Flutter framework. The final version of the package enables the programmer to generate a set of mobility features with just 3 lines of code. A field study was conducted where a mobile application running the package was used by 10 participants. The application collected the participants' location several thousand times per day and participants would fill out a small questionnaire each day pertaining to the features. In comparing the answers to the computed features, promising results were achieved for the participants which were diligent in tracking their location and providing answers.

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 16, 2020



Thomas Nygaard Nilsson, s144470

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 helping with the thesis writing I would like to thanks Darius Rohani and Marie Mørch for taking interest in reading and correcting my thesis, severely aiding the writing process and outside-reader perspectives.

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.

Most people walk around with a very powerful computer in their pocket, that contains a variety of sensors capable of painting a picture of the user's current context. A user's context is a set of digital characteristics, also called features, which arise from an individual interacting with their environment, such as when moving around between places in one's everyday life. This thesis will describe the design and implementation of a software package for the Flutter framework capable of generating a fully-automated mobility context in real-time, by tracking location data on a smartphone. The package will in the future integrate into the CARP Mobile Sensing Framework (CAMS)[1] which provides an array of digital phenotyping capabilities to an application developer.

The research problem addressed in this project is focused on the software engineering aspect of developing a digital phenotyping library and relates to the *re-usability of source code* as well as the current *lack of support for real-time computation.*

Firstly, there have been numerous contributions within the field of using smartphone data for phenotyping and predicting the user's state for many years. However, much of the research conducted, specifically within the field of user-mobility context generation, has been done without publishing source code in a way such that it may be used by other researchers in the future. Mainly this comes down to researchers not publishing the source code at all, due to source code not being the focus of their research. Sometimes the source code is released, however, but then over time suffers from an (understandable) lack of maintenance and thus becomes obsolete. Furthermore, researchers tend to focus on only the Android platform since this is the more

[1] `https://github.com/cph-cachet/carp.sensing-flutter/wiki`

prevalent platform of choice in many countries, however, this is not a good research practice for reasons we will elaborate on later.

In addition, the existing contributions gather all of the data before feature computation is carried out. This has the clear advantage of making it easier to run and evaluate the mobility feature algorithms. However, this also has the disadvantage that the feature generation is not possible in real-time, where they are the most useful. Certain features rely on prior contexts being computed and readily available and are therefore easily implemented when done in an off-line fashion on a desktop computer. However, in a real-time situation on a phone, these prior contexts will need to be stored on a device that has limited storage, and the algorithms must be adjusted to read in these stored set contexts. The proposed software package will solve the problem of implementing these algorithms from scratch as well as allow the application programmer to compute the user's mobility context in real-time.

Researchers within the field of mental health illnesses from a health-tech background will benefit greatly from using a Flutter package with a high level of abstraction, in which the researcher's focus is moved from software engineering to data analysis and study design. Currently, researchers often choose to start from scratch which is a laborious task that takes up valuable time which could be spent on more relevant work. Furthermore, since Flutter is capable of compiling to both iOS and Android source code, studies need not be restricted to a single mobile platform, hence why Flutter was the app-framework of choice.

An additional benefit of computing features on the phone itself, in contrast to computing it on a central server or desktop, is the anonymization of GPS data; the generated context contains information that is much less sensitive and therefore has implications on privacy as well as GDPR compliance.

Major Depressive Disoder (MDD), commonly known as depression, is a serious illness that is resource-intensive to treat using traditional methods such as face to face consultations. This costs society a lot of money, and it is, therefore, worthwhile to explore treatments that can improve the quality of life of these patients.

Among many treatments, Behavioural Activation is very well supported by mobile sensing technologies which have become a growing field due to the rise of ubiquitous computing. Currently, the diagnosis in bipolar disorder, also known as manic depression, relies on manual patient information and clinical evaluations and judgments with a lack of objective testing. Some of the essential clinical features of individuals suffering from bipolar disorder are changes to their daily behavior [**objective_smartphone_data_as_** These changes in behaviour can be captured by the user's *mobility context*, and can thus provide very helpful in customizing a Behavioural Activation treatment. A mobility context can be generated by user rules and triggered by events from semi- or full- automated algorithms. Here, it will be investigated how the context can be generated in a fully-automatic, real-time fashion.

Clustering location data into places has been done previously in a more general context by Ashbrook and Starner [**learning_significant_locations**]which used data tolearn significant locations and predicting where the user will move next. Cuttone et al. [**sparse-location-2014**] have contributed to the field by finding that even low accuracy sampling can be used to identify significant user locations. Recently, location data has also been used within the context of depression by Saeb et al. [**Saeb2015**] and Canzian et al. [**Canzian2015**]. These two contributions define a list of mobility features to track changes in user behaviour. In addition Saeb et al. [**Saeb2015**] also showed that certain mobility features correlate strongly with scoring highly on the Patient Health Questionnaire (PHQ-9) (see Appendix **??**), which indicates an individual is depressed. In regards to Behavioural Activation, Rohani [**mubs-rohani**; **moribus**] developed a recommender system for treating depression while working on his PhD at CACHET. The current system relies on manual user input, and an ideal improvement would be to incorporate automatically generated mobility features recommendation-algorithm. Lastly, the work done at CACHET on the CARP Mobile Sensing Framework[2] within mobile sensing provides an ecosystem in which the resulting package would fit in.

Existing contributions within the field of generating mobility features and context are cumbersome, if even possible, to reproduce with regards to their source code. Furthermore, the algorithms that exist in the literature do not necessarily lend themselves to the real-time computation of features. How these two things can be achieved will be investigated in this thesis and are addressed in the following research questions:

[2] `https://github.com/cph-cachet/carp.sensing-flutter/wiki`

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 API design of such a software package look like?*

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.

The paper [**Saeb2015**] 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 procedure described in [**sparse-location-2014**] producing intermediate features has already been carried out by Jonas Busk prior to starting this thesis. From these intermediate features, many of the mobility features can be easily derived.

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, however the *Routine Index* feature as defined by [**extraction-of-behavioural-features**] and [**Saeb2015**] (here called the *Circadian Movement*), may prove to be a challenge to implement in real-time. By its basic definition, this feature lends itself to be computed only once several full days of data have been collected. This makes the feature non-trivial to implement in a real-time fashion where it may computed at any time.

An algorithm/procedure that runs the Mobility Feature algorithms on a smart-phone has to be written which involves a couple of steps. Firstly, 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.

The software package should be designed according to the Flutter packages best practices [3], be properly documented and be released on the Dart package manager [4].

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. The functionality of the application should be in accordance with the contents of *Goal 3*. By developing an actual application and seeing the algorithms in use, the structure of the API should emerge such that it is 'nice to use' from an application developer's point of view.

To test the quality of the Mobility Features in action, a small scale study of 5-10 participants should be conducted and should run for 2-4 weeks (the longer the better). The participants will use the demo application. Also, in order to validate the quality of the features produced daily during these weeks, the participants will have to fill out a small diary in order to compare the results of the algorithms with the subjective experience of the users. This diary will be a part of the demo application. Preferably the users should be reminded daily to fill out the diary such that as much user data is collected as possible.

This section will provide an overview of the thesis in the form of a short summary of each remaining chapter.

In this chapter, the most relevant research with regards to ubiquitous computing and generating mobility features will be laid out. This includes relevant contributions within the broader context of mobile sensor context generation, algorithms for pre-processing spatial data, clustering GPS data, how features can be generated from geospatial clusters, and lastly what the features can be used for.

[3] `flutter.dev/docs/development/packages-and-plugins/developing-packages`
[4] `www.pub.dev`

Here, it will be discussed which requirements the final product should fulfill. This includes the mathematical definitions and the data model for the *Mobility Features*. This includes the algorithms for computing these, and how some of these algorithms have been adapted such that they work on an incomplete, real-time generated dataset.

Here, it will be discussed how the algorithms were concretely implemented both in Flutter such that they fulfill the functional requirements outlined in the *Algorithm Design* chapter.

Here, it will discus how the implemented algorithms were wrapped up into a software package with a public-facing API. This includes public and private data, API considerations and documentation. Furthermore an integration to CAMS will also be discussed.

Validation of the package will be discussed both from a quantitative- as well as a qualitative perspective. The quantitative validation will verify that the algorithms produce meaningful and correct results, which will be validated by means of unit testing as well as data analysis in which subjective data reported by participants will be compared to the features computed by the algorithms. The qualitative part will be much more focused on how the software package was designed, i.e. the ease of use for the application developer as well as design of the API. Concretely this can be evaluated in terms of the number of lines an application developer needs to write in order to use the package.

Here the results will be discussed as well as how one may go about developing the software package further, including changing the existing algorithms and which design- and implementation choices could have been made differently. This includes how the API is designed with regards to trade-offs between varying the abstraction-level, complexity and openness of the API.

The most important findings and things that could be worked on in the future will be described in this chapter.

This chapter will describe what work has been done within the treatment of depression related to digital phenotyping and context generation. Secondly, a series of previous contributions will be presented which deal with computing mobility features including how these features can be used in a medical context. Thirdly, a brief insight into existing mobile sensing frameworks will be given, including how the *Mobility Features Package* will fit into one of these frameworks. Lastly, an example of a recommender system is given for which the *Mobility Features Package* is an ideal use case.

The paper [**digital_phenotyping**] 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 easy access to clean water, ownership of a smartphone and by proxy, rapid 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 [**digital_phenotyping**]. Another current problem is how it 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 is referred to was *Digital Phenotyping*, 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 al-

lows researchers to track patients in their own environment, rather than in a clinical environment.

However, when does measurement become surveillance? Is phenotyping by using data such as geo-location too invasive? A series of ethical issues have to be addressed before digital phenotyping can realistically hope to be employed as a tool for population health. Although, some of these issues have technical solutions, such as with tracking human-computer-interactions which is mostly related to *how* the user interacts with a device, i.e. the pacing with which is typed, scrolled or how long phone calls last - rather than the content of what is provided to the device i.e. what is typed, spoken during a phone call, or the websites visited.

This thesis by [**learning_significant_locations**] from 2002 is one of the earliest contributions to present a system that uses GPS data to infer user context and use this context.

For data collection, a wearable GPS receiver was used which tracked user location for four months with an accuracy of 15 meters, meaning the same physical location may have resulted in slightly different GPS coordinates from day today. For pre-processing data some of the data was discarded by setting the sensor to only log data while the user was traveling at a speed greater than 1 mile per hour, with the average human having a walking speed of about 3 miles per hour.

Given a data set of GPS points, a *significant place* is found by examining data and finding periods of low user-movement. The duration of the period was (somewhat arbitrarily) chosen to be 10 minutes. Due to the 15 meter accuracy of the sensor, the location coordinates found may be somewhat noisy from day today. In order to identify the mean of these noisy place locations, a modified version of the *K-means* clustering algorithm was applied to the dataset, with a radius parameter. The optimal radius for the clustering algorithm is found by examining the graph of the number of clusters found (K) as a function of the radius. The knee of the graph (see figure **??**) signifies the radius just before the number of locations begins to converge to the number of clusters found. This procedure effectively figures out which significant places belong to the same *location* and which are outliers points which should be

considered noise, i.e. they either do not belong to a location or there is too low confidence to say whether they do or not.

Within a location, several *sub-locations* may exist, which are perhaps best exemplified by a university campus, as a location, having several buildings for different departments, i.e. *sub-locations*. To find these, all *significant place* data-points belonging to a *location* is given as input to previously described clustering algorithm. The optimal radius is once again found by looking at the knee on the graph and from this optimal parameter choice, the *sub-locations* within a given location are identified.

A Markov model was set up with each state representing a Location and transitions representing real-life transitions between locations. To compute the probability of a path, or a transition between places, the relative frequency of the path was computed.

The DBSCAN algorithm is an essential algorithm for clustering GPS data points and was published by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu in 1996 [**density-based-1996**]. The core concept of DBSCAN is to cluster location data based on a density measure rather than a pure distance measure, such as is

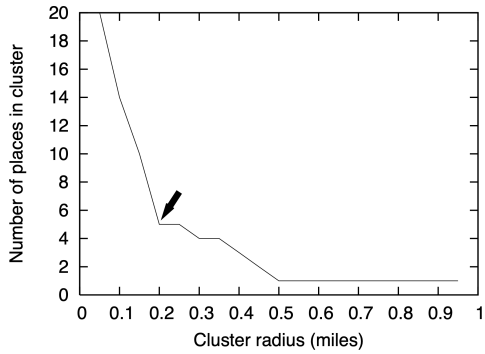


Figure 2.1: The number of places found in the dataset the location radius parameter changes. The arrow indicates the knee in the graph at which point the radius will be used to find *sub-locations*. Source [**learning_significant_locations**].

the case with the *K-means* clustering algorithm. This allows the clusters to take on nearly any shapes and sizes, in contrast to round shapes which are found by using *K-means* and *Gaussian Mixture Model*, see figure **??** for examples.

GPS data has a higher density inside clusters than outside the clusters and the density in noisy areas is lower than density in clusters. Here, *noisy* refers to points that are spread randomly within some areas and do not cluster around a centroid.

The DBSCAN algorithm will, given a set of geospatial data points and a small set of parameters, find these dense clusters, as well as noisy data points, where the clusters correspond to places and the noisy data points, are outliers which do not belong to a place. The output of the algorithm is labeling of each point in the input dataset as either belonging to a cluster or being a noisy data point.

The DBSCAN paper defines a *Neighbourhood* as a collection of data points within some radius, i.e. an area defined by a distance function. Within a *Neighbourhood*, two types of points can reside: *Core points*, that is, points which are inside in the neighborhood, and *Border points* which lie on the edge of the neighborhood and delimit it.

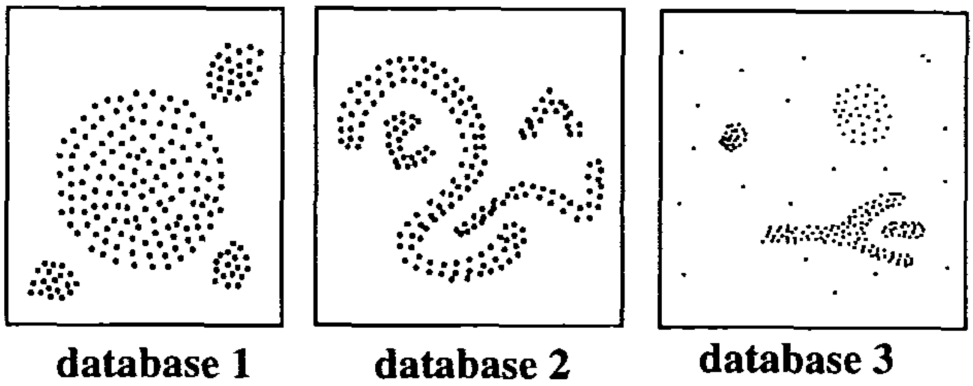


Figure 2.2: Data from three different databases with very distinct cluster shapes. Database #1 has very round clusters which would identified correctly with K-means whereas database #2 and #3 have clusters which would require a different clustering approach. Source: [**density-based-1996**].

