GatherMySteps

Personally relevant GPS track cleaning and analysis

Rui Pedro Lopes Gil

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Daniel Jorge Viegas Gonçalves

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Daniel Jorge Viegas Gonçalves
Member of the Committee: Prof. João Manuel Brisson Lopes

October 2016

Acknowledgments

To mom and dad.

Abstract

Smartphones have become ubiquitous and a part of ourselves. Since we carry them everywhere, they are the perfect device to record our memories, through photos, videos and through lifelogging. One specific form of lifelogging is geographic lifelogging, which consists in recording our own trajectories. Anyone with a GPS enabled device, can effortlessly record their own trajectories. Often they contain noise, introduced by the lack of accuracy of GPS devices, due to sensor or human error. Also, raw recordings do not contain any explicit personally relevant information, such as the relevant places that we moved between, the purpose of our visits or the transportation modes used in trajectories. Furthermore, the recordings that need to be processed in a geographic lifelogging context may range from a few, to tens, on a daily basis. We propose GatherMySteps, an application focused on simplifying the processing of GPS recordings — tracks, with an emphasis on usability and proactivity (suggesting the most likely corrections and annotations) to make the lifelogging process as efficient as possible. It provides simplification, editing and learning capabilities so that the user can correct and validate the tracks. Then, a fast and powerful semantic editor is used to annotate the locations and transportation modes of all the trajectories in a day. We evaluated our system using usability user tests and utility use cases. From those tests we conclude that our solution is efficient to process and annotate tracks, with a small amount of clicks and time and fills the needs of lifeloggers.

Keywords: GPS Location, Location aware, GPS Trajectory, Map matching, Commute analysis, Place recognition, GPS tracks, Route inference, User interface, Infer transportation mode, Recognize human behavior.

Resumo

Os smartphones tornaram-se dispositivos omnipresentes e uma parte de nós mesmos, isso faz deles o dispositivo perfeito para gravar as nossas memórias através de lifelogging. O lifelogging geográfico é uma forma específica de lifelogging, que consiste em gravar as nossas trajectórias, utilizando dispositivos como smartphones. Contudo, muitas dessas gravações contêm erros, causados pela falta de precisão dos dispositivos e devido à falta de dados. Para além disso, as gravações na sua forma original não contêm informação implícita pessoalmente relevante, tal como as localizações, qual o objectivo da nossa visita a um local, ou os meios de transporte que usamos para chegar lá. No contexto de lifelogging geográfico, também é preciso ter em conta a quantidade de gravações que são produzidas e que precisam de ser processadas; podendo haver desde duas gravações, a dezenas de gravações por dia. É neste modelo que propomos GatherMySteps, uma aplicação focada em simplificar o processamento de gravações GPS de trajectórias, com ênfase na usabilidade e proactividade (sugerindo as correções e anotações mais prováveis ao utilizador) para o processamento o mais eficiente possível. Esta ferramenta fornece ao utilizador funções de simplificação, edição e aprendizagem para corrigir e validar as gravações. O editor semântico é usado para anotar as localizações e meios de transporte de todas as trajectórias de um dia. Para avaliar o nosso sistema foram desenvolvidos e aplicados testes de utilizações e casos de estudo de utilidade. Através dos resultados podemos concluir que a nossa solução é eficiente para processar e anotar gravações de trajectórias.

Palavras-Chave: Localização GPS, Location aware, Trajectória GPS, Map matching, Análise de viagens, Reconhecimento de localizações, GPS tracks, Inferência de trajectos, Interface do utilizador, Inferência de meio de transporte, Reconhecimento de comportamento humano.

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 17, 37, 38, 79

AST Abstract Syntax Tree. 53

CSS Cascading Style Sheets. 42

CSV Comma-separated values. 22

EPCGI'2016 Encontro Português de Computação Gráfica e Interação 2016. 3

GPS Global Positioning System. vii, xiii, 1–3, 5–8, 10–31, 34, 37, 41–43, 50, 54, 56, 67, 69, 75, 76, 79, 80

GPX GPS Exchange Format. 22–24, 37, 43, 54, 68

HMM Hidden Markov Model. 14, 15, 20, 25

HTML HyperText Markup Language. 42, 44, 52

ID3 Iterative Dichotomiser 3. 34

IDE Integrated Development Environment. 51, 54

IQR Interquartile range. 71

JS JavaScript. 42, 44, 52, 80

JSON JavaScript Object Notation. 37, 38

KML Keyhole Markup Language. 22

LBC Leader-Based Clustering. 9, 10, 31

LMT Logistical Model Trees. 11

MBR Minimum Bounding Rectangles. 36

MLP Multilayers Perceptrons. 11

MSC Mean Shift Clustering. 9

OSM OpenStreetMaps. 25, 44, 46

PEG Parsing Expression Grammar. 53

PELT Pruned Exact Linear Time. 32, 33

RDP Ramer–Douglas–Peucker. 15, 30, 36

REST Representational State Transfer. 17, 37, 79

SDG Stochastic Gradient Descent. 33

SPT Spatiotemporal Trajectory Compression Algorithm. 30

SQL Structured Query Language. 38

SVM Support Vector Machine. 33

TDSP Top-Down Speed-Based Trajectory Compression Algorithm. 30

TDTR Top-Down Time-Ratio Trajectory Compression Algorithm. 30

THD Trajectory-Hausdorff Distance. 35

THR Trajectory-Hausdorff Ratio. 35, 36

TSL Timestamped Location. 18

UI User Interface. xiii, 2, 3, 7, 13, 37, 41–45, 55, 67, 75, 76, 79, 80

UML Unified Modeling Language. xiii, 27, 45

XML Extensible Markup Language. 22

Chapter 1

Introduction

With the rise of smartphones in the consumer space, a previously nonexistent computer power became available. In a short amount of time, a device that everyone could carry everywhere inside their pockets, was being used by a lot of people. What once was a device to make phone calls and exchange text messages became a small computer, capable of running complex applications, sensing the world through GPS, accelerometers, and other sensors. So much so, that in a decade, they became ubiquitous and a marquee of usefulness and functionality, with ever increasing computer power.

These devices go with us everywhere. They stay with us in good and bad times, they are with us when we go running, on holidays, or commuting to work. *They are an extension of ourselves*.

People soon started to explore what data they could collect about themselves, about their habits and their activities using this small computer that follows us everywhere. Simply by installing an app, anyone can now record a myriad of personally relevant data. From the amount of calls one makes, their duration, to the time we spend looking at the screen, or even the places where people have been to.

With this new data at our disposal, we can potentially fill some memory gaps and answer difficult questions about our past. *Where have you been the first day of last month?* or *How did I get to this place?* are some of those questions that even after a small amount of time, are intrinsically difficult to answer. Lifelogging provides answers to some of those questions.

Lifelogging is the act of recording data about one's life. It aims to provide a backdoor entrance to forgotten memories, by later analyzing the data collected. Thriving lifelogging communities, such as Quantified Self[1], are dedicated to this goal. People are tracking the steps they take everyday[2], the food they consume[3], the time they spent at the computer, their sleep quality[4], or even more exotic data such as one's microbiome[5].

The collection of data, however, is a means to an end. The ultimate goal of lifelogging is to analyze it, identifying life patterns to improve performance, health, or to keep a detailed log about our past selves. The latter is the catalyst for this thesis: to see where users have been, to where they have gone, and how they have arrived there; using data collected using a GPS device, such as a smartphone.

If data collection is a simple process, its processing and analysis is not. There is little incentive for people to record information about themselves, because there is a dearth of tools to extract truly meaningful information of that data.

[1] `http://quantifiedself.com`, last accessed on October 15th, 2016

[2] `https://medium.com/@dodeca/step-by-step-9817feb08850`, last accessed on October 15th, 2016

[3] `http://blogs.sas.com/content/jmp/2015/02/20/cleaning-up-and-visualizing-my-food-log-data-with-jmp-12/`, last accessed on October 15th, 2016

[4] `https://www.interworks.com/blog/rrouse/2015/09/14/visualizing-data-my-sleep-tableau`, last accessed on October 15th, 2016

[5] `http://www.ubiomeblog.com/bacterial-functions-diversity-score-and-a-halloween-treat/`, last accessed on October 15th, 2016

Some applications exist that are capable of inferring places meaningful to the users, such as their home and workplace, can infer the transportation method used on a recorded trajectory (track) or efficiently display this information.

Alas, most of those solutions do not focus on what is personally relevant to users. They do not take into account user input regarding particular meanings of the information and try to answer all the questions that they purpose in an automatic way, using generic techniques that do not adapt to their users directly. To extract personally meaningful information, one should have into account differences between dealing with data generated directly by ourselves and between dealing with data generated by taxis, birds or by different kinds people that do not reflect our way of living. If, besides the raw sensor data, the user also provides input, the information extracted that could be much more meaningful to the user and less error prone over time, as it learns user patterns.

Geographic data is a prime example of an area where a lot of work needs to be done so that the average user can view and derive conclusions more easily, without needing to use professional and expensive tools. Comparing with other lifelogging areas, one cannot extract meaningful information by plotting complex geographic information in a graph or table.

The ease of use is as important as the effectiveness and efficiency of the algorithms to power such systems. If a user records a track a day, every single day, which is a simple and straightforward process — start recording when leaving a place and stop the recording when one has arrived at the destination — annotating the data, to give it personally relevant meaning, could become a bothersome task that can demotivate the users.

Also, GPS data has a lot of measurement errors, in some cases such as when used for satellite navigation those inconsistencies with the reality might not be important, or there are already specific techniques to solve those specific problems. But when we want to extract information from tracks, those errors can skew results, and thus compromise important conclusions and the true story of our lives.

1.1 Objective

Within this frame we have built *GatherMySteps*, which is a semi-automatic, user-assisted application to process GPS tracks. Our main goal is to:

provide users with interactive tools to efficiently process track data from personal GPS tracks, taking into consideration meaningful personal semantics regarding location and travel information.

In our usage scenario users collect tracks of their trajectories, that they will later upload to the system, to clean, correct and annotate with their personal information. The amount of tracks collected can be huge and the time interval that they represent can range from days to months, or even years.

Also, the amount of time that the users have to dedicate to do all those steps is not insignificant, it can become burdensome quickly, leading to dropouts as it may not represent significant short-term benefits.

Our application uses personal GPS tracks, that users can edit and annotate. As such, referring back to the highly time demanding task of processing GPS tracks, usability is our main concern.

To help the user identify and better recall a day, the map and the tracks being processed are the focal point of the UI, helping users see which path they have taken, from place to place. It also makes it easy

to spot any errors, caused either by recording or by the automatic processing, allowing users to correct them.

Furthermore, we have streamlined the number of steps to process and annotate a track, to a bare minimum; having three major stages: preview, adjust and annotate. The preview and adjust stages allow users to make changes and correct tracks. This process is simplified by doing automatic processing, approximating raw GPS tracks to the true trajectories taken by users — trips. This helps users identify and recall their days.

The annotation stage allows users to annotate their days, with the help of the recorded trips, marking their locations, transportation modes, and other personally relevant information. To further help users, the system offers suggestions about locations and transportation modes, which can be learned iteratively, adapting to the users.

To test our application, we did usability user tests and utility use cases. Usability user tests allowed us to infer whether *GatherMySteps* reached the goals of being user-friendly and simplify the processing of everyday personal GPS trajectories in the context of lifelogging. To assess how useful our application is to lifeloggers we asked 5 participants to record their personal trajectories and then to use our application. The results show that our application is easy to use and understand.

1.2 Contributions

We have created and open-sourced the user interface, **GatherMySteps**. During our work we have also made other contributions to the scientific community and to the open-source software community in general.

We developed, and open-sourced the **TrackToTrip** library, which allows to manipulate tracks, and extract information from them. Within the frame of **TrackToTrip** we have created the **Trajectory-Hausdorff Ratio algorithm**, which allows to compare two tracks, giving a score of how similar they are.

We have also forked the track editor as a separate web application, **GPXplorer** [6], to allow anyone around the world to explore and edit their GPS recordings.

Furthermore, we have published a paper [Gil & Gonçalves, 2016] based on this dissertation was accepeted at the Encontro Português de Computação Gráfica e Interação 2016 (EPCGI'2016) conference.

1.3 Dissertation outline

The rest of the document is organized as follows. In Chapter 2, we discuss related work of systems that display and process GPS data highlighting where they fall short and differences in approach, analyzing it in Section 2.5. Chapter 3 is dedicated to analyze geographic lifelogging, its problems and challenges, and how to work through them. In Chapter 4 we detail the *TrackToTrip* library, and the decisions behind the current implementation. The backend module, *ProcessMySteps*, is discussed at Chapter 5. In Chapter 6 we discuss the problems and decisions behind the main component of our system, the UI, *GatherMySteps*. Chapter 7 describes how we evaluated our application. Finally, in Chapter 8, we draw conclusions upon our work, and reflect about how to improve this project.

[6] `https://ruipgil.com/GatherMySteps`, last accessed on October 15th, 2016

Chapter 2

State of the Art

Our application will be comprised of various components that translate our concerns. We start in the *Visualization* section, which presents how some projects have displayed GPS track information. In the *Where am I?* section we go over some work done on how to extract location data. Next, the *How did I get there?* section focus on applications that infer transportation modes based on GPS information. Finally the *Filling the gaps* section presents some papers on to deal with some common problems inherent with GPS tracks.

2.1 Visualization

Tracks are incredibly difficult, if not impossible, for a user to comprehend without a visualization. Below we are going to see some projects, that while not necessarily centered on the visualization, provide ways to visualize tracks and information associated with them.

2.1.1 GeoLife: Managing and Understanding Your Past Life over Maps

GeoLife [Zheng *et al.* , 2008a] is a system that takes data provided by users and aims to display it in a meaningful way. This is achieved by inferring the transportation methods, by extracting significant places and life patterns of the user. This is all done using the raw GPS points of past trajectories.

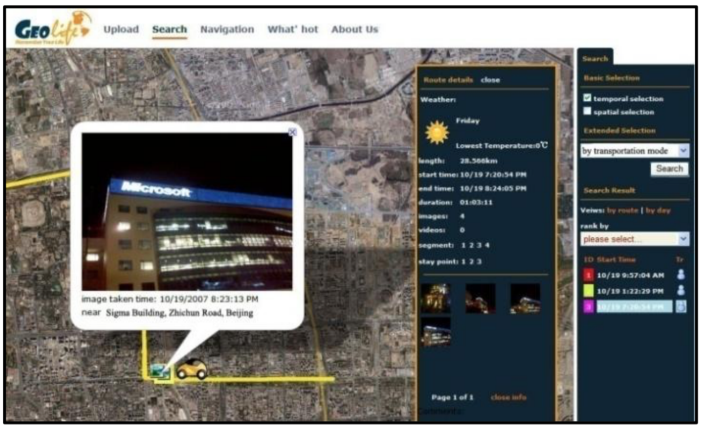


Figure 2.1: *GeoLife* prototype showing details about a route

Using publicly available data *GeoLife* is also capable of learning the most popular sports trajectories and travel routes, as well as popular places and traffic conditions of trajectories along the time.

To infer the transportation mode, it uses supervised learning to distinguish between a set of different transportation methods, such as: walking, taking a bus, riding a bike and driving. This method of inference is also capable of segmenting a trajectory into multiple transportation methods. This is completely independent from maps and other sensors and adapts to the user.

Users are capable of navigating past trajectories through a web app, where they can see the transportation methods used in each part of the trajectory, as shown in Figure 2.1. To recollect past experiences, and memories users can make use of spatiotemporal queries.

2.1.2 Searching your life on web maps

GeoLife, as described in Section 2.1.1, is at its core an application that allows users to explore their past life events, mainly through recorded trajectories.

Besides the development of the application, the team behind *GeoLife*, also tried to answer two questions [Zheng *et al.* , 2008c]:

- Do GPS track logs and associated images improve our memory of the past experiences beyond what is normally remembered?
- And, are spatiotemporal search functions more effective, when it comes to triggering memories from past events?

Visually, at the application level, tracks are displayed on a map and an animation can be played to replay the trajectory recorded. Each GPS track has a profile, which includes the date, weather information, start and end time, length, duration, average speed, etc. To further improve recalling past experiences, users can upload geotagged images, which in turn will be associated with tracks.

To provide a better navigation between tracks, users can apply spatiotemporal queries. By specify a time interval or region, or both, tracks that conform with those constraints are filtered and displayed.

To answer the questions cited above, the authors of *GeoLife* conducted an experiment involving a group of 36 users, each one of them was asked to record their trajectories for 30 days.

The participants were then divided into two groups, of 15 people each. Each group was then, submitted to two rounds of recall tests. In the first round, each group would try to recall past events using only their memories. In the second round, one group used *Map-Image Browsing* and the other *Map-Image Searching* to recall past experiences.

The results show that no matter the type of cues, memory about past life deteriorates over time. Recalling past experiences is, however, more difficult when there is no cue available. Using *Map-Image Browsing* and *Map-Image Searching* users were able to increase the recall by 40%. Between the two methods, *Map-Image Browsing* and *Map-Image Searching*, *Map-Image Searching* produce slightly better results, around 0.5% and requires less effort, reducing the time subjects spent on answering the questions in more than 200ms (around 20%).

The *GeoLife* offers simple, yet, good tools that allows users to recall past experiences. Even though it could use more personally relevant data, such as user annotations of locations, which in turn may help users recall faster, and more accurately past experiences.

2.1.3 System for Real Time Storage, Retrieval and Visualization of GPS Tracks

The *MOPSI* system [Waga *et al.* , 2012b] [Waga *et al.* , 2013a], allows users to see and explore GPS tracks on a digital map, such as *Google Maps* or *OpenStreetMaps*. It also allows transportation mode extraction from those tracks.

To complement the lifelogging aspect users, can use geotagged media, such as photographs and videos, which are associated with the tracks.

The system is available through a mobile app and through a web interface. And it is comprised of three stages: recording, data management and visual representation. The mobile app is capable of recording the position of users as a GPS point, which is a triple composed of latitude, longitude and timestamp at a given interval, usually every two seconds. GPS tracks are not stored in the user's devices, instead they are sent to the cloud.

When the data arrives at the servers it is stored but it is not immediately processed, instead — and due to the large amount of data — it is processed by a cron job at a given interval. The database stores the track's metadata and characteristics, such as its bounding box, and start and end point. The full track is also stored as a plain text file.

When processing the GPS points, first, they are grouped into tracks and then, the features of each track, like transportation mode, average velocity and acceleration are extracted. Finally, a track that is similar in shape but with a fraction of the points is computed using the *Multiresolution Polygonal Approximation Algorithm for GPS Trajectory Simplification* [Chen *et al.* , 2012]. The simplification step is crucial to provide better performance when transferring and displaying tracks.

The system can show the location of the start and of the end of a track and animate it, recreating the user trajectory. The trajectory is displayed as a line, based on the set of points given to the algorithm. Additional features such as transportation method, photos that the user took along the path, duration, length and average speed are also displayed, as represented in Figure 2.2.

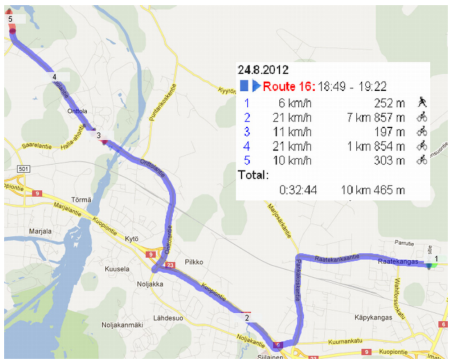


Figure 2.2: *MOPSI* UI, showing route segmentation and statistics review

To display tracks, the UI component queries the servers for the points inside a bounding box. To keep the user interface smooth and memory efficient, the bounding box requested is 50% bigger than the viewport size being used by the user.

After experiments, results show the team behind *MOPSI* was able to create a system capable of displaying tracks efficiently. Tracks with 10000 points, before being simplified, take one second to be displayed. While online tools like *GPS visualizer* and *GmapGis* needs approximately five seconds.

Profiling the whole system shows that 82% of the time is spent querying while 14% of the time is spent in the browser and 4% is used for the bounding box.

MOPSI follows the good policy of not over-simplifying the track, reducing the number of points in a non optimal manner, which allows data mining to be more precise. It also makes good use of *Google Maps* and *OpenStreetMaps*, using one over the other when it is more beneficial to the users.

2.2 Where am I?

There is a lot of information that could be derived from tracks, one of them is the personally relevant location of the place from where one goes and from where one comes from. The following papers focus on extracting these places.

2.2.1 Discovering Personal Places from Location Traces

When GPS logs are recorded their value is hidden in its semantic information. Muhammad Umair [Umair *et al.* , 2014], proposes an algorithm to extract the most important places to a user, based on his GPS logs. It can also be used to produce a recommendation system based on the user location history.

The most important places to a user are considered places where he spends more time, so by correlation, where there is a higher concentration of GPS points.

Figure 2.3: Places inferred from raw GPS data

An Android application was developed to record GPS signal, process the signal and extract the semantic locations, and to visualize those results.

The application records the GPS signal and timestamp, at dynamic time intervals, depending on the current activity.

The points recorded are then used by the algorithm, which is responsible to identify dense areas of points. By checking the places where the user spends the most amount of time, the algorithm is able to determine potential personally relevant places, such those in Figure 2.3. To prevent some of GPS's problems, like the lack of accuracy around the same place, outliers are removed.

The visualization component shows the users their routes while highlighting their important places in the map.

Experimental results show that this algorithm is cable of correctly identifying, compared to the ground truth, the location of GPS trajectories.

This technique gives a good prediction of the important places to the user, with a technique based on relatively simple principles, and it is good by not needing to know a priori the number of significant places that there will be. It lacks the fact that the user cannot provide its own places, or adjust existing places.

2.2.2 Extracting patterns from location history

Location history can be used to determine which places the user frequents and how much time he spends there. Andrew Kirmse [Kirmse *et al.* , 2011] presents a technique to extract those patterns.

The location history is a list of timestamped points, which in turn are latitude and longitude tuples, the accuracy of the measured points, and the input source, which can be one of GPS, WiFi or cell tower triangulation.

The first step is to pre-process the location history, removing noise points to compensate for the, already discussed, GPS accuracy problems. All points over sea waters, before the collection has been

started and points that present non-physical velocity are removed. Jitter is identified and the points that cause it are removed.

With a cleaner and more truthful set of points, the computation of the most frequent visited places, the computation of home and work locations, and the commute analysis can be done.

To identify frequently visited places, the points where the user is moving at a low velocity, described as stationary points, need to be filtered and clustered. Two algorithms are used to cluster points, Leader-Based Clustering (LBC) and Mean Shift Clustering (MSC) [Cheng, 1995].

LBC consists on, for every point, checking if it belongs to any of the already generated clusters by computing the distance to the cluster leader. It runs in $O(NC)$, where N is the number of points and C is the number of clusters. This method may create different clusters with different ordering of the points.

MSC is an iterative method that finds the centroid, or centroids, of a point cloud by gravitating a centroid to a region. Points with higher accuracy have lower weight, so that points with less accuracy will gravitate towards high accuracy points. It reverts back to leader-based clustering if the points do not cluster or do not converge quickly enough.

Usually, MSC generates better clusters than LBC and does not depend on order, but it is computationally more expensive. Tests show that MSC converges in 2.4 iterations on average, and 3% of points would not be clustered just with MSC. It also did not cause a significant reduction of computed clusters (less than one percent), compared with LBC, therefore it does not justify the additional computational cost.

After obtaining the clusters with one of the algorithms above described, relevant information can be extracted from them.

The Adaptive Radius Clustering algorithm is used to rate how interesting a cluster is. It consists of relating the importance of a cluster to its size. Users were surveyed and the majority of the clusters produced were relevant to the users.

Home and work location are computed using data from the points. Generally, home corresponds to the clusters composed of main night points and work is derived from the main weekdays cluster.

Places visited, within a distance threshold of home and work, are removed, as well as points that correspond to the commute between home and work. Since a stop at the middle of the trajectory could cause a cluster. Points near airports are also removed.

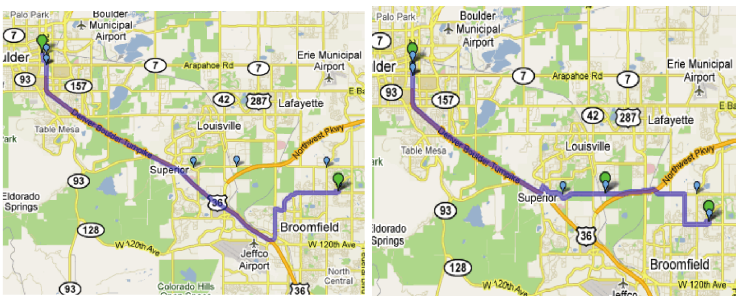


Figure 2.4: Iterations to reach the final map matched path

To analyze the track a set of commute points are extracted. The initial set of points are filtered out, based on their low accuracy. And a pair of source-destination points is then obtained. The commute points are then map matched using iterative queries to *Google Maps*. First with the source and destination points, if the commute points are within the accuracy threshold distance the route is identified, if not the farthest point is added iteratively until all points are within the accuracy distance.