An *Epsilon Neighbourhood* is a *Neighborhood* defined on a point p which includes all the points q which are inside a radius of ϵ of p. Formally this is defined as:

$$N_\epsilon(p) = q \in D \quad | \quad dist(p,q) \leq \epsilon$$

Note: The N_ϵ of a border point contains much fewer points than that of a core point.

We Require for all points p and a given cluster C: There must be a a point $q \in C$ st. $p \in N_\epsilon(q)$ of and $N_\epsilon(q)$ contains at least T_{min} points where T_{min} is a parameter which defines the minimum number of points required.

A point p is *Directly Density-Reachable* from point q wrt. the parameters ϵ and T_{min} if $p \in N_\epsilon(q)$ and $|N_\epsilon(q)| \geq T_{min}$ (*Core Point Condition*).

This implies a border point p may be *Directly Density-Reachable* from a core point q, but it is likely, not true the other way around, since $N_\epsilon(p)$ is probably smaller than the minimum amount of points needed to satisfy the *Core Point Condition*.

A point p is *Density-Reachable* (DR) from a point q if there exists a chain of points $p_1, ..., p_n$, where $p_1 = q$ and $p_n = p$, such that p_{i+1} is *Directly Density-Reachable* from p_i. This relation is denoted $DR(p,q)$.

Note that, only symmetric for core points, but not symmetric in general and that two border points from the same cluster may not be Density-Reachable due to them not satisfying the Core Point Condition.

A point p is *Density-Connected* (DC) to q if there exists a point o st. both p and q are *Density-Reachable* from o, this is denoted $DC(p,q)$.

Density-Connected is a symmetric relation meaning that $DC(p,q) \iff DC(q,p)$.

A cluster is a set of *Density-Connected* points which is have maximal *Density-Reachability*.

That is, for all points p, q, if $p \in C$ and $DR(p, q)$ then $q \in C$.

For all points $p, q \in C$ it holds that p is *Density-Connected* to q, i.e. $DC(p, q)$.

The parameters ϵ and T_{min} are global, i.e. the same for all clusters.

Noisy points are defined as all the points p for which $p \in D \wedge p \notin C_i$ for all clusters $i = 1, ..., N$, that is, all the points not belonging to any cluster.

Let p be a point in D for which the core point condition holds. Then, the set O which is the set of *Density-Reachable* points from p (wrt. the parameters) is a cluster wrt. to the parameters. This means, a cluster C contains exactly the points which are *Density-Reachable* from an arbitrary core point of C.

Let C be a cluster wrt. to the chosen parameters and let p be a core point in C, then it holds that $C = O$.

There is no reliable way of knowing the parameters ϵ and T_{min} for each cluster in advance. Instead, the parameters are determined based on the least dense cluster and used globally for all clusters.

Start with a random point p and get all *Density-Reachable* points from p wrt. the parameters. Given $p \in D$:

- If p is a core point, then $O = DR(p)$ is a cluster (*Lemma 2*).
- Otherwise, if p is a border point, then $DR(p) = \{\}$ and the algorithm proceeds to the next point q

Once all points have been considered, the algorithm terminates.
Note: Since the parameters are global, the algorithm may merge two clusters according to *Definition 5* if two clusters are close to each other (even if the two densities are different).

This paper by Saeb et al. published in 2015 [**Saeb2015**] forms the basis for the context generation of this thesis, with regards how one may derive mobility features from GPS data with the features *Location Entropy* and *Circadian Movement* being new contributions at the time at which the paper was written. In addition, this paper also shows how certain features, in particular, correlate strongly with a *PHQ-9* questionnaire score, which is relevant in a mental health context. Since the *PHQ-9* questionnaire requires manual input from the user, and the user may forget to fill out the questionnaire, the strong correlation is great news. This could imply that it is possible to automate the patient data gathering process by simply tracking the location of the patient with a high frequency, and thus get around the issue of manually gathering biweekly subjective questionnaire data. The *PHQ-9* questionnaire (see Appendix **??**).

The pre-processing of GPS data is split into two phases: Firstly, each data point is labeled as either being in a stationary or transitional state, for which the time-stamp of the current, the previous, and the next data point is used. By using this time difference as well as the distance between points the average velocity is calculated. The labeling is afterward done by using a threshold of 1 km/h; a speed lower than the threshold indicates the state is stationary and higher is a transitional point. For analyzing the data further, only the stationary points are considered. Secondly, the stationary points are processed using k-means clustering to identify 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. Concretely the algorithm for finding K is to increase K until the largest cluster found has a radius of 500 meters or lower.

After the pre-processing step, the following features can be computed:

Number of Clusters: This feature represents the total number of clusters found by the clustering algorithm.
'Cluster' and 'Place' may be used interchangeably when describing features. A cluster is simply the mathematical term for a collection of data points which corresponds to a place with some GPS coordinates in the real world.

$$N = K$$

Location Variance (LV): 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.

$$LV = \log(\sigma_{lat}^2 + \sigma_{lon}^2 + 1)$$

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. Calculated as

$$H = -\sum_{i=1}^{N} p_i \cdot \log p_i \tag{2.1}$$

where each i represents a location cluster, N denotes the total number of location clusters, and p_i is the percentage of time the participant spent at the location cluster i.

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.

$$H_N = \frac{H}{\log N}$$

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.

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.

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

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

Circadian Movement: This feature measures to what extent the changes in a participant's location follow a 24-hour, or circadian, rhythm. To calculate the circadian movement, we obtained the distribution of the periodicity of the stationary location data and then calculated the percentage of it that falls in the 24±0.5 hour periodicity. However, the paper does not describe in great detail how this feature is implemented.

The features which correlated strongest with the *PHQ-9* scores over two weeks were the following:

Circadian Movement: This features correlated negatively with the *PHQ-9* score, and can be interpreted as depressed individuals tend to have less of a routine than healthy individuals.

Location Variance & Normalized Location Entropy: These features have a similar meaning and both correlate negatively with the *PHQ-9* score. The takeaway here is likely that visiting very different places every single day can be seen as a sign that the individual is depressed, which also correlates with the Circadian Movement, since visiting different places every day results in a low level of routine.

Home Stay: This feature had a Strong positive correlation with the *PHQ-9* score, and thus the main take away from this feature is that spending a lot of time at home is an indicator of being depressed.

The paper finds that depressive symptoms tend to change slowly over weeks, with little day-to-day variation and thus it makes more sensors to group data on a weekly basis. This may explain why two weeks of sensor feature correlated more strongly with the *PHQ-9* than either daily sensor features or EMA measures. Unlike EMA ratings, which are momentary, the *PHQ-9* assessment reports symptoms over a period of two weeks. The study conducted suggests that it is possible to monitor depression passively using phone sensor data, and in particular, GPS. Most people are unwilling to answer questions repeatedly over long periods of time, while passive monitoring could improve the management of depression in populations, allowing at-risk patients to be treated more quickly as symptoms emerge, or monitoring patients' responses during treatment.

Published in 2019 [**extraction-of-behavioural-features**], this paper focuses on many channels for collecting user data, with each channel resulting in a set of features. Examples of channels are GPS data, phone usage, step count, etc. The upside of having many channels is improved accuracy since more data is available to describe the behavior of the user, and if certain channels are inactive during certain periods of time, the other channels can cover the missing data, and thus periods of missing data will be avoided with high likelihood.
The paper uses the *AWARE Framework* [**aware2015**] to collect data such as *Bluetooth*, *Call-log*, *Location*, *Campus Map*, *Phone Usage*, *Step Count* and *Sleep State* (via FitBit).

The daily behavior of the user is predicted through four high-level features extracted from the data channels.

User mobility: The movement patterns of the user are derived from location tracking and the campus map feature.

Communication Patterns: Derived from call log and text messages.

Mobility and communication: Derived from Bluetooth device IDs within range, i.e. Bluetooth scan.

Physical activity: Derived from step-count during different windows of the day, i.e. when the user is walking a lot.

The day (24 hours) is split into 4-time windows, which helps identify where the home is since the user likely sleeps the same place most days of the week, which is likely during the *night* time-window.

- Night (00-06)
- Morning (06-12)
- Afternoon (12-18)
- Evening (18-00)

GPS coordinates collected throughout the day have a time stamp connected to them, which makes it possible to calculate a list of interesting features within a day of the user moving around. The features are similar to those found in [**Saeb2015**] adds two additional features, namely *Radius of gyration* and *Time spent at top3 clusters*.
However a couple of additional contributions are found. Firstly a *bout* is defined as a small window of time, which can be defined based on the granularity wanted, i.e. 5 minutes.
Furthermore, a clear definition for the *home* cluster is given as being everything within 10 meters of the center of the home location center. Time spent at home is defined at the time spent within 100 meters of the home centroid. The activity of studying is defined as spending 30 minutes or more in an academic building, while sedentary, which is taking fewer than 10 steps per *bout*.

From the Bluetooth Scan, the devices found each has a unique ID and can, therefore, be categorized into the following groups:

- Self, i.e. the user (the device scanned most often)
- Related, ex-colleagues, flatmates (scanned less often)
- Other, other people (scanned least often)

The three categories are created through K-means clustering, by firstly creating two clusters ($K = 2$), i.e. self and others. Secondly another model is fitted to the data with $K = 3$ clusters, and the model which fits the data the best is chosen, i.e. through cross-validation or BIC.

Phone usage is based on the screen state which at any time can be only one of the following: On, off, lock, unlock. Tracking the state of the screen together with the timestamp at which the state changes, a few features can be calculated, which are:

- Number of unlocks per min
- Time spent interacting with the phone
- Total time unlocked
- Hour of the day of first unlock/turned on
- Hour of the day of last unlock, lock, turned off

- Time spent interacting with the phone (sum of time between unlocking and off/lock)
- Standard deviation of time spent interacting with the phone

By tracking heart rate, and other signals with a FitBit watch, the Fitbit API gives access to the *sleep state* of the user, which at any time is one of the following: Asleep, restless, awake, unknown.
A number of samples will be made during the night and thus the simplest features which can be created are the number of samples in each of the four states. From this, two higher-level features are computed, which are:

Weak Sleep Efficiency: $WSE = \frac{n_{asleep}+n_{restless}}{n_{asleep}+n_{restless}+n_{awake}}$

Strong Sleep Efficiency: $SSE = \frac{n_{asleep}}{n_{asleep}+n_{restless}+n_{awake}}$

With n_{state} being the number of samples for that state.

From the step-count channel features pertaining to the fine-grained activity, without the location, for a whole day, can be calculated:

- Number of steps
- Max steps in a *bout*
- Number of active *bouts*
- Min, Max, Avg. length of sequence active *bouts* in a row

The change in the user's behavior over a longer time period, i.e. a few weeks can be tracked by using a number of the proposed features, and is done by applying a simple linear regression as well as a piecewise linear regression (with two lines) to the dataset. The calculate behavioral change features are then the following:

- Derivatives (a_i) of the linear model: $y = a_1x_1 + a_2x_2 + ... + a_nx_n + c$.
- Change in slope (derivative) for each behavioral feature

Given data for $n \in \mathbb{Z}$ timesteps, i.e. weeks, choose some breakpoint $m < n \in \mathbb{Z}$ and create the piecewiese linear function:

$$\begin{cases} a_1 x_1 + a_2 x_2 + ... + a_n x_n + c & t < m \\ b_1 x_1 + b_2 x_2 + ... + b_n x_n + d & m \leq n \end{cases}$$

The two-line segments are then fitted to the data with a feature selection algorithm. The resulting slope vectors for the selected features are then behavioral features.

Human Mobility is relevant for a number of different applications such *disease spread, urban planning*, *managing traffic* as well as understanding *social interactions*. Previous contributions within these fields have been using Call Detail Records (CDR) which is the art of triangulating the user's location by use of cell-phone towers. These sources produce very coarse-grained estimates wrt. location and time. For this paper a more traditional location data sampling was used, using the GPS tracker in smartphones, however, 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 study with 6 participants was conducted in which location data collected continuously and the users filled out an online diary on a daily basis. Given a series of temporally-ordered data points for a user, the aim was to compute features called *Stops* and *Places of Interest (POI)*. A *POI* is defined as a location of high relevance such as that person's school, gym or a supermarket as is denoted by a type $POI = (ID, lat, lon)$ and a *Stop* at a POI is defined as the period of time in which the user stayed at a POI with a given ID, i.e. $Stop = (arrival, departure, ID)$. To find Stops, both a *Gaussian Mixture Model*, as well as the *DBSCAN* algorithm, were applied for clustering. To evaluate the best model, the f_1 score of each model was computed, and in the end, it was found that both algorithms performed similarly for each of the participants.

In this paper by Canzian et. al from 2015 [**Canzian2015**], it is discussed how existing systems for diagnosing depression require the user to interact with the device. These interactions can be input such as mood-state a few times a day, which can be highly subjective and error-prone. The paper addresses this issue via objective patient data, called *Mobility Features*, which was collected via unobtrusive monitoring by using

smart-phone location data. There was a significant correlation between these *Mobility Features* and depressive moods similar to [**Saeb2015**]. The paper also describes concrete models to *predict* changes in depressive mood by using these features. The paper uses the *PHQ-8* questionnaire as a reference, which is the almost the same questionnaire as the *PHQ-9* (see Appendix **??**), except for only including the first 8 questions. While this paper describes many of the same features as [**Saeb2015**], it does so in a much more detailed and mathematical way, such as with the *Routine Index* feature, which corresponds to the *Circadian Rhythm* feature in [**Saeb2015**].

The feature described is as follows:

Place: $p = (id, t_a, t_d, c)$
A place is a cluster of geo-location points with a unique id and a geographical center $c = (lat, lon)$. When the user arrives at the place it happens at t_a and the user departs from the place at t_d.

Mobility Trace: $MT(t_1, t_2) = [p_1, ..., p_n]$
A mobility trace is a list of all places visited in the time interval (t_1, t_2). $N(t_1, t_2) = n$ is the number of places visited and the time gap between $t_d(p_i)$ and $t_a(p_{i+1})$ is a period of movement.

Total Distance: $D_T(t_1, t_2)$
The total geodesic distance traveled in the time interval (t_1, t_2).

Max Distance: $D_M(t_1, t_2)$
The maximum distance between any two places visited in the interval (t_1, t_2).

Radius of Gyration: $G(t1, t2)$
The geographical area covered in the interval (t_1, t_2)

Standard Deviation of Displacement: σ_{dis}
The standard deviation between each displacement i.e. distances from one point to another.

Home Cluster: H
The cluster most frequented cluster at 02:00, 06:00 and 20:30 on week-days.

Max Distance from Home: $D_H(t_1, t_2)$
The distance to the point the furthest away from home, during the interval (t_1, t_2).

Number of different places visited: $N_{dif}(t_1, t_2)$
The number of different clusters found in the interval (t_1, t_2).

Number of different significant places visited: $N_{sig}(t_1, t_2)$
The number of significant places visited in the interval (t_1, t_2). The maximum number of significant places is 10.

Routine Index: $R(t1, t2)$
The degree to which the place-time distribution during the interval (t_1, t_2) on a given day is similar to the distribution on other days, during the interval (t_1, t_2).