Paths are then spatially and temporally clustered to generate the most common commute taken by the user. Two paths are spatially close if their *Hausdorff distance* [Rockafellar & Wets, 2009] is within

a threshold. A variant of LBC is used to generate commute clusters; this procedure is illustrated in Figure 2.4.

Inferring a location based on the time of day and the frequency of those places are a good idea, one that falls short when the users do not have a standard schedule, like nurses. Also, there are some privacy concerns, because the tracks are map-matched with *Google Maps*. That factor, also makes it very coupled with out-of-control software and may hurt performance.

2.2.3 Learning Location Correlation from GPS Trajectories

When faced with a large amount of trajectories, there are valuable information that can be extracted. The results may help make bus routes more efficient, it may help produce more effective and personalized tour guides and travel recommendation, or **it can shed light upon past experiences of the user**. Yu Zheng [Zheng & Xie, 2010] describes how locations, extracted from stay points, can be used to extract relevant information.

A track is composed of a sequence of GPS points. When a set of track points are close they can be grouped into stay points. This stay points can carry meaning to a location. Extracting them can give important cues of where the users have been. Different tracks will produce different stay points. The method proposed takes tracks and groups them, based on clusters formed by their stay points.

Given a different number of tracks, one can infer the travel experience by correlating locations.

To determine travel experience, its assumed that the interest of a location is directly related with user experiences visiting it. Based on that, a user experience and each location interests can be derived using a Power Iteration Method.

Meanwhile, location correlation between two locations are not solely dependent on the number of users visiting it, but also on the users travel experiences. Locations visited continuously are more correlated than those visited discontinuously. Such correlation can be calculated by integrating the travel experiences of the users who have visited them in a trip in a weighted manner.

To test this methods, and assumptions, an experiment was conducted with 23 subjects, that gathered GPS tracks representing their location histories over a year. The subjects knew the region well, as they had been living in the region for more than six years.

The results show that this method is more effective than the *Weighted Slope One* [Lemire & Maclachlan, 2005] algorithm with a slight increase in computation demand, while giving similar effectiveness but being more efficient compared with the *Pearson Correlation-Based Model* [Huang *et al.* , 2004] algorithm.

This approach makes some assumptions, and while giving important insight, it may fall short is some cases specially when applied to local residents, since there are some places where the time spent at one place will not correlate directly to the experience, as users stuck at a line in a convenience shop will have their experience degrading. This method seems more reliable and appropriate when applied to tourists.

2.3 How did I get there?

In a way, how we live our lives can be inferred from how we move. This information is often lost, even when we record tracks of our daily trajectories. To address that concern the next section shows how transportation mode can be inferred.

2.3.1 Motion Pattern Analysis Enabling Accurate Travel Mode Detection from GPS Data Only

Analyzing travel mode in smaller cities, such as many European cities, results in a different set of challenges, since the topology of the cities are different.

Richard Brunauer [Brunauer *et al.* , 2013], explores three different machine learning techniques to infer the travel mode used in a trajectory, composed only of timestamped GPS points, recorded with a sampling rate of one point every one to ten seconds.

From the GPS points a total of 54 features, such as: speed, acceleration and horizontal angular speed are extracted. From those features, track segments are labeled as either *walk*, *bike*, *drive*, *bus* and *train*.

These features are then feed to Multilayers Perceptrons (MLP) and two kinds of decision trees, which consider the distinct frequency distributions of each transportation mode, the transportation mode used in this trajectory is then extracted.

MLP classifiers are feed-forward networks with three layers: input, hidden and output. The training is done with a back-propagation algorithm.

C4.5 and Logistical Model Trees (LMT) were the selected types of decision tree classifiers. *C4.5* presents a staircase-shape structure, while the LMT have an additional logistic regression function at their leaf nodes. The models are trained with the *LogicBoost* algorithm.

All classifiers are validated using *k-fold* cross-validation. To overcome overfitting in decision trees, lower branches, very specific are pruned.

The 54 features extracted can be divided into three categories: eight statistical features representing general motion characteristics, 45 representing movement characteristics of different modes and a final time series-based feature.

The statistical features are minimum, maximum, mean and standard deviation of average speed and average acceleration.

The motion pattern features are derived from discretization of average speed, average acceleration and average horizontal angular speed. This results in frequencies for each components and can be visualized with three histograms. Also, the absolute frequencies are transformed to relative frequencies for the three components.

The final time series-base feature represents the bendiness of a trajectory.

MLP, LMT and *C4.5* show an accuracy of 92.24%, 92.09% and 84.48% respectively. This shows that MLP and LMT have identical accuracy. But, they fail in different scenarios. For instance, MLP is better at recognizing bicycle, while LMT is better at classifying train and bus classes. Both *decision trees* are worse at identifying bicycle transportation modes. All three methods are likely to misclassifying car and bus, this is because the derived motion characteristics on motorized vehicles are similar. Bicycle and walk are classified with a high accuracy, around 98%, because of their relatively large speed bins.

Also, there are classes prone to confusion, such as, highway drive vs intercity train, and city drive vs public bus. Also, using these methods to classify biking and hiking in mountainous areas may result in misclassification, since data errors, such as jumps in the coordinates, are more likely and frequent.

The authors also experimented reducing the feature set to ten features and the accuracy recorded was around 90%, for a set of 30 features an accuracy of 92% was obtained.

This method is fixed to the same, hard coded, types of motion. This has benefits, as they can choose which methods produce best results when distinguishing them, but activities like running or other user defined activity is not taken into account. It does not take into account past motions, that could be annotated by the user.

2.3.2 SenseMe: A System for continuous, On-Device, and Multi-dimensional Context and Activity Recognition

SenseMe [Bhargava *et al.* , 2014a] aims to be a context-aware system using not only GPS positions but also other sensor information provided by modern smartphones, such as gyroscope, accelerometer, bluetooth connections, wifi and GSM network information.

It tries to answer the questions 'Who, What, Where and When?' by giving information on the following areas: environmental context of the user (indoor, outdoor and indoor-outdoor), physical activity (stationary, walking, running and in-vehicle), location (context aware location, such as home, work, and a sports arena), smartphone usage habits (what apps a user uses, for how long, and when) and finally the amount of social interactions (how many people the user has had contact and how long those interactions have lasted). The system runs entirely on the smartphone, where the data is collected and processed.

To infer the context, a *C4.5* classifier is used, chosen among others as the most accurate alternative. To classify the users' location as indoor, outdoor or indoor-outdoor, GPS position is collected every ten seconds. Each minute, its six samples are classified and the majority of labels will be assigned to the entire minute. The use of a *first-order Markov Chain* makes the process resilient to outliers.

Physical activity is inferred using raw GPS signals. The velocity is extracted in an interval between 10 to 60 seconds, depending on the motion of the device. Each minute, a vector with those values is collected. From it, features will be extracted, such as the minimum, maximum, average and the variance of the speeds.

Those features are used as input to a *C4.5* classifier, which will label the activity as either *Stationary*, *Walking*, *Running* and *In-vehicle*. Smooth the results, a sliding window of size three is used, which will filter out any outliers.

The context-aware location is extracted when the user is deemed stationary. It is inferred using the, previously extracted, environmental context and current physical activity. If the room is indoor or indoor-outdoor, it will use the *Locus system*.

The *Locus System* uses WiFi to determine the room, floor and building where the user is. If a position cannot be determined, it resolves to the last outdoor location, and stores all the unresolved indoor locations. The location type is also extracted, when the user is outdoors using *FourSquare Venues API* or *Google Places API*, and when the user is indoors the user can introduce the type of the building and the type of the room.

Social context recognition is done checking the number of persisting bluetooth connections that are over two minutes.

Data for training and to measure the system was collected by for members of the lab in different conditions, several times a day, and it was annotated to provide ground truth. Results show that the system shows good accuracy compared with the ground truth, with:

- the environment context recognition to be 91.23%
- the physical activity recognition to be 95.75%
- the context-aware location to be 93.12%
- device activity recognition to be 99.1%
- social context recognition to be 87.5%

Also, upon interviews users of the system found the environmental context metrics to be the most interesting, and voted the Device Activity and Social context to be the least interesting.

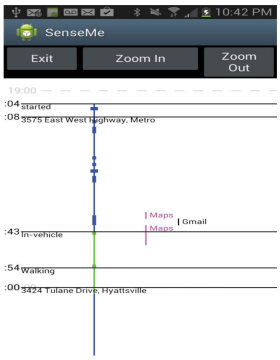


Figure 2.5: SenseMe UI for viewing past tracks

SenseMe has two major pitfalls. The UI, Figure 2.5, to view and review data looks outdated and not particularly clear. There is only one screen where all the information is displayed and it cannot be explored in a finer detail. And also, the lack of mechanisms to use, and learn with, annotation made by the user.

2.3.3 Learning Transportation Mode from Raw GPS Data for Geographic Applications on the Web

GeoLife, as described in Section 2.1.1, aims to provide a solution to an array of problems, as it has been explained previously. One of those problems is to learn transportation mode, using raw GPS tracks, composed of timestamped GPS points [Zheng *et al.* , 2008b].

To learn the transportation mode, the GPS tracks need to go through three different stages: a change point segmentation method, an inference model and a post-processing algorithm based on conditional probability. After this process they will be labeled as either *Bike*, *Bus*, *Car* and *Walk*.

The change point segmentation method is based on common sense knowledge of the real world. It works by assuming that when people change transportation mode, there is a small spatiotemporal pause and, that walk should always be the transition between two segments. From each segment, a set of features is extracted. They are: length, mean velocity, expectation of velocity, covariance of velocity, top three velocities and top three accelerations. This information is then passed along to the inference model.

The authors of *GeoLife* tested four inference models: Decision Tree, Bayesian Net, Support Vector Machine and Conditional Random Field Classifiers. After initial experimentation the Decision Tree was chosen as it outperformed the other models.

Finally, the post-processing algorithm which considers the transition probability between each transportation method is used to smooth the results, reducing errors such when a car is stuck in traffic.

2.3.4 Detecting Movement Type by Route Segmentation and Classification

MOPSI system, as described back in Section 2.1.3, [Waga *et al.* , 2012a] also aims to use the sensor data from the device to infer information about trajectories, such as: his position, context and transportation mode.

Transportation mode inferring is done using GPS and accelerometer data. And consists of route segmentation, to find different transportation modes within the same trajectory; and moving type classification.

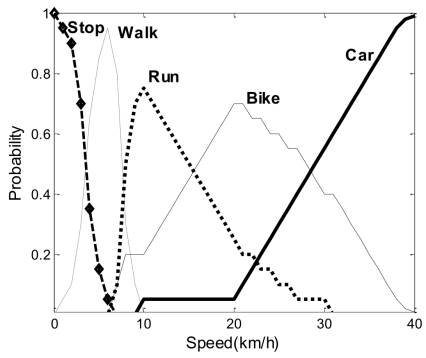


Figure 2.6: Overlap between a priori probabilities for soft classification of the route segments

Route segmentation is done by dividing a route into segments with consecutive similar speeds. With the segmentation features, such as acceleration, duration, and distance, can be extracted. But, training a classifier directly on these features is not accurate because they have huge overlaps.

First, the segments are soft classified, based on a priori probabilities (Figure 2.6). Each segment is labeled either: *stop*, *walk*, *run*, *bicycle* or *motor vehicle*.

Then a Hidden Markov Model (HMM) is used to exploit correlations between neighboring segments, where the hidden states represent the movement types and the observed data are the features for each segment. Which is then extended to second order HMM by exploiting correlations both to the previous and the next segment. The transition matrix is empirically initialized, but could be trained.

The experiments show that, although the algorithm is not perfect it is good enough, and it can detect speed changes within the same transportation mode. But, one segment misclassified as motor vehicle increases the probability of similar segments to be classified with the same transportation mode. A more personal and adaptive system should include annotations to improve its training.

2.4 Filling the gaps

In the next section we look at work, and their techniques, that addresses the fact that consumer grade GPS recording equipment is unreliable, producing points that are in fact errors, and they may lose connection with satellites, that causes a void of information. We also analyze alternative ways to represent tracks internally.

2.4.1 Path Estimation from GPS Tracks

GPS data collected from small, consumer grade devices have a low accuracy and a lot of the points deviate from the true position of the device. This can be seen on Figure 2.7, where the points are plotted on a map are scattered around the true trajectory. Chris Brunsdon [Brunsdon, 2007] proposes a improved approach, based on the *Principal Curve Analysis* [Hastie & Stuetzle, 1989], to generate a 'middle curve' represented by a set of points, which is a estimation of the middle curve.

Figure 2.7: Point cloud of a GPS track

With *Principal Curve Analysis* the distance to be minimized can be in any direction, depending on the line joining a GPS point and the closest point to it on the principal curve while in standard deviation the distance can be measure in the any direction. This method is good enough with low noise, but when there are some sequential noise points it will deviate from the true trajectory too much. To address these problems, a new and more robust algorithm was devised. First, by using the standard approach a curve is generated, then, the distances are normalized according to their standard deviations. Finally, the principal algorithm is applied, using weights (as a decreasing monotone function) in the nonparametric regression stage.

The accuracy, assessed using images such as in Figure 2.7, is good. It creates close approximations to the trajectory taken. The precision, assessed though the *Bootstrap Approach* [Efron, 1981], produces a lower sampling variability, which leads to more approximate graphs to the ground truth.

As the point cloud increases so the points of the points of the principal curve. The Ramer–Douglas–Peucker (RDP) algorithm is proposed to reduce the number of points, while efficient and effective, it does not maintain spatiotemporal constraints.

This method produces good results fitting a line into a point cloud, but it lacks in computational efficiency making it ill suited for online track processing.

2.4.2 Multi-track Map Matching

Track map matching is a technique that approximates a GPS track to the road topology. Map matching is simpler when dealing with a dense point cloud. But, when dealing with a sparse point cloud there is more uncertainty, making it harder to map match it to the correct road segments. To address these concerns Michael Javanmard [Javanmard *et al.* , 2012] uses multiple tracks to infer the correct path.

The objective is to turn a sparse point cloud, using one track, into a dense point cloud, using multiple tracks. Multi-track map matching is a specialization of single-track map matching.

Single-track map matching aims to minimize a regularized cost function that balances the data and model errors on measuring the quality of path. The cost function is very similar to what has been used before, on Section 2.3.4, based on HMM.

Even for a small map, the state space to find the global minimum is huge. To improve performance a pruning procedure is applied, reducing the solution space. To improve performance even further, dynamic programming is used to find the minimum solution of the pruned set.

The next step is to build a *multipartite graph*, where for each part we consider a small set of road points, match candidates, one of which will be matched to a point from the GPS point cloud. The match candidates should be chosen such that they represent well all possible points that might have generated the point in the road segment, while keeping its size small for fast computation.

Multi-track map matching is achieved using the described single-track map matching technique. But first, there needs to be a global ordering of the points in the point cloud. There are two alternative methods presented: *iterative projection* and *graph laplacian*.

The *iterative projection* method consists on projecting a point into a segment, and orders them according to their position in the segment, the result will be a candidate path, which is then map matched. The procedure is then repeated until it either converges or it has been run for too many rounds.

The *graph laplacian* method defines a distance metric between any two points, estimating the path distance between the original samples on the real path — which may have come from different tracks. The *laplacian* is then formed using exponential weights. The eigenvector is computed, the values in the vector are sorted, and returned as ordered.

Since both the *iterative projection* and the *graph laplacian* approaches are susceptible to noises, specially when there are outliers with an extreme amount of noise, it can cause the methods to get trapped in the wrong path. To prevent this issue, and consequently improve performance of both methods a boosting process is used.

The boosting process consists on generating a number of subsets, sampling the original set of data points with some probability. Each subset is then ordered and scored. This way outliers have a greater probability to be excluded.

The single-track map matching technique produces visual results close to the ground truth. Results also show that when increasing the number of tracks, booth multi-track map matching methods produce good results, the *iterative laplacian*, however, is better than the *graph laplacian* with a lower amount of tracks. Both methods improve significantly with the boosting process.

2.4.3 Constructing popular routes from uncertain trajectories

Humans are creatures of habit, both individually and collectively. In our day-to-day lives we tend to walk the same paths, take the same buses, or drive the same roads. With this in mind Ling-Yin Wei [Wei *et al.* , 2012] proposes an approach to infer a trajectory based on multiple trajectories uncertain trajectories, using the GPS points. Which is particularly useful when dealing with low sampling rate of GPS tracks.

The method is composed of two components: a routable graph and edge inference.

A routable graph maps the uncertain trajectories into a grid, such that track points will correspond to one cell of the grid. Each cell forms a vertex in the routable graph. Edges between cells are inferred with the uncertain trajectories. Edges can be inside or between regions. Inferred information consists of a moving direction, a transition support and a travel time indicating the transition between the relationship between two cells.

Edge inference receive as input, inferred edges and a query. The algorithm infers the top-k rough routes, where each one of them is represented by a constructed graph. Then the qualified sub routes between any two consecutive query locations are extracted and concatenates them into completed routes in a branch-and-bound manner.

To give more importance to the more traveled routes a score function ranks the routes based on their history. Finally, the rough routes are derived to produce a detailed route represented by a sequence of consecutive segments, all based on historical data points of uncertain trajectories.

Experiments show an accuracy of 90%. And it can find the top three routes within 0.5 seconds, with a distance error smaller than 300 meters.

This technique, while conceptually simple, produces a high degree of accuracy, but it lefts out some important features about the trajectories, that could be used to provide a even more accurate model. Such features could be the time that a user took to reach one place.

2.4.4 StarTrack: A Framework for Enabling Track-Based Applications

StarTrack [Ananthanarayanan *et al.* , 2009] aims to provide the building blocks for application that deal with tracks, by providing a Representational State Transfer (REST) Application Programming Interface (API), on a client-server structure.

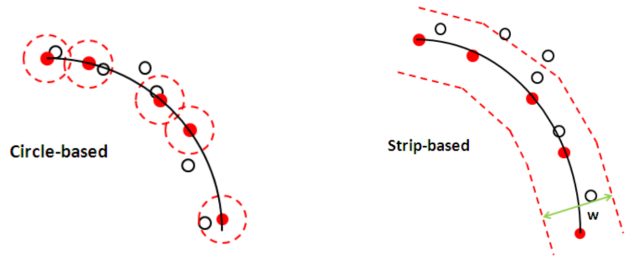
It is independent of the type of data provided, points can originate from GPS, WiFi or cell tower triangulation, where all that is required is a sequence of timestamped latitude and longitude tuples.

The *StarTrack* system provides five categories of operations over tracks: recording, manipulating, comparing, clustering and querying. The most relevant of them are the operations of comparing and clustering.

The comparing operation determined if two tracks are similar to one another, and to which degree. When comparing tracks, there are two dimensions that should be considered: shape and time. Comparing a track temporally is straight forward. Spatial comparisons of tracks are more complex.

It is important to notice that a track is an approximation of a path, and when it is collected its susceptible to errors, noise and small variations. Factors such as sampling variability and minor detours can alter to what degree a track is similar to another, but it should have a small contribution to the overall similarity. Also, when a path represented by one track is included as a subset in the path represented by another track, and interesting segments, such that long paths with different start and end locations may have segments that they share, thus making tracks with a relevant degree of similarity.

To do it a statistical filtering technique is used, by creating a region that involves all the points of one track, and then checking which points of the second track fall inside that region.



(a) Circle-based method (b) Strip-based method

Figure 2.8: StartTrack's region creation algorithms

Regions can be defined by two algorithms: a circle-based algorithm (Figure 2.8a), that defines a circle of a certain radius around track points and, a strip algorithm (Figure 2.8b), that creates a strip around the track itself with a certain width. Both radius and with of the respective algorithms represent the accuracy of the comparison and they may vary throughout the segments.

Track clustering is used to eliminate near duplicates. It uses the k-*medians* approach, which consists of dividing a group of points into, k groups, such that the distance between the points and the center of the cluster is minimized. k-*medians* needs, however, to know beforehand the number of clusters to generate and only works for offline tracks. Since all the input needs to be available, from time to time, clusters may need to be recomputed.

The system uses a SQL database with support for spatial data types and indexes, where points and tracks are all stored.

Experiments were performed using tracks from nine volunteers, with a sampling rate of 30 seconds. Synthetic tracks were also generated from the terrain topology, and include differences in sampling rate and measurement errors.

The results show that for the comparison algorithms, the strip algorithm is more accurate. The circle

algorithm produces good enough results, too. When using dynamic accuracy (adapt radius and width), the results were much more precise at a negligible cost. The time complexity of the algorithm stands at $O(N^4)$.

When querying the system, the slowest part of the process is the comparison of the tracks, specially for a large amount of them. Comparing the two algorithms, the circle algorithm is up to 37% faster than the strip algorithm.

2.4.5 StarTrack Next Generation: A Scalable Infrastructure for Track-Based Applications

The *StarTrack* system described above, had a small scalability problem when faced with a large amount of tracks. The system was re-designed Maya Haridasan [Haridasan *et al.* , 2010] to, amongst other things, improve performance.

There were several core changes to the project. Track collections were introduced, so that operations could be performed over a group of tracks, instead of working with one track at a time. There were changes regarding the API, the track representation, storage and overall architectural structure.

The past representation of tracks, represented by GPS points, consumed too many resources and it did not make the comparison easy. Upon this realization, instead of storing every point, they create canonical representations of tracks. The canonical representation is a simplification of the original track, done by map-matching points to the road network. The result is a tree with nodes, that represent intersections, and edges, that represent road segments.

This new representation allows for a more efficient comparison between tracks. To further boost the performance of the system, the tree is cached in memory.

The evaluation was done similarly to the original *StarTrack* [Ananthanarayanan *et al.* , 2009] system, where there are real tracks, in this case ̃16,000 tracks, as well as synthetic tracks, that amount to more than four million tracks.

The results show that the performance of transforming a track to its canonical form, is dependent upon sampling rate, length and GPS noise. A track with a length of 20km and 400 points is transformed under 250ms.

An average track was 20km long, containing 400 GPS points and yielding 163 points after canonicalization. The construction of the track tree takes linear space, and slightly super-linear time. The height of the tree grows logarithmically with the number of tracks learned. Using a track tree makes the retrieval of similar tracks, a highly efficient process, compared with the past method. Although, it may not return all tracks that are similar enough to a value (algorithmically it is not complete).

2.5 Discussion

Looking at the papers referenced earlier, we can see where they overlap and the different techniques they use to solve problems related with the GPS signal, its recording, visualization and ways to extract hidden information. That comparison is summarized on Table 2.1. The table compares different dimensions.

The input types, that can be GPS points, Timestamped Location (TSL) points, or a combination of more sensors. GPS and TSL are very similar and in practice do not differ, since the only information extracted from them is the location.

Some of the projects analyzed, focus on three main categories when it comes to the platform where they were designed to run.

Paper	Input	Platform	Additional Metadata	Personal	User input	Learns	Preprocessing	Transportation	Location	Route inference
[Zheng *et al.* , 2008a,c,b]	GPS	Web	Photos, Weather	No	No	Yes		Change Point-Based Segmentation, Decision Tree, post-processing (Bike, Bus, Car and Walk)	No	No
[Umair *et al.* , 2014]	GPS	Mobile	No	Yes	No	Yes		No	Time clusters	No
[Waga *et al.* , 2012b,a, 2013a]	GPS	Mobile, Web	Photos	Yes	No	No	Track simplification: Multiresolution Polygonal Approximation	Consecutive Similar Speed, HMM (stop, walk, reun, bicycle, motor vehicle)	No	No
[Bhargava *et al.* , 2014a]	GPS, Sensors, WiFi, GSM	Mobile	No	Yes	No	No		C4.5, 1st order Markov Chain (stationary, walking, bus, bike, driving)	Locus, Google Places API, Foursquare Venues API	No
[Kirmse *et al.* , 2011]	TSL	Algorithm	No	Yes	No	Yes	Rejects invalid locations, map matching with Google Maps	Commute, Non-commute	LBC, MSC	No
[Zheng & Xie, 2010]	GPS	Algorithm	No	No	No	No	No	No	Power Iteration Method	No
[Brunsdon, 2007]	GPS	Algorithm	No	No	No	No	Noise reduction: variant of Standard Non-parametric Regression	No	No	No
[Javanmard *et al.* , 2012]	GPS	Algorithm	No	No	No	No	Map matching: Iterative Projection, Graph Laplacian & Boosting Process	No	No	No
[Brunauer *et al.* , 2013]	GPS	Algorithm	No	No	No	No		MLP, C4.5, LMT (walk, bike, drive, bus and train)	No	No
[Wei *et al.* , 2012]	GPS	Algorithm	No	No	No	Yes	Yes	No	No	Routable Graph
[Ananthanarayanan *et al.* , 2009]	TLP	Algorithms (API)	No	Yes	No	No	Track simplification and clustering: Region Based	No	No	No
[Haridasan *et al.* , 2010]	GPS	Algorithms (API)	No	Yes	No	Yes	Track simplification: Canonical representation	No	No	No

Table 2.1: Comparison between the papers analysed

The personal dimension specifies if the work is user centric, and more importantly. The preprocessing dimension specifies the main techniques to clean and derive a new track, that helps further processing. The transportation dimension indicates the algorithms used to derive the transportation method and the possible labels returned by it. The location dimension presents how the papers infer semantic location. Finally, the route inference column shows how the different papers infer similar routes.