In addition to concrete definitions of features, the paper also provides an algorithm for sampling location in a way which saves phone battery. GPS data collection is very expensive if done with a high frequency and by using cheaper sensors to turn it off when necessary can, therefore, save a lot of battery. Concretely the accelerometer is used by an activity recognition algorithm to determine the user's activity state. Concretely this is done by modelling the user as being in a movement state, of which there are 3: *Static (S)*, *Moving (M)* and *Undecided (U)*, where location tracking only takes place in the U or M states.

The definitions for the states are as follows:

Static (S)
The user is in the same location i.e. school/work. Never sample location data while in this state since it is unnecessary.

Moving (M)
The user is moving from place to place and the phone should, therefore, keep location tracking on, while in this state.

Undecided (U)
It is unknown whether the user is moving or not and moving to either state S or M is now impending. When this state is entered, the user's location l_1 is sampled, and after 5 minutes the location is sampled again, providing l_2. Then, The distance $d = dist(l_1, l_2)$ is calculated and from this, the algorithm will transition to either S or M depending on the distance between the two locations.
The transitions are made, as follows:

U $\rightarrow$ **S**: If the distance $d < 250$ m.

U $\rightarrow$ **M**: If the distance $d \geq 250$ m.

S $\rightarrow$ **U**: If two consecutive activity recognition samples indicate user activity with over 50% confidence.

M $\rightarrow$ **U**: If two consecutive samples are less than 250 m apart, or when an activity recognition sample indicates the user is still with more than 50% confidence.

The *AWARE Framework* [**aware2015**] is an open-source toolkit and a reusable platform for capturing, inferring, and generating user-contexts 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* therefore aims to ensure an easy way of collection contexts for the application developer, and it is demonstrated how the tool can reduce the software development effort of researchers when building mobile tools for developing context-aware apps and doing so with minimal battery impact. 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 and supports a number of data channels [1] such as the built-in sensors, as well as *Application Usage*, *SMS*, and *Phone Call Logs*. Some of the channels are however not available on iOS due to the iOS developer API, in general, being more restrictive than that of its Android counterpart.

The prevalence of depression has led to the development of many mobile health-tech solutions and applications for monitoring diseases related to depression. However few of these applications have focused on the treatment of depression. There is clinical evidence showing that depressive symptoms can be alleviated through Behavioural Activation which is a behavior-focused treatment method for treating depression. Rohani is a postdoc at CACHET and has published and developed two systems during his Ph.D. within the realm of Behavioural Activation which is the MORIBUS- and MUBS systems.

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

The *MORIBUS* system [**moribus**] by Rohani aims to make it easy for depressed patients to remember and register their behavior in order to increase their every-day positive outcomes. The system comes with a catalog of 54 pre-made activities such as *Get ready in the morning* and *Eat breakfast* and has support for patients to add their own activities by manually inputting text and a rating. When an activity is completed, the patient registers this in the app along with a short reflection. The application is able to provide the user with statistics and summaries of activity frequency and reflections.

The *MORIBUS* system later evolved into the *MUBS* system which was rewritten in Flutter [**mubs-rohani**]. The application uses a larger catalog of pleasant activities and gives recommended activities to perform, personalized to each user. The catalog is a database of 384 common enjoyable activities where each activity has been manually labeled by researchers with an *activity level*, as well as a *category*. The recommender model uses a Naive Bayes model with a set of features representing each feature and a corresponding label representing whether it is positive or negative for that individual user. The model is designed to recommend already-known activities as well as novel activities to the user which is in the spirit of behavioral activation. Future work of the system includes adding additional features such as *Location*, *Local Weather* and *Ambient Noise* to provide an even more detailed context to the system.

Both systems rely on manual input from the user in order to learn which activities to recommend in the future. This can pose a problem, especially when running longitudinal studies since participant retention may suffer if the input process is too involved for the user. These systems by Rohani, therefore, provide examples of use cases for the automated feature generation algorithm such as the *Mobility Features Package* which is easy to integrate into an existing Flutter application.

The CARP Mobile Sensing (CAMS) Framework[2] is a Flutter framework for adding digital phenotyping capabilities to a mobile-health app. An integration for the *Mobility Features Package* into the CAMS Framework is a future goal beyond this thesis, but describing CAMS will provide some insight into the direction of the package.

[2] `https://github.com/cph-cachet/carp.sensing-flutter/wiki`

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, by enabling the programmers to add an array of mobile sensing capabilities in a flexible and simple manner. This would include, firstly, adding support for collecting health-related data channels such as *ECG*, *GPS*, *Sleep*, *Activity*, *Step Count* and much more. Secondly, to format the resulting data according to standardized health data formats (like *Open mHealth* schemas [3]. Thirdly, to upload the collected data to a specific server, using a specific API (such as *REST*), in a specific format, such that it may be used 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 in terms of a solid programming API and a runtime execution environment, that is being maintained as the underlying mobile phone operating systems and APIs are evolving. Moreover, the focus is on providing an extensible API and runtime environment, which together allow for adding application-specific data sampling, wearable devices, data formatting, data management, and data uploading functionality to an application. In addition, in order to reduce the number of integrations needed, the framework must be cross-platform such that it supports both Android and iOS, without having a codebase for each platform.

The data collection pipeline is defined firstly by a *Study* which contains all the information needed to conduct a real-world study, which includes the collection of certain data types with a set of rules and saving it somewhere.

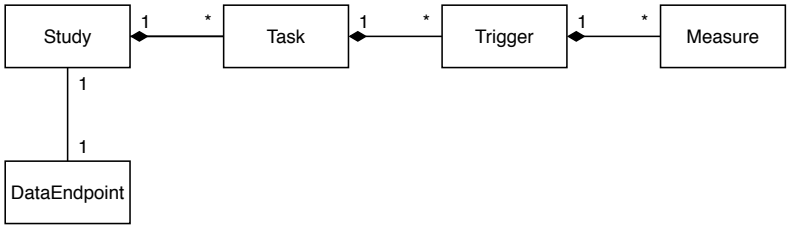


Figure 2.3: A UML diagram of the *CAMS* Domain Model..

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

Concretely, a *Study* holds a set of *Triggers*, which define *when* sampling is carried out, such as by scheduling sampling at given time each day, or when a certain event is registered, such as entering a specific *GeoFence*[4].
Each *Trigger* holds one or more *Tasks*, each of which define which *Measures* should run simultaneously. In addition, a *Task* also defines whether or not data is sampled in the background, or whether the user needs to interact with the device in order to perform sampling.
Each *Task* holds a set of *Measures* each of which defines what to sample, i.e. which data channel to listen to. Lastly, a *Study* also holds a reference to the *DataEndpoint* specifying which where data ends up, i.e. by uploading it to a server.

[4] `https://developers.google.com/location-context/geofencing`

This chapter will describe the theoretical background for the Mobility Features Package. Firstly discussion of location tracking will be given. Afterward the features will be described and discussed. Lastly, a concrete, detailed mathematical definition will be given for each feature.

The Navbar Global Positioning System (GPS) is a space-based radio-navigation [**gps-navstar**] system which provides location data to GPS receivers such as the one found in smart-phones. GPS is capable to 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 such as in [**learning_significant_locations**] in order to get this data, but nowadays smart-phone contain GPS receivers which enables users to use a variety of navigation services and applications with their phone alone. From a developer perspective, tracking location data on the Android- or iOS platform works by using an Application Programming Interface (API) which uses the GPS receiver inside the phone to place it on the Earth geographically. This API allows the streaming of location data in a continuous fashion which is used for navigation such as with Google Maps, where the user's location changes continually and it is important that the user is notified if they have to make a turn in due time.

As mentioned, location data is defined by a latitude- and longitude coordinate which map to a point on the globe, however since the earth is a sphere and not a plane, a standard Euclidean distance metric cannot be used to calculate the distance between two points on the surface of the Earth. A few methods of calculating the distance between GPS coordinates exist, one of the fastest to compute being the Haversine formula which calculates the great-circle distance between the two points on a sphere. The Earth is however not exactly spherical, and therefore the Haversine is just an approximation which works well when the distance has to be calculated many times, such as when calculating the distance of a path consisting of many points close to each other.

Given the radius of a sphere r and two points on the sphere, A and B, the haversine distance [**haversine-formula**] between the two points can be directly computed as

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

The Mobility Features which were used were a subset of the features discussed by [**Saeb2015**; **Canzian2015**]. In addition a set of, what we will refer to as *intermediate features* were also used which from the work of [**sparse-location-2014**].

Common for many of the algorithms for finding user mobility features is that they rely on clustering of data points, in order to find the number of places. However when dealing with large amounts of data points it may be necessary to reduce the initial amount of data points such that these clustering algorithms are able to run faster. This down-sampling process will be carried out by clustering raw data points into what we shall refer to as *Stops* indicating locations where the participant did not move around a lot. The *Stop* notion is loosely based on the [**sparse-location-2014**]. The pre-processing produces *intermediate features* from which the final mobility features are derived. We define these *intermediate features* as follows:

A collection of GPS points which together represent a visit at a known Place (see below) for an extended period of time. A *Stop* is defined by a location that represents the centroid of a collection of data points, from which a *Stop* is created. In addition a *Stop* also has an *arrival-* and a *departure* time-stamp, representing when the user arrived at the place and when the user left the place. From the arrival- and departure timestamps of the *Stop* the duration can be computed.

A group of stops that were clustered by the DBSCAN algorithm [**density-based-1996**]. From the cluster of stops, the centroid of the stops can be found, i.e. the center location. In addition, it can be computed how long a user has visited a given place by summing over the duration of all the stops at that place.

The travel between two Stops, which the user will pass though a path of GPS points. The distance of a Move can be computed as the sum of using the haversine distance of this path. Given the distance travelled as well as departure and arrival timestamp from the Stops, the average speed at which the user traveled can be derived.

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.

The features derived from this class are *Home Stay (%)*, *Location Variance*, *Number of Places*, *Entropy Normalized Entropy* , *Distance Travelled* and the *Routine Index (%)*. These features are computed daily which for example means Home Stay refers to the portion of the time today which was spent at home. Below a short definition of each feature will be given since some of them differ somewhat from the descriptions given by [**Canzian2015**; **Saeb2015**]. A more detailed description will be given in Section **??**.

The portion (percentage) of the total time elapsed since midnight which was spent at home. Elapsed time is calculated from the departure time of the last known stop.

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

Figure 3.1: An *Hour Matrix*.

The statistical variance in the latitude- and longitudinal coordinates.

The number of places visited today.

The entropy with respect to time spent at places.

The normalized entropy with respect to time spent at places.

The total distance travelled today (in meters), i.e. not limited to walking or running.

The percentage of today that overlapped with the previous, maximally, 28 days.

When defining a *Place*, several edge cases come to mind such as visiting multiple buildings on a university campus, visiting two stores right next to each other, walking around in a park, i.e. having a large dispersion of the data points. There are likely many more edge cases which exist which have not been considered, but we need to generalize when coming up with a definition. To generalize we define *Stops* using a radius parameter r_{stop} to group data points, and likewise a radius parameter r_{place} to cluster Places. These parameters define the maximally allowed dispersion of the data points which allows us to filter out singular noisy data points. In order to find suitable values for these parameters, a small synthetic dataset was used and afterwards a real-life dataset from the author moving around in Munich for 9 days was used. No in-depth parameter search was conducted, but the algorithms produces very precise results with the following parameter values:

$$r_{stop} = 25\text{m}, \quad r_{place} = 25\text{m}$$

For calculating the Home Stay two time quantities are used: The *total time* spent at home and the *total time spent at all places*, where the total duration spent at home can be derived from the Stops which were made at the home cluster. For computing the total amount of time spent, two approaches can be used: The simplest approach is using the time elapsed since midnight at the time of calculating the features. The other slightly more precise approach is to use the departure time of the last known Stop. The first approach will not take into account that the user, at the time of feature computation is likely still gathering data, which is not yet compressed into a stop. Since the data is not compressed into a stop yet, the data will therefore not count however the time which it took to gather this data will be used to calculate the total time. The more precise approach uses the timestamp of the departure of the last known stop, which is a tighter estimate. It can be done either way and will likely produce similar results unless an extreme edge case is considered where a Stop has not been found for a very long time due to the user transporting themselves. Here, the results would differ quite a lot but would also tell two different stories where one might be more correct than the other, depending on the meaning to be derived from the feature.

We define a routine as the repeating of a pattern - in this case in terms of the places visited at a certain time of the day. This includes where how much time is spent at home and when. However, most people will go on vacation during the year, which means the place where they sleep changes. In general, 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 not unlikely to be very different. However just because someone changes their routine over time, does not mean they don't currently possess one. Therefore it was chosen to base the *Routine Index* on the last 4 weeks (28 days) of data in order to base the routine overlap on more recent days. An issue which 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 since it means the *Routine Index* cannot exceed $\frac{2}{3}$ due to a third of the day's total hours being spent at a different place than usual. To add to this, 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 interesting to investigate whether or not comparing Mondays to Mondays and vice versa for every day in the week would yield more accurate results. In addition, the Routine Index cannot rely on a full day of data if it is to be calculated in real-time. To make the feature represent something meaningful given 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. Because of this, the *Routine Index* may be high early in the day since people usually sleep the same place, but are open to deviating as the day progresses. This variance in real-time can be useful to an application programmer in a recommender-system setting, where a trigger based on the *Routine Index* can be set, such that the user is alerted when the value falls below a certain threshold.

In order to calculate the *Routine Index* we need to have access to the data from previous days. Essentially, the Routine Index can be computed from the Stops only, since a set of Stops cluster into a set of Places, after which the Hour Matrix, that is used for the *Routine Index* computation, can be derived. At its core data needs to be stored on the device such that it may be retrieved in the future and there are two approaches to do this, either by saving all the Stops for today, or compute the Hour Matrix and save this instead, since it will take up less space. In a field study (described in chapter **??**), the author was found to have had just around 20 stops per day which amounts to under 600 Stops for for a 4 week period. This is a manageable number of data points to cluster with DBSCAN, which would be the only bottleneck in this approach.

The second approach involves keeping storing the derived Hour Matrix for each day instead. However for this approach, the *Places* would also need to be saved such that the coordinates of the Places found in the future could be compared (wrt. distance). This is to ensure that the columns of the different Hour Matrices refer to the same *Places*. This approach has the potential to be computationally cheaper but is more complex to implement and was therefore not chosen in this iteration.

Here, an overview of the algorithms used by the *Mobility Features Package* will be provided. The overview will not discuss implementation details but will provide precise definitions for how these are computed and the considerations which were made. Most of the features were simple to implement with support for real-time computation and were just a matter of performing arithmetic with regards to distance and time spent and places, however, the ones which need some explanation are outlined in this section.

A period is a set of several dates is defined as $D = \{d_1, d_2, ..., d_{|D|}\}$ with $|D| \leq 28$ and the date following the period being d_t. This can also be translated as D are the

historical dates to the date d_t.

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* is defined as $\delta(x_a, x_b) = \delta(l_a, l_b)$ and δ is the *Haversine* distance function.