Most of the projects use GPS collected data, with [Bhargava *et al.* , 2014a] being the only one that, in addition, uses other sensor data. All of them have either, or both, two principal features in mind. They can display trajectories. And, they can extract information from them. Additionally, and to aid the user recalling past trajectories and experiences, some systems record, or allow the user to upload other metadata, such as photos or weather information.

Systems that are deemed personal have a visual interface, usually a web interface or a mobile application.

All of them use some sort of track preprocessing, either to simplify tracks for a easier retrieval or displaying, or to adjust to the noise produced by GPS track recording. Map matching is a recurring technique used to simplify and approximate tracks to a road network.

Transportation mode and location recognition are the focus of systems try to extract from tracks, mainly to provide a better overview and more detailed information about the user past.

Transportation modes are extracted using machine learning processes, where the most used are C4.5 and HMM. When comparing transportation inferring, we can see that *SenseMe* [Bhargava *et al.* , 2014a] — the only system to use more than location data — with systems that only use GPS data, the gains in accuracy achieved by having sensor information are negligible.

There is also a lack work concerning with how to complete incomplete tracks, the only work that, in a generic and impersonal way, tries to shed light on the subject is [Wei *et al.* , 2012].

What comes out has evident from this research is that the user is not being introduced in the loop. Different users have different habits, different necessities and different profiles. From the way they commute, to the places they visit. Users that, by whatever reason, do not ride a by to work should not have their track transportation mode marked as a bike, the same goes for other methods of transportation. Assumptions are also good to provide a baseline, to provide a suggestion.

For a personal system, personalized input is required, and **not a solution that fits all needs**. The personal input could make the extraction of features, and consequent predictions, from personal trajectories more personal, and therefore, more accurate within a few iterations of annotation by the user.

These techniques provide a good overview and baseline to what has been done, and some of the best ways to do it. Even though they lack exploration in the crucial area for us, that is **personal *geo logging***.

Chapter 3

Geographic Lifelogging

Nowadays, recording GPS information is cheap and straight forwards. The proliferation of smartphones allows almost everyone to have access to a GPS device, in their pockets, at all times. But, in most cases, these devices are not precise and reliable.

(a) GPS recording with Garmin Edge 500

(b) GPS recording with Motorola Moto G3

Figure 3.1: GPS accuracy between different devices, by Daniel Schmidt

Figure 3.1 shows us the lack of precision of GPS devices, by comparing an affordable smartphone that sold over 2.6 Million units in its first six months, the Motorola Moto G3 (Figure 3.1b), with a dedicated, high-sensitivity GPS device, the Garmin Edge 500 (Figure 3.1a), which was designed for performance-driven cyclists. The results are clear to the naked eye: the dedicated GPS shows better accuracy and a more detailed track, than the affordable smartphone.

Figure 3.2: Stationary GPS recording

Commercial GPS devices will also get wrong readings of the same position, even if stationary. In Figure 3.2 we can see the readings, in an indoor space, of a stationary device. Even though the device was not moving, the tangles created are visible.

If one decides to record personal trajectories, there are also other challenges. First, we have to consider how we record our trajectories, the three main possible workflows are the following:

- Start recording when we leave a semantically relevant location, and stop when we reach our destination. Repeat this process for every trajectory;
- Record the position of a semantically relevant location, at departure and arrival. Repeat this process for every trajectory;
- Record the whole day, never stop recording.

The difference between the first two workflows is that, while the first one records every point of trajectories, the second case only records the start and end points of every trajectory. The third case records all points, thus a more exhaustive cleaning of the data is needed. In our application, from the workflows presented above, we only consider the first and third cases.

Besides the workflow, one important consideration is human and machine error. The human error has to do with how accurately the user start and end the recording. The first case is much more prone to human error than the second. Because users may, accidentally, start recording after they have left a semantic relevant place or they may forget to restart the recording after they have made a quick stop, for instance in a grocery shop on their way home.

The machine error is introduced by the lack of reliability of devices. Sometimes a recording is stopped, either because there is no available GPS signal or the device fails to record a point to the device because of device restrictions.

3.1 Recording Personally Relevant Trips

There is a myriad of formats to encode GPS recordings, that range from CSV, to KML. The most widely used format, however, is GPS Exchange Format (GPX) [1]. The GPX format is a XML schema data format

[1] `http://www.topografix.com/gpx.asp`, last accessed on October 15th, 2016

that allows to describe tracks, segments, and waypoints, which are composed of points. Each point has to specify its latitude and longitude, with other properties, such as time and elevation, being optional.

The way tracks are recorded, and GPX files are created is tied to the application being used. Some applications may use features of the specification that others might overlook. An example of this is the creation of a new *trkseg*, for each start. While able to work with these cases, our expected format is to have all the points (*trkpt*) inside a *trkseg*, inside a *trk* element, for each GPX file.

Also, different recording applications vary on how new GPX files are created. Some of them create a file per day, so that all recordings that occur in the same day are in the same file, others create a file for each start. There is a myriad of permutations to consider when loading GPS tracks.

While, when it comes to recording GPS trajectories, GPX is the standard format, when it comes to annotate the recorded trajectories with semantic meaning, there are no standard that defines how things may be done. The *LIFE* format has been created [2] to allow the annotation of geographic information.

The *LIFE* format offers flexible and simple ways to annotate locations (stays) and trips. It allows to mark those stays and trips with tags and semantics. It also allows the implementation of categories and hierarchy of locations, as well as location name changes. Further specifications and the usage of tags allows us to annotate it with more information. It is possible to annotate a trip or parts of a trip to specify the transportation modes used.

```
--2016_01_09
0000-0705: home
0705-0712: home->bus stop close to home [walk]
0712-0734: bus stop close to home
0734-0801: bus stop close to home -> bus stop close to work [bus|bus:767]
0801-0811: bus stop close to work -> work [walk]
0811-1211: work
1211-1219: work->Le Bistro [walk]
1219-1331: Le Bistro [lunch]
1331-1350: Le Bistro->work [walk]
1350-1822: work
1822-1824: work->bus stop close to work [walk]
1824-1825: bus stop close to work
1825-1841: bus stop close to work->bus stop close to home [bus|bus:767]
1841-1848: bus stop close to home->home [walk]
1848-2359: home   [dinner]
```

Figure 3.3: Example of a day annotated with the *LIFE* format

LIFE files are plain text files, easily understandable by humans, as we can see in Figure 3.3. The specification was designed with that purpose, as it allows the flexibility of text editing to make changes, not locking users into obscure and not understandable formats.

3.2 Information Black Holes

When it comes to lifelogging one of the most important details of choosing applications to use is how can one use the data, long after an application has ceased to exist. Geographic lifelogging is no different. If an application ceases to exist or becomes unsupported, our data may be no longer be accessible. Or, if a new and better application is created to do the same functionalities, we need a way to move data between them.

This is a concern of ours. Our mantra when it comes to data is: exported by default. What this means is that, besides saving data to a server, we should keep the results and the original data as files

[2] Created by Daniel Gonçalves, `https://github.com/domiriel/LIFE`, last accessed on October 15th, 2016

accessible at any time. Tracks and trips should be exported as GPX files, while semantic annotations should be saved to a *LIFE* file. This way, if an application comes along that is better, users are not tied to our ecosystem and to our way of doing things. They can just take their data with them, without needing to export it. For us, personal data belong to their users.

3.3 All Together

An application that deals with geographic lifelogging has to take into consideration, besides usual usability concerns, all of the factors mentioned above. If one has to build such application, some of the requirements are:

- **Make personal GPS tracks understandable**

  As we have seen the complete story of a day, told by GPS tracks, can be hard to grasp. Applications to deal with GPS tracks must be able allow the user to understand that information as quickly as possible.

- **Deal with lack of accuracy and reliability from GPS recordings**

  There are two way to make this happen:

  - Automatic processing, cleaning tracks
  - Tools to edit tracks

    Tools such as those should allow to join two track, split a track, edit points individually, and to delete a track.

- **Support different workflows**

  Lifelogging is a personal experience, and a tool for lifelogging should adapt to an acceptable range of scenarios. If the goal is to process the trajectories between personally relevant places, workflows 1 and 3 must be supported.

- **Provide a way to easily annotate tracks**

  Annotated tracks make them personal to the user, telling one's day-to-day life.

- **Make personal information available to the user by default**

  User's information should be exported by default, allowing users the ability to switch between applications and to have access to their data at anytime.

- **Fast to process GPS tracks**

  Any application should do what it is supposed to do, in the fastest way possible. When it comes to lifelogging this aspect is amplified. Users already put important time into recording their lives, they processing of that recorded information should be a simple, straightforward process.

Chapter 4

TrackToTrip

In the previous section, we have seen the challenges that we face when dealing with GPS recordings, how consumer devices record an excessive and inconsistent amount of noise.

If we want to extract more and better information from raw GPS recordings, we have to overcome and correct GPS's lack of accuracy and reliability. With that in mind, we propose an automatic process to transform a **track into a trip**.

A **trajectory** is the ground truth. It is the true path taken by the user between two semantically relevant locations.

A **track**, as we have seen before, is a collection of points recorded during a trajectory. It is imprecise, because of the lack of accuracy of GPS devices, and it may lack some recordings caused, for instance, by signal loss or because the user forgot to start or stop the recording in the correct places.

A **trip** is an approximation to a trajectory. In our approach, a trip is derived from a track. To go from a **track to a trip**, we have to perform a set of actions. Our first plan considered the execution of four steps, as depicted in Figure 4.1. Those were segmentation, smoothing, simplification and map-matching.

The segmentation step consists of segmenting a track into multiple segments, where each segment represents a trip between two personally relevant locations. The smoothing step cleans the signal noise introduced by the GPS, whereas the simplification phase compresses a track and its segments, maintaining a similar topology and the same velocity between points. The map-matching step fits the track to a road network, further smoothing and compressing the track.

This plan (Figure 4.1), however, had problems. First, as we have seen back in Chapter 2, map-matching in not an easy thing to do. Privacy concerns rendered the usage of online tools — like *Google Maps* — to perform map-matching is not viable. We looked for open-source projects that perform map-matching locally. The only project found that aligned with our needs was a module, part of the *Graphhoper* [1] library.

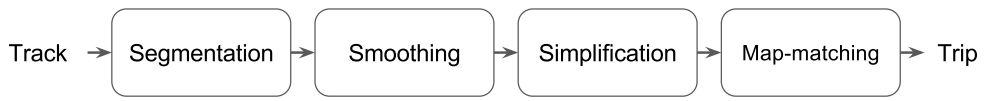


Figure 4.1: Initial conception of the steps necessary to process a track to a trip

The *Graphhopper* library uses Hidden Markov Model (HMM), based on [Newson & Krumm, 2009], to map-match GPS coordinates to a road network (provided by OpenStreetMaps (OSM)). This library produced good results for tracks that used road networks, but it failed in zones that did not have road networks, such as parks, indoor building or ill mapped zones. Also, it was biased towards cars, making

[1] `https://github.com/graphhopper/graphhopper`, last accessed on October 15th, 2016

weird matchings to respect transit flow. These problems led us to drop the map-matching step.

Another problem with our first plan to transform a track into a trip was the order in which steps are executed. We reached the conclusion that, we first needed to remove the noise from a track and only then segment it, because imprecise points may lead to bad segmentations, thus compromising the whole chain with bad results. With that in mind, we swapped the order of the simplification step with the segmentation step. We have maintained the compression step as the last one to delay data loss as much as possible.

Figure 4.2: Automatic processing to transform a track to a trip

Then, the steps needed to transform a track to a trip are (in this order): smoothing, segmentation, and compression (Figure 4.2).

Besides transforming a track into a trip, we can also extract personally relevant information about that track. More precisely, we want to extract personally relevant locations and transportation modes. To extract those, we have to introduce location and transportation mode learning steps.

As we have seen in Section 2.5, there is a dearth of projects and tools, to process GPS tracks, that do not introduce the user into the loop, which is the only way to fulfil personally specific needs. In our case, we introduce users into the loop by allowing them to validate and change locations and transportation modes. Furthermore, users should be able to correct any assumptions made by the track to trip process. Since we ask for simple validations and corrections that do not overload users with a lot of tasks, the system can keep updating its knowledge and, furthermore, improve the results of following predictions.

To help users annotate locations and transportation modes, we infer them first. If, for every trip, we learn its personally relevant locations, after a while, the suggestions will be better, to the point where users just need to accept the suggestions in most cases. The same goes for transportation modes; once they have adapt to the user, suggestions will be more personal.

One other aspect that we have found useful is to keep a global representation of the top trips made by the user — canonical trips. This representation allows the reconstruction missing parts of trips, by querying canonical trips between two points. This can only be possible if we learn every trip that has been processed and validated. Figure 4.3 shows all the steps of our system and the optimal flow, to (semi-)automatically process tracks into trips and to extract and infer information from them.

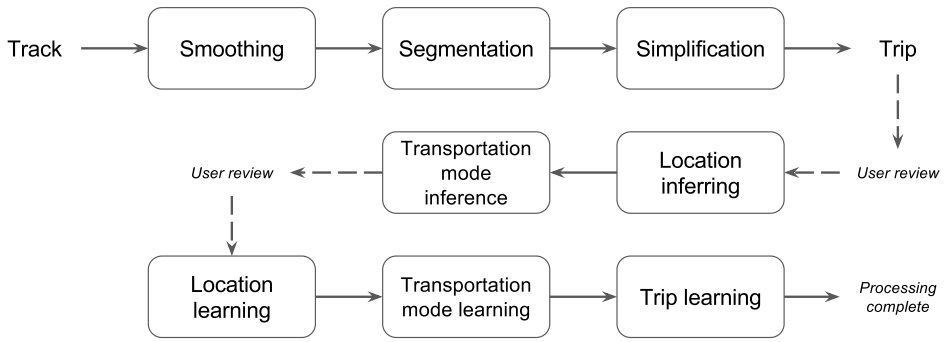


Figure 4.3: Steps to process a track

A purely automatic processing and information extraction can also be done by considering the trips, locations and transportation modes valid by default. Transportation mode and location learning, however, do not make sense in such case, as it causes self-feedback and bias. Albeit, the trip learning step would

still make sense to use. In the context of lifelogging, this may be useful when users want to process personal GPS tracks of common day.

To accommodate for all of these steps and processing, we have created the *TrackToTrip* library, which implements all of the explained steps and provides a baseline to work simply and efficiently with tracks. In the following sections, we will explain how and why we have implemented the steps described above.

We choose to developed *TrackToTrip* using *python*, because of its flexibility, because of the rich module system and because we were already familiar with it. At development, we tried to follow a simplistic approach, that conforms with *python*'s way of doing things. This way, it will be easier for anyone familiar with *python* to use the library and dive into the code changing it, or just understanding how it works.

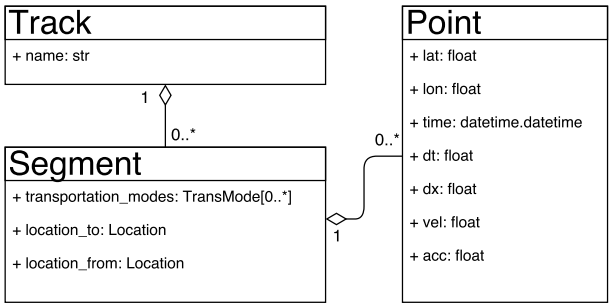


Figure 4.4: *TrackToTrip* simplified UML diagram

Figure 4.4 shows a simplified UML of the classes that we developed and use. We strove to keep classes light in both attributes and methods, mixing both object-oriented programming and functional programming. For instance, all the steps described above have their own separate functions that receive either *Track* or *Segment* instances. The *Track* class is the entry point to the system, where most functions are applied. Architecturally, a track has the same structure as a trip. They only differ in their semantic meaning.

4.1 Smoothing

Commercial GPS devices have low accuracy and reliability, as we have seen in Chapter 3. Recordings of the same location may result in different coordinates. To overcome this, we need to smooth the points recorded to have a better representation of the true trajectory, recorded in a GPS track.

Our first experiments consisted on a simplistic approach to smoothing: remove obvious noise points identified based on unrealistic accelerations. We experimented with various acceleration thresholds. From $5g$, after which humans start to feel disoriented, dizzy and faint [Voshell, 2004], to smaller accelerations, such as $0.5g$, which is the typical maximum acceleration of an electric motor vehicle with high acceleration, from its resting state. Because our domain is lifelogging and because the great majority of people experience low accelerations on a day-to-day basis, we assumed the $0.5g$ threshold value.

This approach, however, did not produce the desired results, not removing most noise points or removing more points than it should. This is because, although GPS readings of the same location approximate to the same place, GPS recordings of moving points do not offer reliable acceleration readings. We needed to find a method, that according to the characteristics of a track, could remove or approximate the points to the ground truth.

To solve the accuracy problem, we first looked at median and mean filters. Median and mean filters take a sliding window and apply, respectively, the median and mean functions to those points. These algorithms are fast, simple and lightweight, but not resistant to consecutive noise points.

Then, we looked at more sophisticated methods, such as Kalman [Goodwin & Sin, 2014] and Particle [Del Moral, 1996] Filters. Upon experimenting with a Kalman Filter implementation, *pykalman*[2], and reading [Zheng, 2015] about the benefits and drawbacks of using Kalman Filters against Particle Filters, we settled with the Kalman Filter method.

The *pykalman* library, in spite of being flexible, was too slow. We searched for other options and found *ikalman*[3]. *ikalman* is a Kalman Filter, implemented in the *C* programming language, intended to use with GPS data.

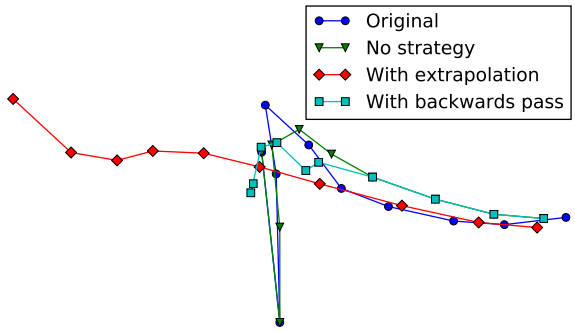
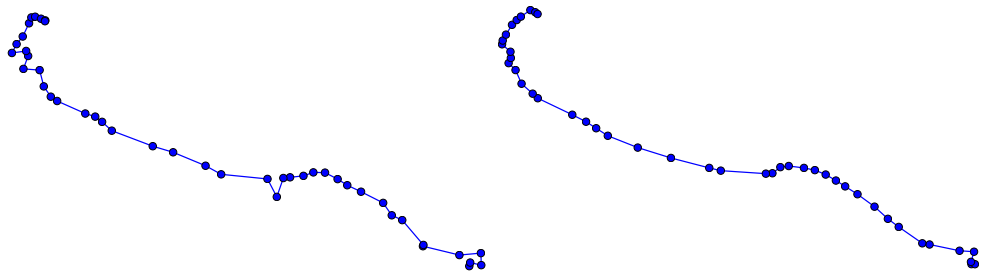


Figure 4.5: Comparison of strategies applied to improve Kalman Filter

Still, the Kalman Filter is imprecise in its first predictions. To accommodate for that, we came up with two methods. The first method, consists of applying the Kalman Filter twice. Once with the points in the normal order, then, once with the points in reversed order. The first part of the resulting track comes from the backwards pass and the second part from the forward pass. The middle point is averaged from the two passes. This method is slower but more accurate, because we have to apply the Kalman Filter twice, but noisy points at the start of a track can be adjusted more accurately. We call this method *Kalman Filter with backwards pass*.

The second method extrapolates linearly the first ten points at the beginning of the track, and after applying the Kalman Filter, removes them. This method is faster, but has less precision, since initial noise can affect the quality of the extrapolated points. We call this method *Kalman Filter with interpolation*.

Figure 4.5 shows an edge case, where the second point of the original tracks is noisy. The *Kalman Filter with backwards pass* is capable of reducing the noise of that point considerably, while the Kalman Filter with no strategy is incapable of reducing the noise. The *Kalman Filter with interpolation* is severely influenced by the noise point at the beginning, since the extrapolations are not effective.



(a) Raw track (b) Kalman with backwards pass

Figure 4.6: Results of smoothing with Kalman Filter

[2] `https://pykalman.github.io/`, last accessed on October 15th, 2016
[3] `https://github.com/lacker/ikalman`, last accessed on October 15th, 2016

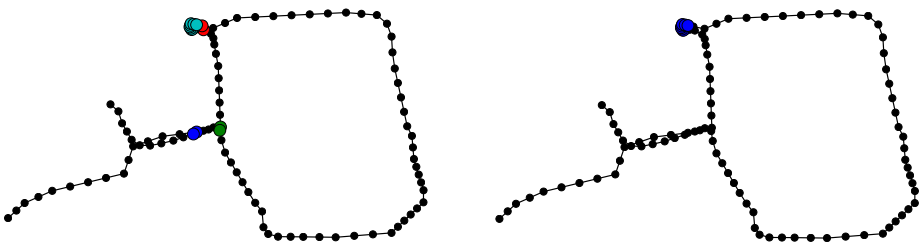
The track on Figure 4.6a is smoothed using the *Kalman Filter with the backward pass*. The result, in Figure 4.6b, is a smoother track where spikes and imprecisions are removed. This will lead to a better segmentation and compression later on in the processing chain.

4.2 Segmentation

The segmentation step is used to split a track into one or more track segments. A simple, yet effective method, is to split a track when there is a time distance between two points higher than a given threshold (such as 60 seconds). This was our first approach to the problem. Since some applications record multiple trajectories into a single track, this allowed us to separate a track that contained multiple trajectories.

This simple approach, although useful, was not sufficient. It works best when users start and stop recording at the right time, often at the right place. While there are cases in this scenario where this method works, because the user is indoors and has lost the signal, there are cases where the user forgets to stop and then start recording and keeps getting GPS signal. In those cases, this method fails to split a track. To identify those places, we have to determine if and where the user has spent a sufficient amount of time during a trajectory.

This is a clustering problem. By identifying clusters of points, we are identifying locations where users recorded more points, thus where they spent more time. Spatial point clustering, however, is not good enough. We have to include the temporal component of each point to prevent clusters when, for instance, there are multiple and brief visits to the same location, at distinct times of the day. This problem is exemplified in Figure 4.7. Figure 4.7b identifies the cluster correctly, while Figure 4.7a identifies other, undesirable, clusters.



(a) With spatial component (b) With spatiotemporal components

Figure 4.7: Clustering a track, using different components

Upon research, we considered the *k-means* [Hartigan & Wong, 1979], *Mean Shift* [Cheng, 1995] and *DBSCAN* [Ester *et al.* , 1996] algorithms.

From those, *DBSCAN* is the most suitable for our purposes, since it works best at identifying areas of high density (locations) separated by areas of low density (path between locations), and contrary to *k-means*, we do not need to know the number of clusters in advance. *DBSCAN* clusters can also have any shape, as opposed to *k-means*, which considers convex clusters. *DBSCAN* also allows us to define user understandable parameters. It takes *eps* and *min samples*, which means that there are at least *min samples*, within a distance lower than *eps*.

A core difference and deciding factor that made us choose *DBSCAN* over *Mean Shift*, was that *Mean Shift* considers distances between points, while *DBSCAN* considers distances between nearest points. This means that *DBSCAN* is better suited to deal with noise around a certain location, which we encounter when dealing with GPS data — as we have seen the lack of accuracy, both by the user,

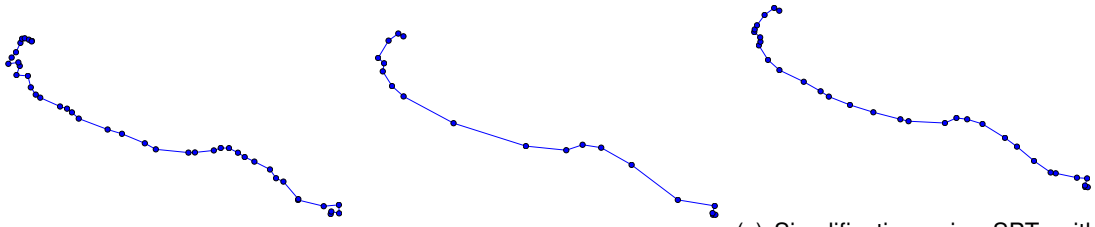
in starting and stopping the recording at the right time, and by the GPS itself, as it lacks accuracy. For those reasons, we use the *DBSCAN* algorithm, implemented by *scikit-learn* [4] package.