Finding *Stops* is done by traversing every *Location Samples* in temporal order, i.e. the timestamp is used. The *Stops* for a given date is found by clustering Location Samples on that date based on time and distance. The set of *Stops* found for the period D is defined as

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

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 the following procedure:

(1) Let X be the set of time ordered data points

(2) Let $S \leftarrow \{\}$

(3) If there is at least one data point x' left in X:

  (I) Let stop $s \leftarrow \{x'\}$

  (II) For every remaining point $x \in X$:

    (i) Let $X \leftarrow X - x$

    (ii) If p is close to the median location of s: $s \leftarrow s \cup \{x\}$

    (iii) Else: $S \leftarrow S \cup \{s\}$ and go to (3)

(4) Return S

Figure 3.2: Procedure for finding *Stops*.

Places can now be found by applying the *DBSCAN* algorithm to the *Stops* found. It is important to note that if the *Routine Index* for a given date over a given period is

to be evaluated later on, all *Stops* found for this period should be used. This means all *Stops* from previous dates need to be stored on the device.
The set of Places found for the period D is defined as

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

Places are found using the following procedure:

(1) LET S be the set of Stops.

(2) LET L be the cluster labels found by DBSCAN where s_i has label l_i

(3) GROUP each stop $s \in S$ by its label, and let $S' = \{s_i \mid l_i \geq 0\},\ |S'| = N$

(4) LET $p_i = S'_i$ where each stop $s \in S'_i$ has the label l_i

(5) RETURN $P = \{p_i : i = 0, ..., N\}$

Figure 3.3: Procedure for finding *Places*.

Moves for a given can be calculated using the *Stops*- and the *Location Samples* from that date. The *Moves* are found by going through each *Stop* and calculating the distance between the current *Stop* and the following *Stop* by going through all the *Location Samples* which were sampled in the time interval between these the *Stops*. These points form the path which was taken between the two *Stops* and the path is used to calculate the exact distance traveled.
A set of *Moves* is defined as

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

is a set of time-ordered Location Samples. Moves are found using the following procedure:

This matrix is made 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.

(1) Let S be the set of stops and X be the set of time-ordered data points

(2) Let $M = \{\}$

(3) For each stop $s_i \in S$:

  (I) Let X_i be the path of data points sampled between s_i and s_{i+1}.

  (II) Let $d_i = \sum_{x_j \in X_i} \delta(x_j, x_{j+1})$

  (III) Let $m_i = (s_i, s_{i+1}, d_i)$

  (IV) Let $M = M \cup \{m_i\}$

(4) Return M

Figure 3.4: Procedure for finding *Moves*.

- An entry represents the portion of the given hour-slot that was spent at a given *Place*.
- Each row can maximally sum to 1.

Formally, given a period for which the number of *Places* is given as N the *Hour Matrix* H for a given day d is defined as:

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

Given an array of *Hour Matrices* for a period with D days, the *Mean Hour Matrix* for the period is defined as the average entry for each *Hour Matrix* in the period which are indexed with the date d_k:

$$\mathsf{H}^{\mu}(D)_{i,j} = \sum_{k=1}^{|D|} \frac{1}{|D|} \mathsf{H}(d_k)_{i,j}$$

Given a Stop s with arrival T_{arr} and departure T_{dep} the hour matrix as follows:
Let $i = hour(T_{arr})$, $j = hour(T_{dep})$ and $\Delta_T(\cdot)$ is the function for calculating the duration in hours.

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

.
If $i = j$ then $T = \Delta_T(T_{dep} - T_{arr})$, otherwise the following algorithm is applied:

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

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

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

The Home Stay feature indicates the percentage of time spent at the Home cluster, out of all the time of the day. Firstly a definition for Home needs to be clear; in the literature it is defined as the cluster which the user on average spends the most time at between 00:00 and 06:00 over a period of time [**Saeb2015**; **Canzian2015**]. However since the *Home Place,* may change from day to day, it was decided to let *Home* for a specific day be defined as the cluster for which the most time was spent during 00:00 and 06:00 on that day only. Formally, the *Home Place* $p_h(d_t)$ for today d_t is the place p_h where the following holds:

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

However in the literature [**Saeb2015**; **Canzian2015**] it is not stated how this would be calculated for an incomplete day. A choice was made for this to be calculated using the sum of durations for all *Stops* belonging to today's *Home Cluster*, divided by the time elapsed since midnight. The duration *Stop* s will be denoted $\Delta T(s)$ and similar the duration spent at a place p is defined as $\Delta T(p)$

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

Where $T_{now} - T_0$ is the time elapsed since 00:00:00.

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.

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

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

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

where M refers to the moves on date d_t.

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.

$$N = |K|$$

Defined by [**Saeb2015**], the Location Variance is defined as the natural logarithm to the variance of the latitude and longitude coordinates summed together:

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

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

Within the field of Information Theory, entropy is described in [**information-theory**] 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 $H(X)$ of the set of outcomes $X = \{x_1, x_2, ..., x_n\}$ is defined as

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

.
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

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

We define the Normalized Entropy as the Entropy H divided by the maximum possible entropy H_{max}:

$$H_N(X) = \frac{H(X)}{H_{max}(X)} \in [0, 1]$$

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.

$$H(P) = -\sum_{i=1}^{n} Pr(p_i) \log Pr(p_i)$$

and

$$H_N(P) = \frac{H(P)}{H_{max}(P)} \in [0, 1]$$

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 such 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.

The features described in the literature by [**Saeb2015**] which can be found in Section **??**. The time distribution for a day is defined by spending a duration of time at a certain space within a certain time-slot. However the *Routine Index* [**Saeb2015**; **Canzian2015**] was much more demanding to implement and it will, therefore, be a feature that is highly discussed in this thesis. To recap, the *Routine Index* describes how similar the place-time distribution of a given day is, compared to previous days for some period of days. A concrete 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. However the implementation by [**Canzian2015**] was very complex and therefore a much more simple version was chosen for the first iteration of the software package. We define the *Routine Index* as a similarity measure with a value between 0 and 1, where a low value indicates little overlap and a high value indicates a high degree of overlap. By representing each day with an Hour Matrix the similarity function can be defined. An illustration of the Hour Matrix for two different days is shown in Figure **??**, from which a Routine Index can be computed.
For the overlap function we define a union operator $A \cap B$ defining the *overlap* of two matrices A and B as

$$A \cap B = \sum_{i=1}^{24} \sum_{j=1}^{N} \min(A_{ij}, B_{ij}) \mid A_{ij} \geq 0, B_{ij} \geq 0 \tag{3.1}$$

The *Routine Index* for today d_t given the historical dates D, is defined as:

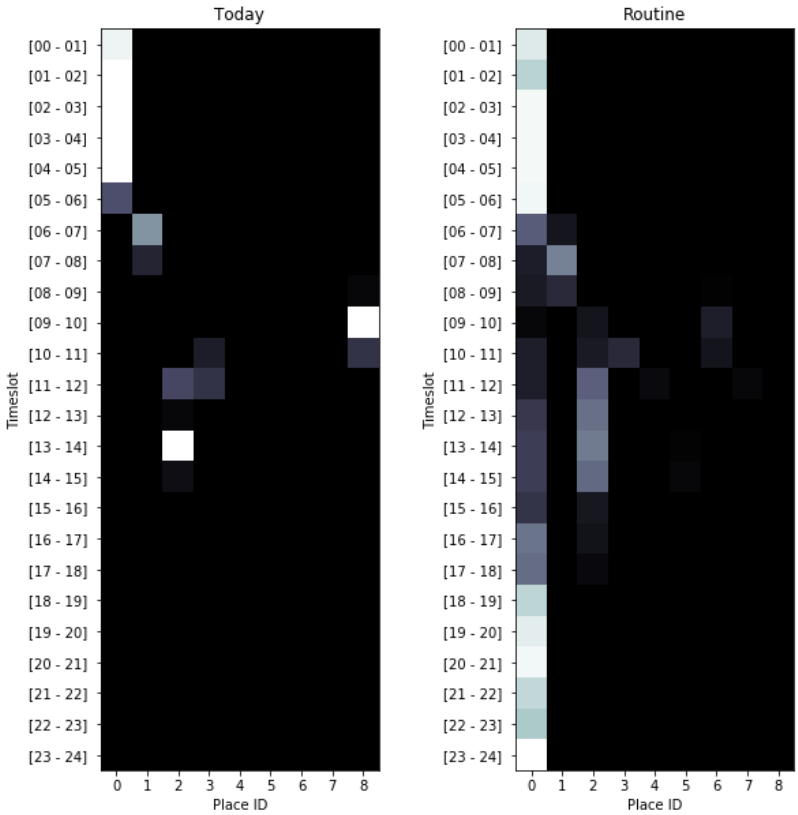


Figure 3.5: An illustration of the Hour Matrix for the a given day and the 'average day', i.e. the Routine.

$$r(d_t, D) = \frac{\sum(\mathsf{H}(d_t) \cap \mathsf{H}^{\mu}(D))}{\min\left(\sum \mathsf{H}(d_t), \sum \mathsf{H}^{\mu}(D)\right)}$$

The numerator

$$\sum(\mathsf{H}(d_t) \cap \mathsf{H}^{\mu}(D))$$

defines the *actual overlap* as the sum of the overlapping entries between today's data and the historical data. The denominator

$$\min\left(\sum \mathsf{H}(d_t), \sum \mathsf{H}^{\mu}(D)\right)$$

is the smallest sum of either matrix, whichever is smaller, and defines the *maximum potential overlap* between the two matrices. If one matrix is very sparse then the potential overlap is very low, and vice versa. If the *actual overlap* and the *maximum potential overlap* are the same, it means the matrices are the same and the Routine Index will have a value of 1.

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.

The functional requirements the package should meet, and how these are met will be described in this section. Fundamentally the package should support the computation of the features described in **??** however there are several layers to achieving this, including the following:

- Saving and loading of Location Samples on the device
- Computing intermediate features from Location Samples
- Saving and loading Stops and Moves on the device
- Computing the features

Saving and loading of Location Samples is necessary in order to not lose data. When an application tracks location data for a pro-longed period of time, i.e. a whole day, it will risky to keep all the collected data in RAM, since all data is lost if the app is killed by accident either by the OS or the user. For computing the Routine Index it was necessary to store historical data, in order to compare days. Intermediate features made this possible by bringing the storage requirements down significantly, compared to storing raw Location Samples on the device. A normal day of tracking could result in up to 18,000 Location Samples which is quite a lot of data considering the algorithms need up to 28 days of data. In addition, converting a dataset of raw Location Samples into Stops also made the computation for finding Places, and thereby many of the mobility features, much cheaper. Lastly, computing mobility features should be possible at any time, even if the day is incomplete. This has been taken care of by the definitions made in Chapter **??** by redefining features such that they can be evaluated on incomplete days.

Since historical data is a major factor in computing the features, it was decided to include an easy way for the programmer to store and load Location Samples. To store objects in an Object Oriented Language serialization [1] can be used, which is act of transforming an object into a graph of smaller objects in a data format which can be written to a file. A serialized object, in contrast to an entity stored in database is already pre-assembled. In a database this assembly happens via joins since the object is spread over multiple tables. The database may store the object more efficiently, but it does not come pre-assembled. This is analogous to how a set of Legos blocks comes in a box rather than already being pre-assembled. While traditional databases usually scale better than serialization they come with an overhead cost of being time consuming to set up properly. For this reason serialization was chosen in favor of databases, since the project was small. In addition it is also very easy to import serialized objects into other programming languages, such as Python for data analysis.

The data format used was JSON (JavaScript Object Notation) which is a very common, human-readable data format that uses key-value pairs to store data. A JSON object can be stored in a database, or it can be transformed to a string in order to be stored. It was chosen to simply transform JSON objects to a string and write them to a local file, rather than storing the information in a database on the phone. JSON supports a limited number of simple data types, such as strings, numbers, booleans, arrays, objects and null values. This means that in order to translate a runtime object to JSON, all of the object's data must be serializable. Serialization therefore requires components which are to be serialized to be converted into some representation which is purely consisting of these simpler data types.

The road from starting with a blank slate to computing mobility is depicted in the flowchart in Figure **??**. First the programmer needs to initialize data collection and save the collected samples with some frequency. Exactly how this is done is left the programmer. When feature computation is requested, the stored data including Location Samples, Stops and Moves are loaded into memory. The Mobility Context for today is computed from this data, and afterwards it is checked whether or not prior contexts should be used. If so, then the historical Stops and Moves are used to generate these, and they are added to the Context of today. In any case, today's Context is returned to the method caller.

[1] `https://www.martinfowler.com/eaaCatalog/serializedLOB.html`

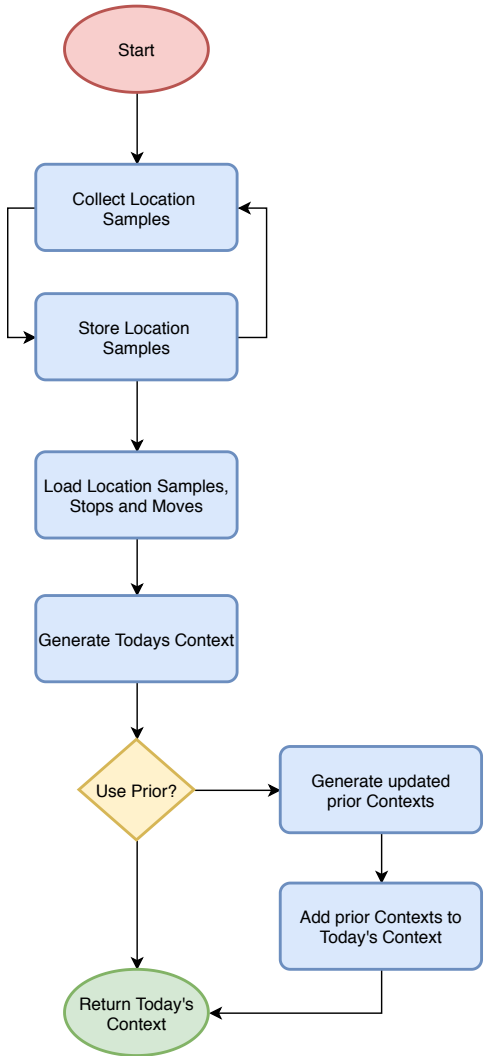


Figure 4.1: The flowchart for computing mobility features using historical data.

The domain model provides clarity and direction for a software system, even on a small scale such as in a library. In this section the design choices of the data model will be discussed.

In order to capture the data model in an object-oriented programming language, a UML diagram was maintained as the implementation went along in order to keep track of relationships between the classes.

A *GeoPosition* is defined by a geographical *latitude* and *longitude* and represents a 2D position on the Earth's surface.

A *Single Location Data Point* is a time-stamped *Location*. 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) [2] which is used to transfer GPS data from an arbitrary Location plugin to the *Mobility Features Package*.

An *Hour Matrix* is a matrix with 24 rows and columns equal to the number of places of some period. The *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.

A *Stop* is constructed from a centroid of a data point cluster (i.e. a Location) in addition to an arrival- and a departure timestamp, and a place ID indicating which place it belongs to.

A *Place* is constructed from a place ID, as well as a collection of *Stops* belonging to that *Place*.

[2] `https://martinfowler.com/eaaCatalog/dataTransferObject.html`

A *Move* is constructed from a pair of *Stops* as well as the set of *LocationSamples* which were sampled in between the two *Stops* which is the path the user took between the two *Stops*.