4.3 Simplify

In the simplification step, we aim to compress data, removing only redundant information. Simplistic approaches to compress GPS tracks may use the RDP [Douglas & Peucker, 1973] algorithm or other line simplification algorithms. Such approaches, however, will lead to losing temporal characteristics of a track. Our goal was to maintain temporal and spatial consistency.

Our first approach was to use *Where Have I Been* [Filipe, 2015] simplification algorithm, which at its core, is a time aware RDP algorithm.

Further research led us to Top-Down Speed-Based Trajectory Compression Algorithm (TDSP), Top-Down Time-Ratio Trajectory Compression Algorithm (TDTR) and a combination of both, Spatiotemporal Trajectory Compression Algorithm (SPT) [Meratnia & Rolf, 2004]. Those algorithms define, respectively, the parameters: maximum distance error (in meters), maximum speed error (in km/h) and both.



(a) Raw track

(b) Simplification using RDP

(c) Simplification using SPT, with max distance error of 2 meters and max velocity error of 1.0m/s

Figure 4.8: Different compression methods

Figure 4.8 shows two different compression methods for the track smoothed in Section 4.1 (Figure 4.6b). From the Figure 4.8b we can see that the RDP algorithm is good enough for topology compression of tracks, but the temporal characteristics of the track are distorted.

For instance, the average velocity from the original track (Figure 4.6b) is 2.3 km/h, the average velocity after applying the RDP algorithm (Figure 4.8b) is 1.8 km/h, while the SPT algorithm (Figure 4.8c) produces a track with an average velocity of 2.25 km/h. When it comes to the actual compression ratio, the RDP algorithm did better, reducing the 44 points down to 19, a compression ratio of 232%, while the SPT algorithm reduced the number of points down to 32, a compression ratio of 84%.

After smoothing, segmenting and compressing, each track becomes one or more trips, where the start and end points are semantically relevant locations.

4.4 Location learning and inference

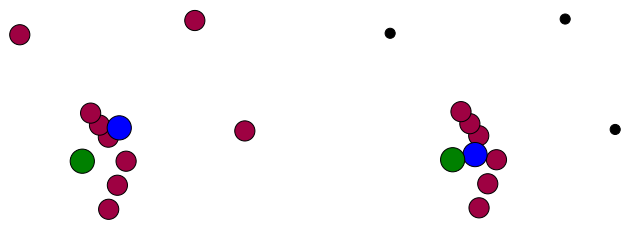
As we have seen, trips contain valuable information such as semantically relevant locations. These locations can be extracted from the start and end points of a trip. To help users annotate those locations, we can learn with their input, refining our knowledge about them, to later offer location suggestions for geographical points. In the lifelogging domain, as we have seen, most of the time, users use the same trajectories in their day-to-day lives. The same principle applies to locations: users have a relatively small set of locations that they visit frequently.

[4] `http://scikit-learn.org/stable/`, last accessed October 15th, 2016

The first challenge is to learn the set of locations and their geographic coordinates. In an initial and simplistic approach, we allowed the user to define a fixed set of locations – canonical locations. With those locations defined, we could check which of those distances was closer and try to predict the name of the location. Although this approach is good enough for certain cases, it starts to fall short when dealing with geographic information. The lack of accuracy of consumer GPS devices makes this simplistic method of inferring a location's name imprecise.

We needed a way to learn and predict the name of a location, given a geographic point. We looked at the literature, specifically at [Kirmse *et al.* , 2011] already discussed in Section 2.2.2, and decided to use Leader-Based Clustering (LBC). In this strategy, we form clusters of known locations and compute their centroids. To learn a new location, we search for the closest location centroid and add the new location to the cluster, computing the new centroid. If a cluster is not close enough (for instance within 20 meters), we create a new location cluster.

To infer the location of a given point, we use a similar method: find the closest centroids to a point. This method is similar to the simplistic approach, with one caveat: this method keeps adapting to the user. At every iteration, the centroid tends to get closer and closer to the true semantically relevant location, which in turn, produces better results over time.



(a) Without outlier removal (b) With outlier removal

Figure 4.9: Centroid computation for a location (centroid in blue, ground truth in green)

This approximation to the true location, however, can be compromised by outliers. Figure 4.9a shows the computed centroid for a location using all given points. We can see that the centroid's position is far from the correct geographic location. To compute the centroid, without taking into account outliers, we use the *DBSCAN* algorithm, where we look for the biggest cluster. The *DBSCAN* algorithm, besides clusters, also identifies outliers which can be discarded when computing the cluster centroid. Figure 4.9b shows the result of using *DBSCAN*. We can see that the computed centroid is closer to the ground truth.

4.5 Transportation mode inference & learning

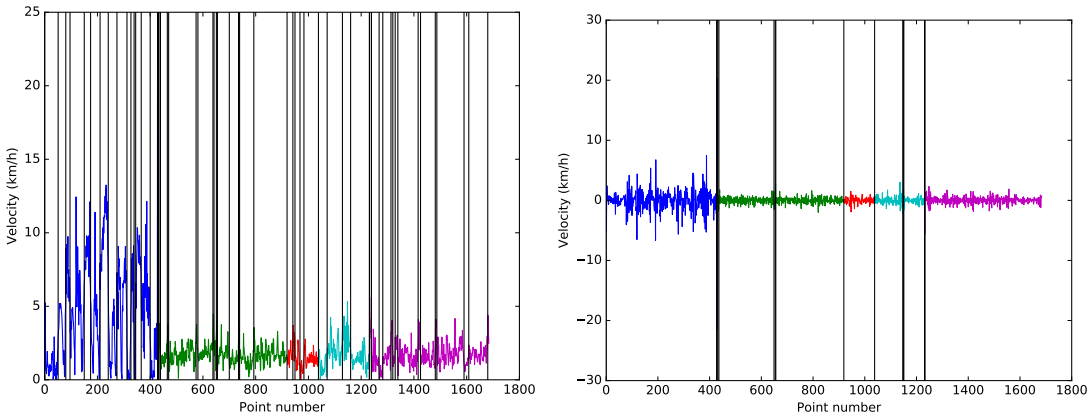
Transportation mode inference is a hard, but explored problem over the years. Projects such as *Geo-Life* [Zheng *et al.* , 2008d] and *MOPSI* [Waga *et al.* , 2013b] used the *GPS* signal to infer transportation modes, whereas *SenseMe* [Bhargava *et al.* , 2014b] used *GPS* and other sensor data. In our case, and since we only have access to tracks (that become trips), we use the *GPS* signal to infer transportation modes.

Before inferring transportation mode, we have to consider that a trip is often composed of more than one transportation mode. Then, in our trip, the identification of the places where the transportation mode has changed is the first step.

We approach this problem using a popular concept in *time-series* analysis: change point segmentation. *Binary Segmentation* [Scott & Knott, 1974] and *Pruned Exact Linear Time (PELT)* [Killick *et al.* , 2012] algorithms are popular methods and were the most promising. Between the two, we opted for the *PELT* algorithm. The deciding factor was that the PELT algorithm was exact, compared with the *Binary Segmentation* algorithm. We also experimented with simpler methods, such as detecting change points based on velocity mean and variance, but without success.

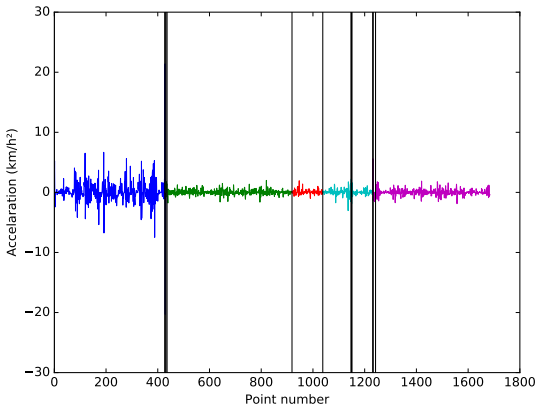
For us, one obstacle about change point segmentation algorithms was the lack of viable implementations. The only public, license compatible, implementation was an *R* package *changepoint*[5] and a port of that package in the *Julia* programming language, *changepoints*[6]. We had to port the *PELT* algorithm to *python*. As yet another, contribution to the community, we released it as a standalone *pypi package*[7].

The goal of the *PELT* algorithm is to minimize a cost function. In our case, and after experimentation, we choose a cost function that expects normally distributed data, with fixed mean and variable variance.

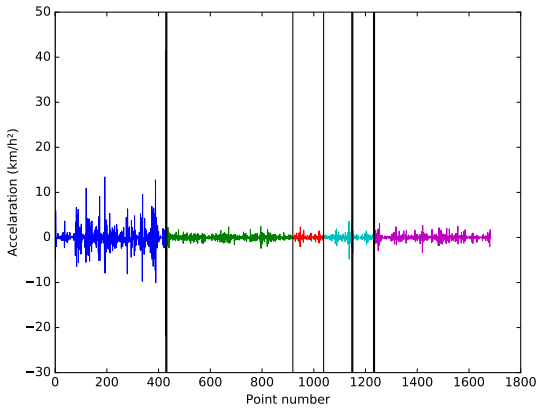


(a) Using velocity, 57 change points

(b) Using velocity difference, 15 change points



(c) Using acceleration, 13 change points



(d) Using acceleration difference, 12 change points

Figure 4.10: Different data applied to the change point algorithm

We also explored which data to feed the *PELT* algorithm and which cost function to use. Our first iterations used track velocities. Figure 4.10 shows the identified change points for each type of data, the only change point occurred around point number 400. In Figure 4.10a, 53 change points were identified, an unrealistic number. This method is too simple, and lacks accuracy. Other approaches, such as velocity difference between consecutive points (Figure 4.10b) or using point accelerations (Figure 4.10c, yield better results.

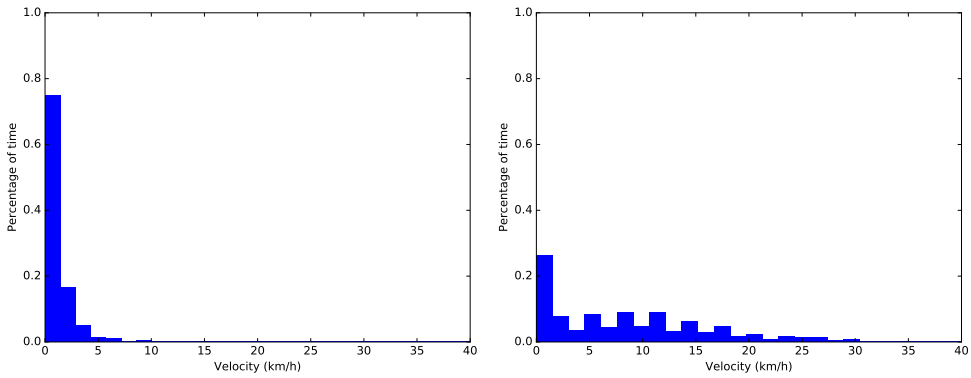
[5] `https://cran.r-project.org/web/packages/changepoint/index.html`, last accessed on October 15th, 2016
[6] `https://github.com/STOR-i/Changepoints.jl/`, last accessed on October 15th, 2016
[7] `https://github.com/ruipgil/changepy`, last accessed on October 15th, 2016

Based on the results (Figure 4.10), we choose to use the *PELT* algorithm using acceleration difference, shown in Figure 4.10d. We can see that it is close to the ground truth, while producing less consecutive points, compared with the other methods. With change points defined we can classify a trip between each change point.

To do that, we use a Support Vector Machine (SVM) [Cortes & Vapnik, 1995], most specifically a Stochastic Gradient Descent (SDG) [Bottou, 1998] classifier, provided by the *sklearn* [8] library. This method of classification was already explored in other projects, such as *SenseMe* [Bhargava *et al.* , 2014b], with some of its drawbacks and benefits already documented. The deciding factor between this classifier and a Decision Tree classifier — another popular classification method — was the ability of the SDG classifier to do dynamic learning.

Dynamic learning allows for the learning of transportation modes that were not used in training, allowing us to keep refining the classifier, adapting further predictions to the user patterns.

We wanted to explore new approaches when selecting features. One method that, to our knowledge, was not mentioned and, intuitively, should give good results was getting the top speeds and the relative amount of time spent at those speeds. We also could not use some feature extraction methods, such as in *GeoLife* [Zheng *et al.* , 2008d], since at this stage we are working with a trip, which is compressed. This, however, is not a problem with the way we extract features, as we look for the relative time at certain speeds. Since we extract a relative profile of the velocities of a trip, it does not matter if a trip is compressed or not, as long as there is minimal information loss after compression (which is our case).