A *Mobility Context* is a collection of features which are derived from a set of intermediate features, where the *Stops* and *Moves* are from a specific date. The *Places* is derived from multiple dates for reasons which will be explained in the implementation details. In addition, a set of *Mobility Contexts* from previous dates can be provided as an optional parameter. A Mobility Context contains the mobility features, although the *Routine Index* is only available if an array of the set of *Mobility Contexts* was provided as a parameter, which is due to the feature depending on the data from previous days in order to compare them.

This interface will impose a getter-method for the GeoPosition of the class which implements it. This allows the Haversine distance to be calculated between objects of different types.

This interface will impose a serialization and de-serialization method for converting between a language object and a JSON object. The interface will allow the Mobility-Serializer component as previosly discussed to more easily implement serialization in a generic manner which is disucssed in Chapter **??**.

The general idea of the Mobility Feature Package is to provide an API that is easy to use for the application programmer, such that the features can be computed with as few lines of code as possible. This requires closing off the API to a large degree which has the upside of leaving very little possibility of error in the hands of the programmer. However, it is indeed a balancing act, in which swaying too much to one side can have consequences.

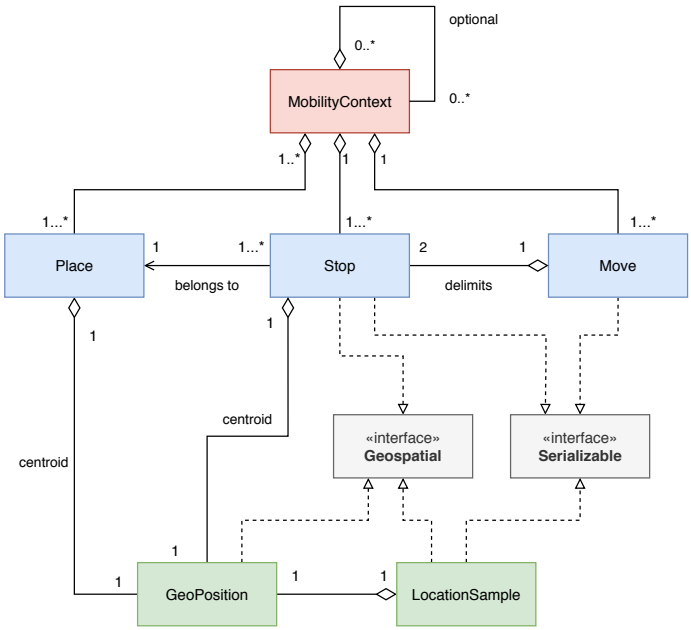


Figure 4.2: UML diagram for the classes used in the *Mobility Features Package*.

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 previous one is the amount of logic for managing historical data contained within the package. Previously most of this work was done by the application programmer 'outside the package' i.e. when writing a specific application. A major step in making the decision of moving logic inside the application was done after developing the study app discussed in chapter **??**. For this application the previous iteration was used, and it became glaringly apparent that too much work had to be done by the programmer. The final iteration is the one discussed in this thesis including the choices and lessons learned on the way of designing and developing it.

Designing an API is a balancing act of managing complexity. This means decisions have to be made in regards to what logic is kept internal and what should be kept external (i.e. left to the application programmer). As mentioned, location data is required to be collected to compute features which is done through a location service API. It is a possibility to move this data collection inside the package, but it has been

chosen not to do so, to ensure the long term maintainability and modularity of the package. The location service depends on the platform and programming framework of choice and leaving this particular choice to the programmer therefore makes sense. To take the middle road, it was chosen to include a data storage API in the package, which allows the programmer to easily store the collected data. As such, the only steps of the process left to the application programmer is to collect location data, store it on the device (through the Mobility Features Package API) and compute features when they are needed. Getting around using a custom location plugin can be done by converting between *Data Transfer Objects* (DTOs) [3] which in this case are simply objects holding location data, i.e. latitude, longitude and a timestamp. The DTOs considered are that of the programmers location plugin and that of the Mobility Features Package, namely *LocationSample*.

Some components of the package should be external, i.e. accessed and used by the application programmer and some should be kept internal. For this external/internal paradigm, class access is used in most object oriented programming languages. Class access refers to the access that the programmer has to a given class, i.e. whether it is private or public, or whether it can be instantiated. This can be applied to to ensure that the the programmer uses the package in the manner in which it was intended. If all classes were publicly available and could be instantiated, many edge cases arise in which the programmer may produce errors. In a worst case scenario this may lead to an app crashing in production where valuable data is lost as a result. This can even happen in spite of proper documentation and keeping certain components private is therefore very desirable. Figure **??** depicts the architecture of the Mobility Features Package, the type face indicates the access:
A bold typeface indicates the class can be instantiated by the user and is publicly available. An underlined typeface indicates the class is static and therefore cannot be instantiated but is publicly available to the programmer. A class with an italic typeface is a class which cannot be directly instantiated by the user but is available through the API.

Bold: Contents of this DTO are transferred to the equivalent DTO of the *Mobility Features Package*, namely *LocationSample* which is workaround for allowing the programmer to use any location service API. The programmer therefore has public access to the *LocationSample* class and can instantiate it.

Underlined: The sole static class in the diagram is the *ContextGenerator* class which is the main interface for the application programmer. This class contains a reference to the *MobilitySerializer* class that allows the user to serialize *Location Samples*, which the user can use to get an instance of the *MobilitySerializer*. This

[3] `https://martinfowler.com/eaaCatalog/dataTransferObject.html`

ContextGenerator component is also responsible for instantiating Mobility Context objects, which is done by loading Location Samples stored via the *MobilitySerializer* (as well as Stops and Moves if applicable) in order to compute the *MobilityContext*. Concretely, a *ContextGenerator* generates a *MobilityContext* which contains a series of features including *Stops*, *Places* and *Moves*.

Italic: Common for the *MobilityContext* as well as *Stops*, *Places* and *Moves* is that neither can directly be directly instantiated by the programmer, but they can be generated through the *ContextGenerator* component.

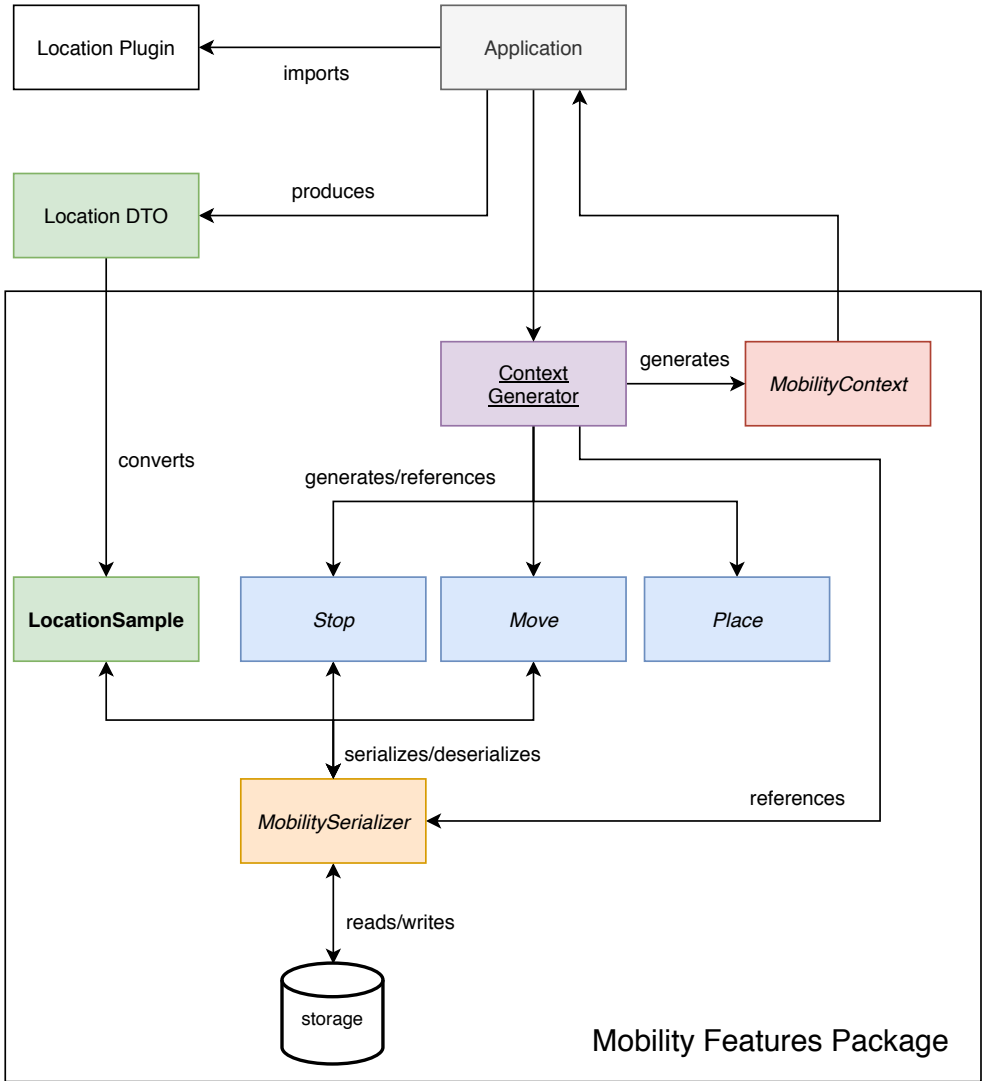


Figure 4.3: Component diagram for the Mobility Feature Package displaying the access of each component when implemented as a class.

This chapter will describe the implementation in of Flutter-specific terms which for the most part will be found in other programming languages. This includes how packages are built, how one goes about using packages, how testing is done and how packages are be published and how versioning works.

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 labour 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 which is hosted on the Dart package manager at `pub.dev`, the Mobility Features Package is hosted at `pub.dev/packages/mobility_features`.

It was chosen to implement the package in Flutter for several reasons: Firstly it is cross-platform, meaning the same code base can compile to both Android and iOS. For this reason one could also have chosen other frameworks such as React Native[1] developed by Facebook. The author was employed at CACHET who uses Flutter for mobile development. An example of a CACHET PhD project is [**mubs-rohani**]. In addition, the author had already authored many packages[2] for the CARP Mobile Sensing Framework by CACHET.

A flutter *package* is a library containing Dart pure code that enables the creation of modular code that can be shared easily. A package declares the package name, version, author, and so on (see Figure **??**, and secondly a source code directory named `lib`. It is relevant to develop a package when common functionality is to be shared among applications, and the functionality can be computed in the Dart

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

programming language alone. The *Mobility Features Package* is a Flutter package that contains a collection of algorithms that provides an application programmer with object-oriented abstractions that allows him/her to calculate relevant features for a mobile health application.
Whenever a functionality is wanted which is only available through a native API, such as the camera or battery level, a *plugin* is used instead. In contrast to a package, a plugin contains three codebases: Flutter (Dart), Android (Kotlin/Java), and iOS (Swift/Objective C). The Dart codebase contains an implementation which can be called from a Flutter app, and in turn calls the implementation in the Android environment and the iOS environment (whichever platform the device runs). It does so by transporting data between the platforms, which means no real computation is performed in the Dart environment; the Dart implementation simply invokes a method in the native environment, the native environment performs the computation, or data collection, and then sends back an answer. For transporting data between the native environment and the Dart environment, messaging channels are used, namely *MethodChannels* and *EventChannels*. A *MethodChannel* is used for communicating when data is to be transferred on a whenever a method is invoked. In contrast to this, the *EventChannel* allows streaming data from the native environment every time an event is triggered in the native environment, such as when a sensor picks up on a new data point. A Location API plugin which streams location data continuously uses an *EventChannel*.
This method invocation library is referred to as a *plugin* within the Flutter world, in contrast to a *package* which simply invokes other Dart code and as such contains no platform-specific source code. The Location API, available on both iOS and Android, will not be invoked directly from this package since that would require it to be a plugin. This has two main upsides: From the point of the application developer, it allows him/her to use their location plugin of choice (of which there are many [3] with specific parameters for how the location is tracked (ex frequency and distance). Secondly, from the perspective of the maintainer of this package, the package becomes much more modular and in turn easier to maintain.

The package contains two main directories and three metadata files as depicted in Figure **??**. 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

[3] `https://pub.dev/packages?q=location`

```
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 5.1: The file structure of the Mobility Features Flutter Package.

a the package 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 which allows an application developer to import a specific version of the package, for 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.

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 and.
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.

Figure 5.2: The pubspec.yaml file for the Mobility Features Package.

Classes and field which are private and still visible internally to other classes within the library. This helps making things communicate internally but have closed private access to the application programmer, for reasons discussed in Chapter **??**

Distributing a Flutter package is done via the Dart Package Manager, Pub. Pub is essentially a 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 is 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 **??** 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.

The package was implemented in Flutter according to the design in Chapter **??** 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 most objective oriented languages, such as Dart, the safest way to use fields in classes is to make them private, and to implement a parameter-less 'getter' method for retrieving the value, and a 'setter' method which takes in the new value as its parameter. In the Dart programming language, a field is declared private by having the the underscore prefix, i.e. `routineIndex` becomes `_routineIndex`, and the corresponding getter method is declared with `get` and is simply called the `routineIndex`: This results in an easy-to-read syntax when getting the value of the field, which looks like this:
The same concept can be applied to a constructor as well as the whole class. A public constructor is declared as:
With the private equivalent being:
A private constructor allows the class to be publicly *available* but not publicly *instantiable*. For classes the underscore prefix is used for the class name, to make it private, similar to field, i.e. `class HourMatrix` becomes `class _HourMatrix`.

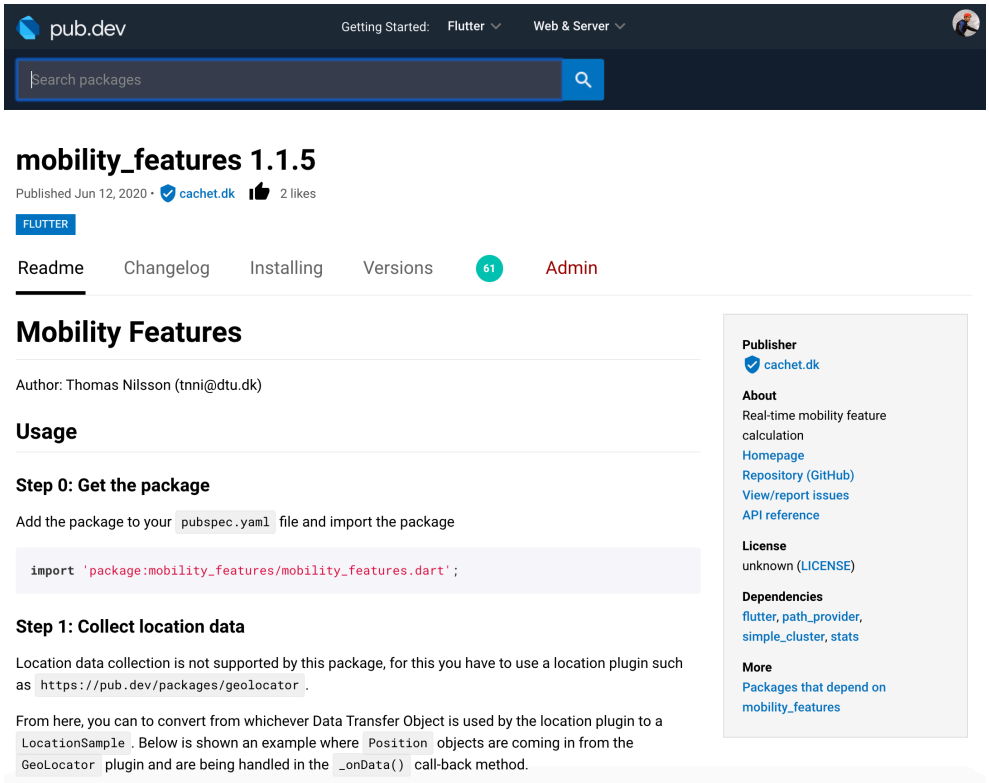


Figure 5.3: The page hosting the Mobility Features Package on www.pub.dev.

Figure 5.4: The API comments for the source code of the Location Sample class.

On the note of constructors, this package makes use of factory constructors which are effectively just methods which generate an object using the normal constructor. A factory constructor may be used to construct an object from JSON data, where each relevant field is extracted from the JSON data and passed onto the real constructor. A factory constructor is defined as follows:

All the components specified in the Domain Model Chapter **??** were implemented with their repsective relations to each other. As specified in the component diagram **??** the only component with a public-facing constructor was LocationSample, and by transitivity, also GeoPosition. This is done, as mentioned, to allow the user instantiate

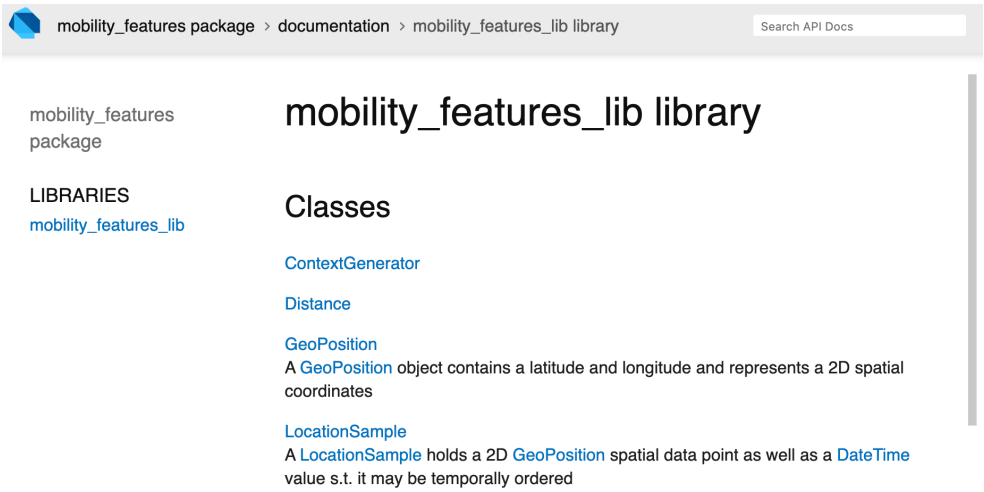


Figure 5.5: The auto generated documentation for the package, hosted on pub.dev, including the code snippet in Figure **??** for the Location Sample class.

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.
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 which implement the same interface. In Dart interfaces and abstract classes are one and 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.

The storing and loading of data, which includes Location Samples, Stops and Moves happen through the MobilitySerializer class. This class allows classes which 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 parameter and creates a runtime object of the given type, from the JSON object. The implemention of this method is left to the individual classes implementing the interface which is done by extracting data from the JSON object.
The MobilitySerializer class is a generic which allows the type `E` to be specified later, with `E` referring to either an 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.
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.
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.
Loading works in the reverse order, where the contents of the specified file is loaded into a string, the string is then split into elements using the delimiter token and each of these elements is turn 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.
Ideally, the switch statement could have been replaced by the following one-liner:
However, this relies on the language feature called reflection [4] which allows the compiler to infer the type of `E` at compile-time. However, Dart does not support *reflection* which makes this impossible.

Finding the intermediate features Stops, Moves and Places were done according to the algorithms described in Chapter **??**.

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.
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.

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

The Stop algorithm takes a List of LocationSamples as input, and uses two while-loops, and two pointers (*start* and *end*) which delimit a subset of the input data we are currently considering with the outer loop. Every time the outer loop moves, the *start* pointer is moved past the end pointer, in order 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. In addition, 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 and likely won't be necessary in the general case.

The distance calculation `Distance.fromGeospatial(centroid, data[end]) <= stopRadius)` 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 afterwards.

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 Locaiton Samples or a Distance, i.e. a double. The factory constructor called `_fromPath` calculates the distance of the path and then uses the normal constructor for create a Move.

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 Location Samples 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 however 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.

The mentioned 'fake' Stops is an addition to the definition in Chapter **??**. They are created to avoid situations in which tracking was started while moving, in this case no Moves are created before the user is stationary for some time, and Stops are 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.

The Place class only has one normal constructor which takes an ID (an integer) and a List of Stops.
The Place algorithm takes a set of Stops for a given period, i.e. Stops over multiple days. The DBSCAN algorithm [**density-based-1996**] 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, and a for each group a Place object is created with the group label and the Stops contained in the group. Lastly, the `placeId` attribute for each Stop in the group is set to the group label.

A Mobility Context object represents features for a given date. The class has private constructor which 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 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.
The other optional parameters is a List of Mobility Contexts is used for computing the Routine Index - how this is achieved will be explained later in this section. The 'derived' features are implemented as doubles (except for Number of Places which is an integer) and are fields in the Mobility Context class. All of these features are accessed via getters, which retrieve the value of the field.

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* [5], which is the idea of having a field be computed only the first time it is needed, and then stored from then on. 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

[5] `https://www.martinfowler.com/eaaCatalog/lazyLoad.html`

Figure 5.6: The getter method for a feature field.

can return the field's value. If the field is not *null* then the feature has already been computed, and can be returned immediately.

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 which 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 which tells which place and how much was visited.
Next, the `average()` factory constructor is discussed. This is a method for creating the Hour Matrix of 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 **??**. 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 severily limited. If either of the sums are zero then -1 is returned, due to either matrix beign 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 sclars is computed and added to the total overlap. The overlap for two scalar values we defined in Equation **??** as the minimum value of the two, given that both values are non-negative.

The derived features are computed according to their definitions in Chapter **??**, using the lazy evaluation template outlined in Figure **??**. The Home Stay feature is not 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

Figure 5.7: Lazy evaluation of a feature.

Figure 5.8: Construction of the Hour Matrix.

Figure 5.9: Construction of the Hour Matrix.

Figure 5.10: 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.

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 average Hour Matrix is computed. Lastly the Routine Index is found by computing the overlap between the two Hour Matrix of today, and the average Hour Matrix, using the `a.computeOverlap(b)` method of the Hour Matrix class.

The instantion of Mobility Contexts is done through 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 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 - they can however take parameters, which by nature are static.

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

Figure 5.11: The method for computing the Home Stay feature.

Figure 5.12: The method for computing the Routine Index feature.

system can be accessed by specifying a file name directly. This is a textbook example of hiding complexity from the application programmer.

Without a doubt, the most interesting part of the ContextGenerator class is the `generate()` method, which is where MobilityContext are 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.

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.

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. Also, in extreme cases, not recomputing Stops and Moves in this manner could lead to none being found at all throughout the day, because of how the dataset is segmented. 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.

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.

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.

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

Figure 5.13: Lazy evaluation of a feature.

This section will be a mirror of the official instructions on how to use the package, as of version 1.1.5.
Firstly, the programmer has to get the package by adding it as a dependency in their `pubspec.yaml` file of the Flutter project. Next once the dependency has been loaded it can be imported as follows:
Computing features can essentially be done in just 3 lines of code, excluding collecting Location Samples:
The exact method for arriving at this stage is outlined in the following 4 steps.

Location data collection is, as mentioned, not part of the Mobility Features package, and the programmer will therefore have tto 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 in the `_onData()` call-back method.

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. Firstly, the `MobilitySerializer` must be instantiated.
Next, given that the programmer has a collected the samples in a list `List<LocationSample> locat`
the samples can be serialized using the `save()` method of the MobilitySerializer.
Ideally, saving the data is done with a certain interval, such as every time 100 samples are collected.

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

Figure 5.14: Caption.

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`.

By default, a *MobilityContext* object uses the current date as 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.

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

Unit testing [6], 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 [7] in which the tests are written first, and the corresponding source code which should pass the test is written afterwards. 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.

[6] `https://martinfowler.com/bliki/UnitTest.html`
[7] `https://martinfowler.com/bliki/TestDrivenDevelopment.html`

There were a few shortcoming 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 which 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 which was partly explored was using a real-world dataset which the author gathered tracking himself however to verify the algorithms on this dataset it would need to manually labelled 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 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.

In this subsection selected unit tests will be exemplified.

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 was successful, the lengths of the original dataset, and the loaded dataset are compared.

Figure 5.15: A unit testing demonstrating storing and loading a small, synthetic dataset.

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

This test works similar 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 meaking 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.

Figure 5.16: Unit test for a single Stop.

Figure 5.17: Unit test for a single Stop.

To validate the Mobility Features Package in a real world, 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.

A small-scale study was conducted which ran for 3 weeks and included 10 participants (including the author). In the study the participants used the application discussed in Chapter **??** that collected their location data and computed mobility features daily. This section will discussed the overall method with which the study was conducted.

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 **??** 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 the 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.

In addition to this the application also had a diary consisting of 4 questions that the user 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 8PM 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 Features which were *Number of Clusters*, *Home Stay* and *Routine Index*. The point of the questions were to get a subjective estimates of the values of these features. It is important to stress the fact that these these answers are estimates since the user

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.

cannot be expected to give very precise answers and secondly they are very subjective since the definition of things such as a 'Place' may vary a lot from person to person.

Answers were collected through a diary in order to evaluate, to choose the most important parameters certain features were evaluated through comparing subjective answers with calculated features. The features had to be ease to formulate as a question such that subjective user answers could be collected, as such features such as entropy and location variance were ill-suited, whereas Home Stay, Number of Places and Routine Index were chosen instead. The questions the user was asked 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')

Where question #3 and #4 relate the Routine Index feature.
Collecting the subjective *number of places* visisted, was done by asking the participant exactly that, making it the easiest of the 3 features to evaluate. For collecting the subjective *Home Stay* percentage, the user 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 user 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 0 to 10, however this put too much on the user, the main information we wished to draw from the user was a very high level overview of whether the day today was a lot like the previous days. 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.

The Corona virus pandemic lead 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 major implications for the study and meant that it would be expected that the participants routine was quite steady, since they were mostly home, and that the home stay percentage was very high and that the number of places visited was very low. In addition it was probably also not common for most people 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.

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

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 in its essence functions a separate App Store for applications in development, which 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 helps the user be reminded to fill out a daily questionnaire. An installation manual (see Appendix **??** was provided to the participants to ensure the applications were set up correctly. The installation process is shown in Figure **??**.

The Main Page of the application merely displays a list of instructions to the user, but has two buttons in the navigation bar which can take the user to the Info Page and the Diary Page. 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 user

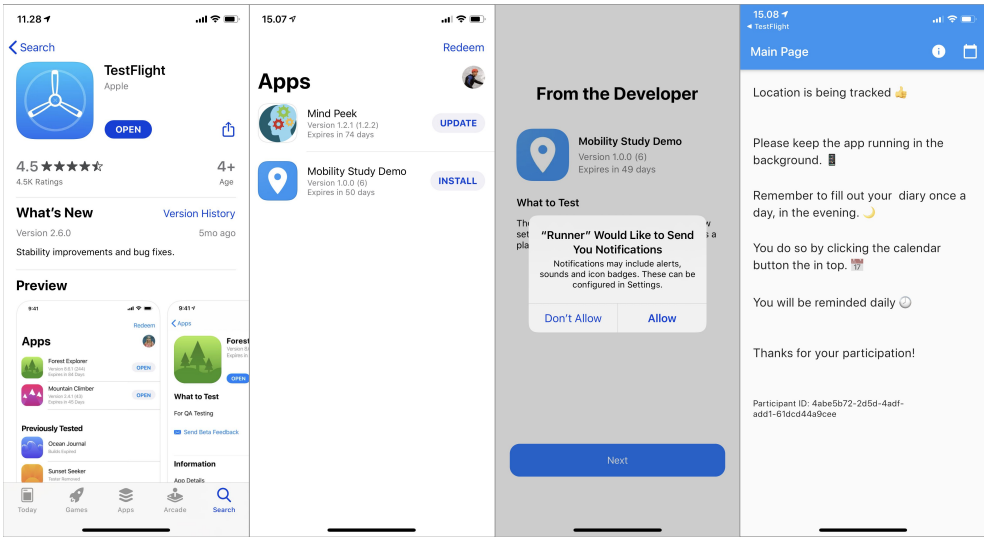


Figure 6.2: The initial installation- and setup process for the user.

navigates to the Diary Page by either tapping the calendar icon on the main page or by tapping the notification which comes in daily. In total there are four answers which answer answered by pressing the Answer button; this shows a wheel of possible values to pick from for providing an answer. Once all answers have been given, the submit button will be enabled and once pressed the answers will be stored on the device and sent to a server. When this is done, the last screen will appear which informs the user the answers have been saved and thanks them for their contribution.

An initial version of the application included a display of the real-time calculated features, which were re-calculated 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.

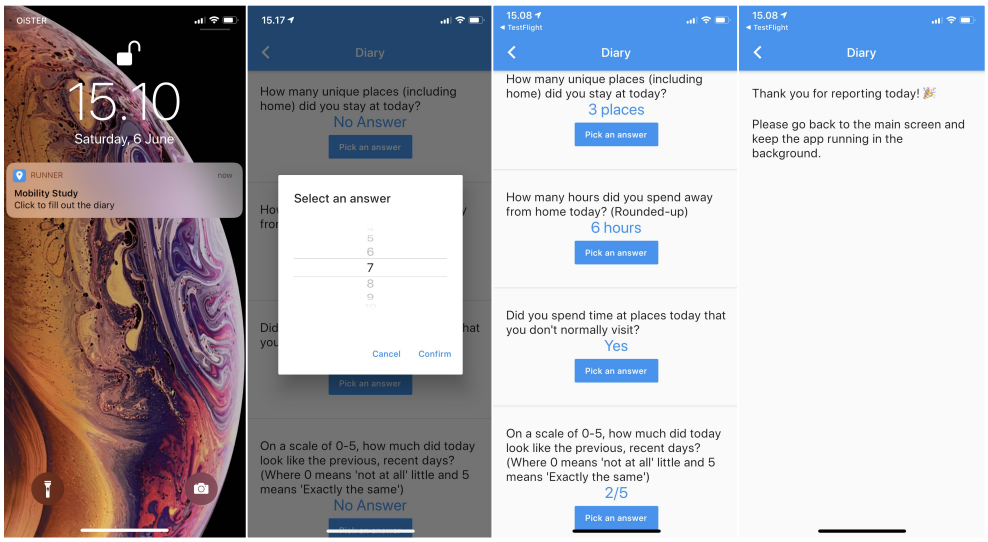


Figure 6.3: The different screens which the user is taken through for submitting answers.

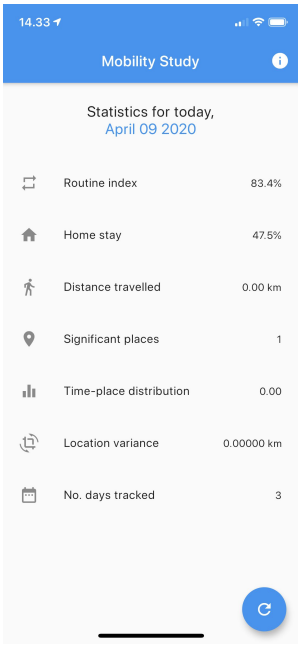


Figure 6.4: An early iteration of the study app in which the feature values were shown.

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 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 the period. 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.

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.

To provide a high level overview of the different components which make up the study application, a component diagram displayed in Figure **??**. 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 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.

For collecting Location Samples, a custom version of the *Geolocator*[1] 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 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:

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[2] and the features was 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.

[1] `https://pub.dev/packages/geolocator`
[2] `https://github.com/Baseflow/flutter-geolocator/issues/390`

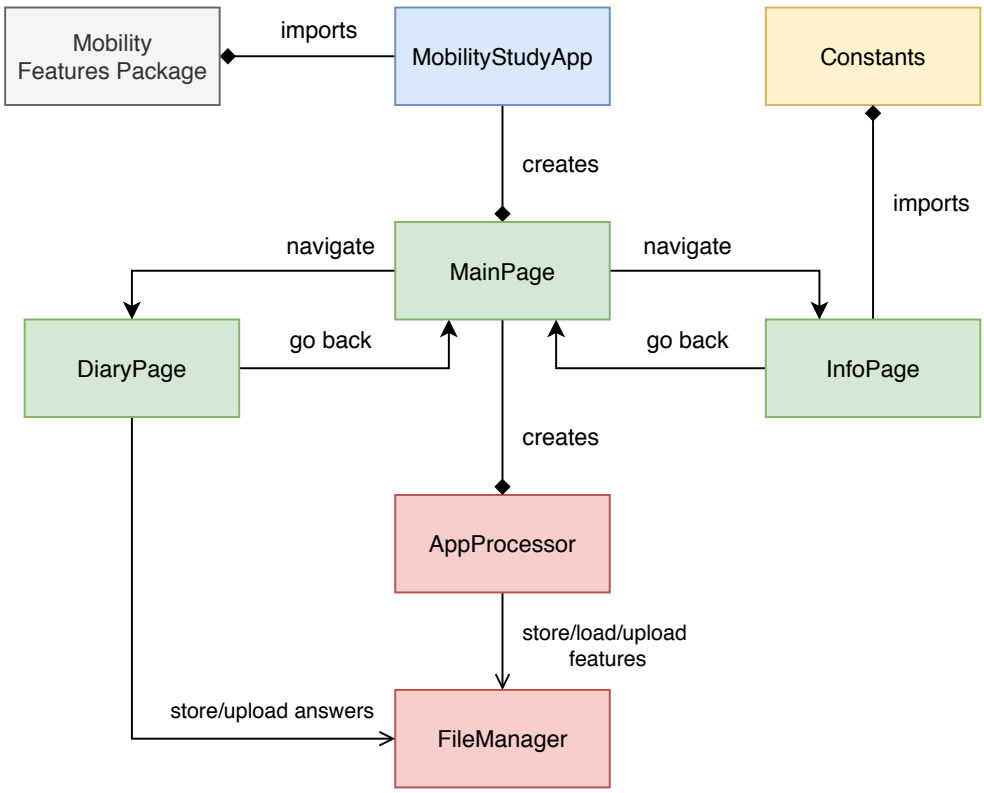


Figure 6.5: Component diagram for the study application displaying the different building blocks and the interactions between them.

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.

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 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`. Afterwards the buffer is emptied, and the process starts anew.

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

Firebase file storage was used to host all the data generated by the study application. Another possibility was to use a Firebase Real-time Database however given that the application already used files for storing data, it seemed most natural to continue using a file-based system. 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 is 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.

Every time the buffer has been 'spilled' to the disk 5 times, features are computed. This was done simple to ensure features were computed regularly in real-time (i.e. with an incomplete dataset) during the study, as is a completely arbitrary trigger. 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 a main thread runs the user interface, and background threads can be spawned in order to perform heavy computation which would otherwise 'clog' the main thread, which means the user will experience a 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 which 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.
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.

Lastly, the `_asyncComputation` method is static which is due to the computation being done in a separate thread than 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 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.

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

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.

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

Regarding data collection, figure **??** 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.
It was computed based on file size, the average size of a serialized *LocationSample* is 100 bytes, and a *Stop* is 139 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 **??**. The number of valid days for a specific feature is a day 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.

A thing to keep in mind is that the participants' answers are not ground truth, and there are multiple reasons why a participants 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. Secondly, some users misunderstood the routine scale and users whose data looked strange were contacted afterwards 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 participants were probably rarely able to give exact answers, according to the their own recollection.

Some users missed several days of filling out the diary which meant the computed features for that day could not be compared to a subjective measurement. Figure **??** shows the number of total days where a participant participated in the study where data was collected vs the days on which they provided an answer. For data analysis it was only possible to calculate the error between computed features and the answers given if the participant actually gave an answer. This mean that for some

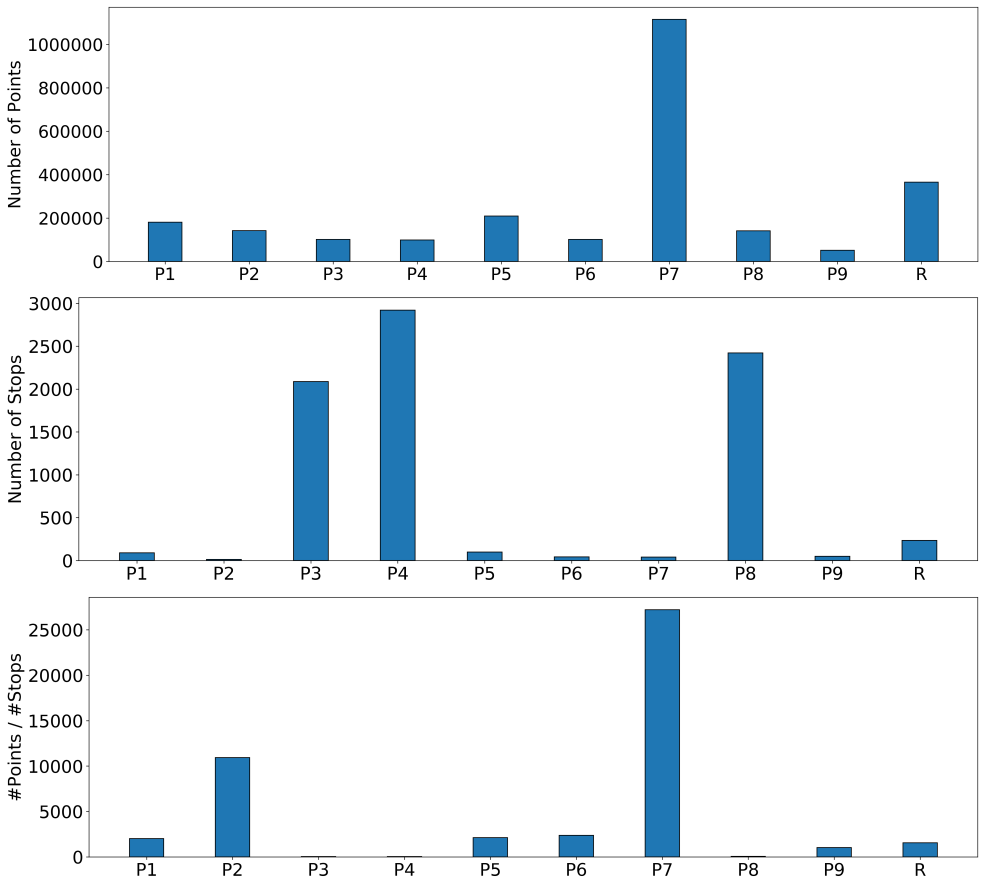


Figure 7.1: The number of raw Location Samples (top) and the number of Stops (middle) collected, and the ratio between the two, for each participant.

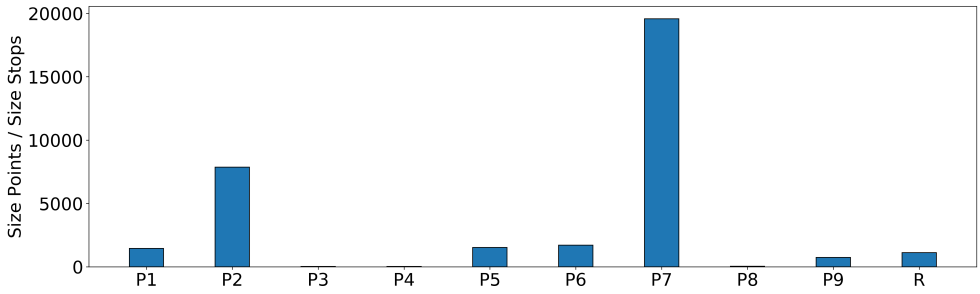


Figure 7.2: The ratio between Location Samples and Stops expressed in terms of storage, in MB.

participants, a lot of data was unfortunately not usable for the data analysis. As an example, it was considered whether or not to entirely get rid of P9 since he or she has very little data both terms of total days collected and even fewer in terms of days with answers.

Another problem which was encountered was when participants would fill out the questionnaire for day n during the night, which mean the resulting date was $n+1$, i.e. the wrong date. This was simple enough to fix by simply moving all answers given between 00:00 and 10:00 to the previous date which got rid of all the problematic data points. In addition some users entirely forgot certain days but remembered it the following day, and reported it manually to the researcher, and these data points were then added manually.

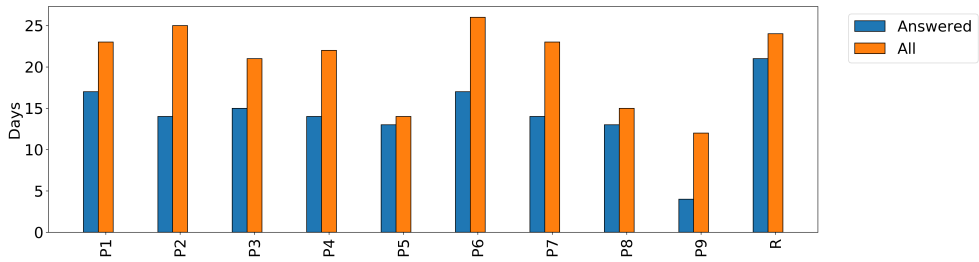


Figure 7.3: The total number of days of participation vs the days for which the diary was filled out by the participant. As can be seen, some users often forgot to give an answer.

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.

Since the Home Stay feature is calculated by using the *tracked* time at home, divided by the total time time midnight, it only takes a few minutes of missing data to make the home stay feature undershoot. It is therefore to be expected that this feature will lie somewhat lower than the answer given by the user, given the user did not round up excessively.

The day-by-day results for the author are displayed in figure **??** and from this plot the computed features have a high correspondence with the provided answers. However, the feature computation is based on the authors own definitions which means 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 **??**.

To say something about whether or not the algorithms undershoots or overshoots, the difference in sum was calculated as $ME = \frac{\sum(A)-\sum(F)}{N}$ (N being the total number of days for the specific feature) meaning that if the result is positive then the answered result was on average higher and vice versa if the result is negative. The alternative is to calculate the mean error one can also calculate the mean of the difference per observation $ME = \frac{1}{N}\sum_{i=0}^{N}(a_i - f_i)$ however this the downside of cancelling out certain observations when the sign differs. Figure **??** shows the mean error for each feature, for each participant. It can be observed that participants generally answer that they have been at more places than the algorithms calculate. The Home Stay feature generally lies 10% lower for half the participants (i.e. positive mean error), which was to be expected for reasons discussed earlier. For a couple of participants it overshoots (i.e. negative mean error) although still within 10% error. For participant 2 however it averages out to near zero percent and for participant P1 the feature undershoots dramatically with an error of 30%. The Routine Index feature is a mixed bag, with 6 participants answering higher than the algorithm and the remaining 4 being lower. All except for participant 6 deviating with almost 40%. An important detail here is that the Routine Index answer was given on scale with 20 percent increments which means that the mean error is likely to lie much higher percentage-wise than the other two features.

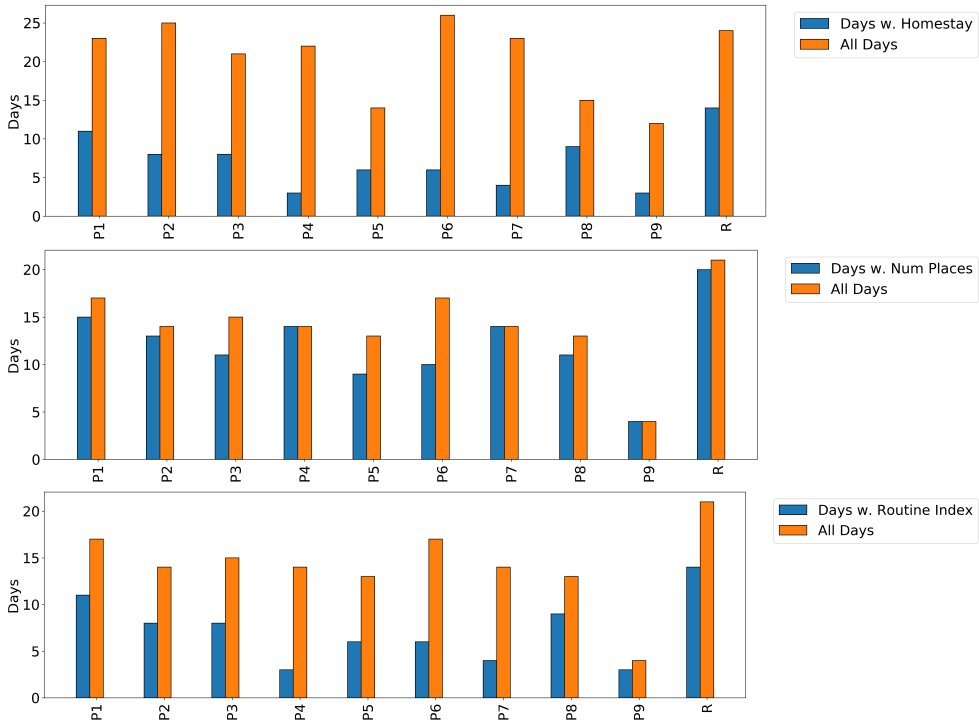


Figure 7.4: The days for which a given feature could be evaluated, out of the total days for which an answer was given and features were computed.

The application development process led to many of the improvements of the package API reflected in Chapter **??** and **??**, mostly pertaining to moving processing inside the package rather than outside, and giving the application developer more restricted access. However, a couple of issues still remain both in terms of the API and the feature accuracy.

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

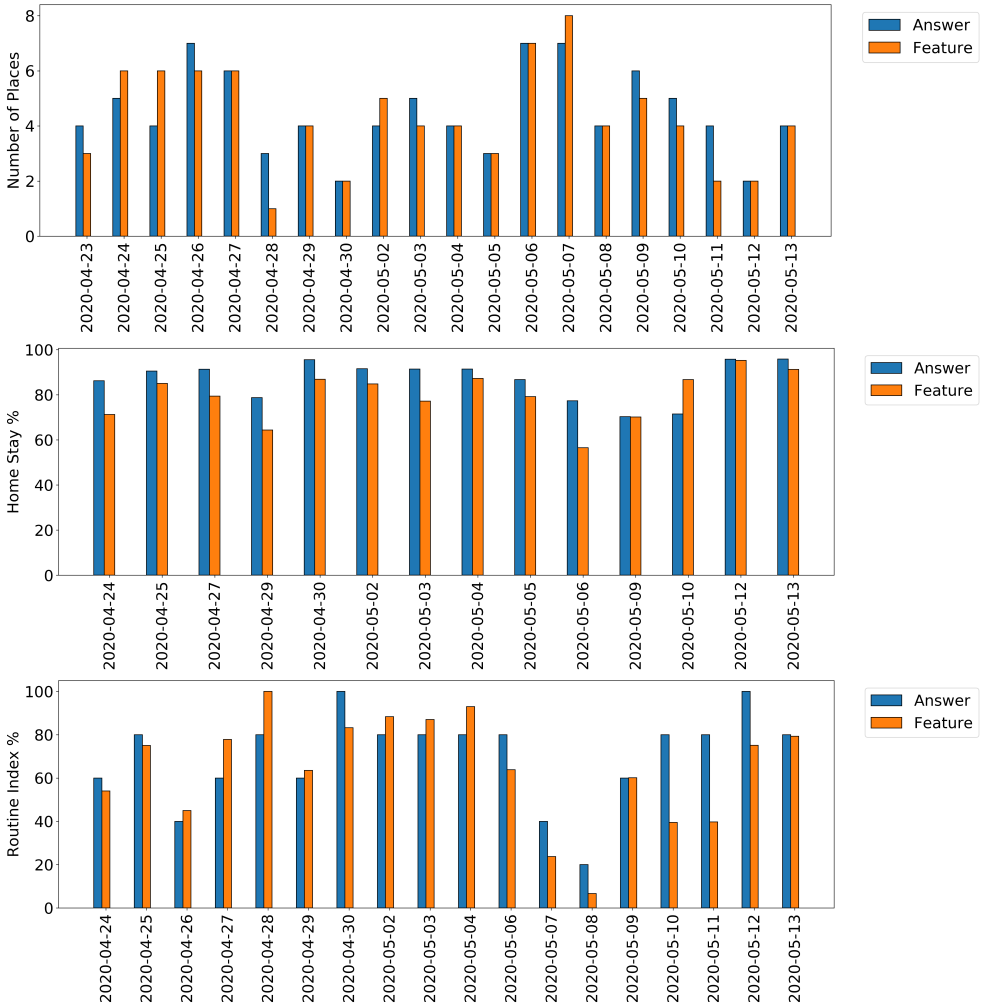


Figure 7.5: The answered and calculated data for each day, for the author.

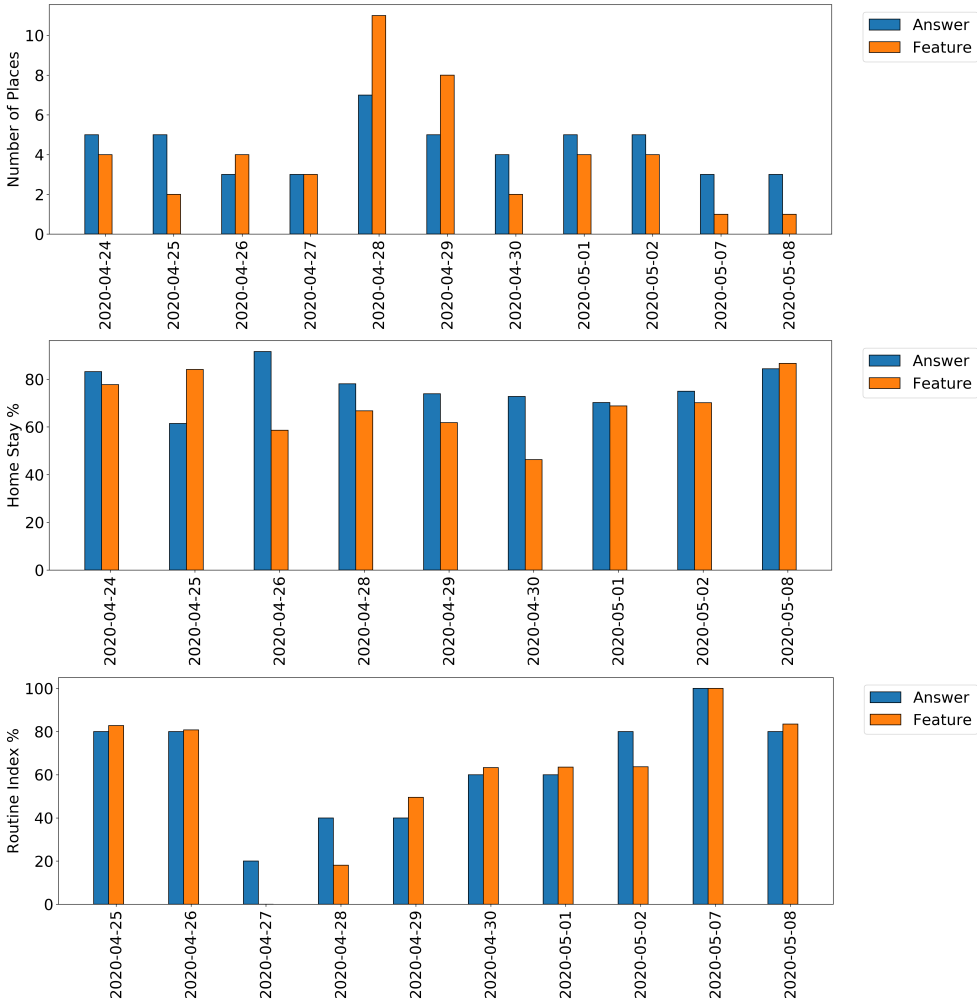


Figure 7.6: The answered and calculated data for each day for a participant P7.

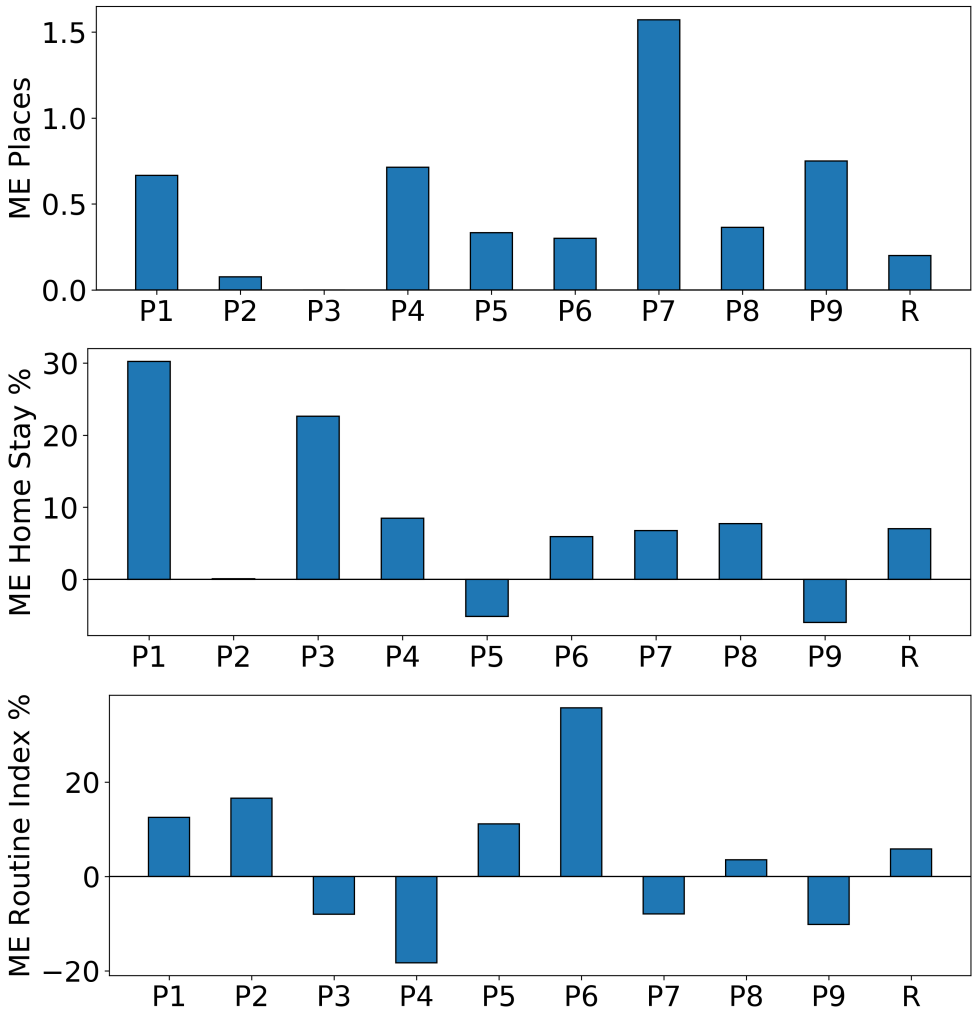


Figure 7.7: The number of data points in terms of days which could be evaluated, for each participant.

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. Not all of the days are annotated as previously discussed, far from it, however a researcher can go by the days manually one by one for each participant and label certain of the features such as places visited, and to some extent the Home Stay and Routine Index, although the latter two would require a great effort of fine-combing the the data-sets. If feature-labeling was performed then it would simply be an optimization task where the optimal parameters for Stops and Moves are the parameters and the error of accuracy the objective to minimize.

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 currently is to override the date, if 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 the following way: When Location Samples are loaded, group them by date, and compute Stops and Moves for each of these dates. Save the Stops and Moves, and then throw away the samples from these previous dates. out.

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 **??**. In the current version, only the intermediate Features will be computed in the background thread, whereas all the derived features contained in the MobilityContext object returned by the asynchronous computation will not yet have been computed upon returning from the background thread. This is due to *lazy evaluation*, which ensures that the features are not unnecessarily computed in advance, and only computed once they are requested. This could potentially mean freezing the UI thread, since the features computed via lazy evaluation will be computed once the Mobility Context object is inside the main thread. The fix for this is to evaluate all features in the constructor of the Mobility Context class, which will imply longer

guaranteed computation time for creating a Mobility Context object but will also guarantee that all features have been pre-computed in the background thread and will not block the UI thread.

The Pub package manager 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 **??** is a good candidate for an example app since it presents the calculated features to the user and can be implemented in a dynamic fashion were features are constantly recomputed and updated.

The package fits into the *CARP Mobile Sensing Framework* 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. This integration will be made in the weeks following the thesis such that it may be easily implemented into exisitng CACHET projects such as MUBS [**mubs-rohani**]. For the foreseeable future it will be maintained by the Author, who will continue his employment at CACHET as a research assistant.

When should the user be given recommendations
Trigger can be time interval or threshold for a feature, i.e. if homestay greater than 0.5 prompt the user to leave the house
Recommendations from depression database (Rohani, 300 items, not part of this thesis)

Calculate routine index from week to week rather than day to day. Currently we assume that all days ideally look identical, which for most people will not be the case. For many people the weekends differ quite a lot from their every day, and naturally the routine index will therefore be lower during the weekend, and weekend days will tend to 'wash out' the routine index calculation.

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

The features Number of Places, Home Stay, Entropy/Normalized Entropy, Location Variance and Routine Index were chosen based on the work by by Saeb. et al [**Saeb2015**] and Doryab et. al [**extraction-of-behavioural-features**]. 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 on a daily basis 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.

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 in which *Stops* from multiple days are used 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.

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.

In addition to the three main questions, the package was validated using unit testing which tested the accuracy of the algorithms for simple synthetic data sets. For validating the algorithms and the package in a broader sense, a field study with 10 participants was conducted. For this study, an application was developed in Flutter which used the package. The demo application was distributed digitally via Apple's *Testflight* app which allowed for quick patching of bugs with minimal installation trouble to the users. It was the development of the study app which lead to the final package design. The application used a version of the package which had a lower level of abstraction which made it much more cumbersome to use which made it apparent that the abstraction level needed to be much higher. The study produced a dataset of 240 MB of Location Samples spread over 10 participants with subjective answers. It would have been possible, if time allowed, to perform parameter tuning for the feature algorithms to improve their accuracy. This would likely require manual labeling of the data since the subjective data is somewhat unreliable in its nature which would be a considerable time sink.

CACHET Copenhagen Center for Health Technology

MDD Major Depressive Disorder

BA Behavioural Activation

Major Depressive Disorder (MDD)
Also known simply as depression, is a mental disorder characterized by low mood, low self-esteem, loss of interest in normally enjoyable activities, low energy, and pain without a clear cause.

Behavioural Activation (BA)
A therapeutic intervention that is often used to treat MDD. BA treatment focuses on keeping depression at bay through activities which produce positive reinforcement for the patient. BA is highly customizable and is a very personal treatment plan.

Mobility
How a person changes location over time.

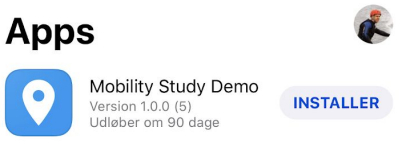
Mobility Feature
A number which describes one of the ways in which a person changes location over time. Many features will paint a more complete picture of the mobility patterns than fewer features.

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.

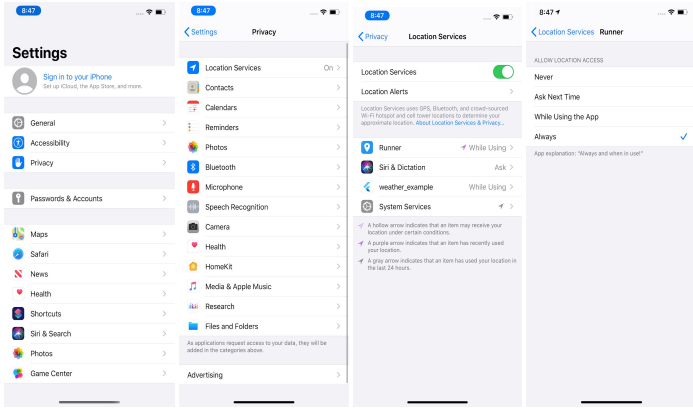
Step 2
Open the TestFlight app, and install the available app. This will install an app called *Runner*, on your device.



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.

Patented by Pfizer
The PHQ-9 questionnaire 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)
- More than half the days (2 points)

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

- 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)

{2.7182818284}ϑφερτυθιοπσδφγηξκλ