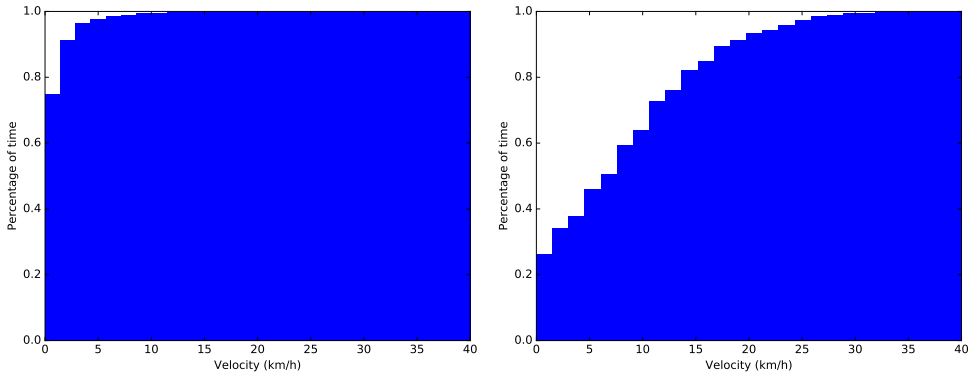
(a) Walk (b) Car

Figure 4.11: Average velocity histograms from GeoLife dataset

In our first experiments of feature extraction, we created a histogram of each sub-trip, where each bin index represented the velocity rounded to the unit and the value of each bin was the amount of time spent at the corresponding velocity. The histogram was then normalized, and the top N velocities extracted, as well as the relative amount of time spent at them, as shown in Figure 4.11. This method, however, proved to be imprecise. Achieving an *f-score* of just 60%, barely better than random guessing.

Deciding not to give up on histograms, we looked at what cumulative histograms (Figure 4.12) looked like. What we found was much more enlightening; each transportation mode had a more defined and distinctive layout. With the cumulative histograms, we can check what ranges of velocities contribute the most to a certain trip. For instance, we can check at what velocity a segment spends more than 50% of the time. What we ended up extracting was the velocity (bin number) at which we spend a certain relative amount of time. In our case, we use nine features, therefore nine thresholds to extract the velocity. They are: 10, 20, 30, 40, 50, 60, 70, 80, and 90%.

[8] `http://scikit-learn.org/`, last accessed on October 15th, 2016



(a) Walk (b) Car

Figure 4.12: Average cumulative velocity histograms from GeoLife dataset

As an example, using the profile of *walk* transportation mode in Figure 4.12a, a feature array can be extracted by checking at which velocity it spends more than 10% of the time, which is at or below 0.49km/h (rounds to zero). The same is done for the other percentages described above, and the resulting feature array is: $[0, 0, 0, 1, 1, 1, 1, 2, 2]$. For a car, from the histogram in Figure 4.12b, the resulting array feature array is: $[0, 0, 1, 5, 6, 7, 8, 9, 13]$.

We use the *GeoLife Trajectory* [9] dataset to train and evaluate our classifier. The *GeoLife Trajectory* dataset provides data for *run*, *walk*, *airplane*, *train*, *subway*, *taxi*, *car*, *vehicle*, *bus*, *motorcycle*, *bike* and *boat*.

To avoid misclassification between similar transportation methods, we have grouped them. We grouped *taxi*, *bus*, *motorcycle* and *car* into the *vehicle* label; *train* and *subway* to *train* label; and *run* and *walk* into the *foot* label. Due to lack of samples, we have discarded *boat* and *bike* samples.

With this method of classification, we achieve values similar to of the projects talked about in Chapter 2. Using the *GeoLife Trajectory* dataset, we perform two-fold validation to evaluate the performance of the classifier. We obtain an average *f-score* of 84.6%. For the sake of comparison, we trained and evaluated an Iterative Dichotomiser 3 (ID3) Decision Tree classifier and obtained an average *f-score* of 83.3%. Further details about this evaluation, such as the precision for each transportation mode, can be found in Appendix A.

4.6 Trip learning

Since humans are creatures of habit, in our day-to-day lives we use — almost always — the same trajectories, even if with subtle changes. When recording GPS trips, signal loss tends to occur, and sometimes recording starts or ends when they should not. Because our system has a personal focus, if we learn our most common trips, we can suggest possible completions to complete trips and track.

To efficiently and effectively learn trips, we needed to create and update paths that topologically represent previous trips, while avoiding similar paths to create new representations. Instead, they should update existing representations. After some research, nothing was found that specifically suited our needs.

The closest that we found was [Wei *et al.* , 2012], that proposed to represent trips based on road networks. This was not enough for us, because it would be too limiting when used in a city park, when hiking, or for that matter, anywhere that road maps are unavailable or not possible.

[9] `https://www.microsoft.com/en-us/download/details.aspx?id=52367`, last accessed on October 15th, 2016

We started to think of this problem as a line comparison algorithm. We needed to compare two tracks that looked similar, but with slight deviations, because of the lack of accuracy when recording GPS signals. We then found out about the Trajectory-Hausdorff Distance (THD) [Lee *et al.* , 2007] algorithm, which allows us to compute how far two trajectories are.

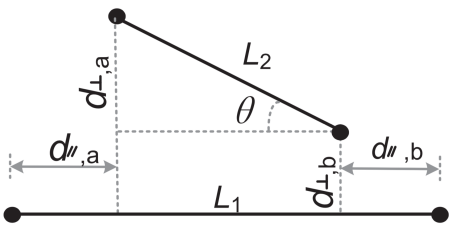


Figure 4.13: Trajectory-Hausdorff Distance

The THD is a three part, weighted sum (Equation 4.1). Figure 4.13, shows the considered parameters to compare two lines. It takes into account the parallel distance (Eq.4.3), the perpendicular distance (Eq.4.2) and the angular distance (Eq.4.4). Furthermore, the weights, w_1, w_2, and w_3 are dependent on the application. The THD was a good start, but it did not fit all of our needs.

$$\mathrm{THD}(L_1, L_2) = w_1 d_\theta(L_1, L_2) + w_2 d_\parallel(L_1, L_2) + w_3 d_\perp(L_1, L_2) \tag{4.1}$$

$$d_\perp(L_1, L_2) = \frac{d_{\perp,a}^2 + d_{\perp,b}^2}{d_{\perp,a} + d_{\perp,b}} \tag{4.2}$$

$$d_\parallel(L_1, L_2) = \min(d_{\parallel,a}, d_{\parallel,b}) \tag{4.3}$$

$$d_\theta(L_1, L_2) = ||L_2|| \cdot \sin\theta \tag{4.4}$$

First of all, we wanted to obtain a ratio of similarity between zero and one, where zero means that the tracks are not similar and one means that they are equal. Second of all, and most importantly, if a smaller track was contained in another bigger track, we wanted them to be considered similar. Inspired by the THD, we created the Trajectory-Hausdorff Ratio (THR) (Equation 4.5 with pseudocode in Appendix B), which allows us to compare how similar two tracks/trips are to each other, where the track/trip that is serving for comparison has equal or greater length, than the other.

$$\mathrm{THR}(L_1, L_2, d_{threshold}) = d_\theta(L_1, L_2) * d_\parallel(L_1, L_2, d_{threshold}) \tag{4.5}$$

With the THD as base, we have made small modifications to fit our needs. First, instead of using a weighted sum, we use multiplication. We also do not consider the perpendicular distance. Then, we changed the distance functions to better fit our needs, returning a ration (from zero to one) of how similar they are.

The angular distance function (Equation 4.6), receives two lines that are normalized and the dot product is applied to them.

$$d_\theta(L_1, L_2) = |\frac{L_1}{||L_1||} \cdot \frac{L_2}{||L_2||}| \tag{4.6}$$

The parallel distance function (Equation 4.7), receives two lines and a distance threshold; distances above that threshold yield a ratio of zero. To compute the ratios, for the start and end points of a line

segment, we compute the distance between that point and the closest point on the other line segment (Equation 4.8). That distance is then transformed into a ratio, so that it returns zero if the distance is bigger than the threshold.

$$d_{\|}(L_1, L_2, d_{threshold}) = \frac{d_{lineratio}(L_1, L_{2_{startpoint}}, d_{threshold}) + d_{lineratio}(L_1, L_{2_{endpoint}}, d_{threshold})}{2} \tag{4.7}$$

The point-to-line distance ratio (Equation 4.8) returns one if a point is over a line, or zero if it is farther than the threshold. The decrease in ratio, from one to zero is linear, but one could change the function to have more tolerance.

$$d_{lineratio}(L, p, d_{threshold}) = \max(0, \frac{-1}{d_{threshold}} * |d_{line}(L, p)| + 1) \tag{4.8}$$

The THR is capable of comparing two lines segments. But a trip is composed of points. To score two trips, we consider a line segment between each consecutive point and then compare all the segments of the bigger trip to the overlapping segments of the smaller trip. Also, each segment in the bigger trip may be overlapped by more than one segment of the smaller trip. With that in mind, we compute the similarity for all of the smaller trip segments, that are overlapping the bigger trips, and extract the maximum score. In the end, we average the scores for each segment.

To speed up the matching of line segments that overlap each other, we compute Minimum Bounding Rectangles (MBR) that is two times bigger and has a minimum width and height, to avoid MBR with near zero width or height. The MBRs of the bigger trip are added to a *rtree* [Guttman, 1984], which is then queried by the MBRs of the smaller trip.

(a) Trips not similar (ratio of 0.6) (b) Similar trips (ratio of 0.82) (c) Trips merged

Figure 4.14: Similarity between trips

Through our experiments, we consider two trips similar if their similarity score is equal or greater than 0.8. If two trips are not considered similar (such as in Figure 4.14b), according to what we have seen previously, we store a RDP simplified version of the trip. We simplify trips, to speed up future similarity comparisons and because we care about the topology, not about the time characteristics of a trip. If a trip is deemed similar, then we update the learned trips, by merging and reordering the two and, again, simplifying them with RDP.

To merge and reorder, we use the points of both trips and, by starting at the end of the new trip, add the closest point of the two, which is removed. We do this until there are no more points left. The problem with our technique is that it sometime creates undesirable tangles. Since this approach gave good enough results, we did not think it was necessary to further improve the ordering algorithm. Although, this is a similar problem to [Brunsdon, 2007], as seen in Section 2.4.1, which consists in fitting a line on a point cloud, the result is presented in Figure 4.14c.

Chapter 5

ProcessMySteps

ProcessMySteps is our backend module, responsible for managing the processing of tracks and controlling the flow of the application.

Right from the start, we organized our work within a client-server architecture (Figure 5.1), where the client (*GatherMySteps*) is a web application and the server is composed of *ProcessMySteps*. The database could be deployed in the same machine as *ProcessMySteps* or it could be configured to be deployed remotely.

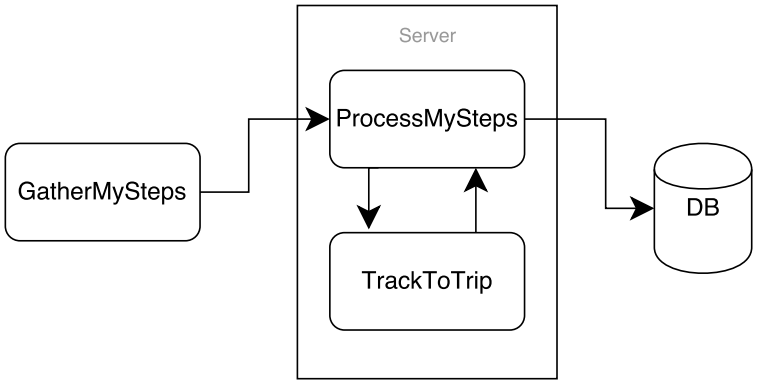


Figure 5.1: High-level architecture of our system

The *ProcessMySteps* modules can be executed from the command line. It spawns a server that exposes a REST API, reads the GPS tracks to be processed and connects with the database.

The server can be configured through the web interface or — at startup — by a JavaScript Object Notation (JSON) file, that defines the folder to search for GPX files, the output folders, database credentials, as well as a myriad of other parameters, such as parameters to fine tune the *TrackToTrip* library (see Appendix C). When it is initialized, or when requested by the user through the UI, a list of available tracks is loaded.

Only the tracks that belong to the day that is being processed are loaded in memory. In our first approach, we loaded all tracks at once. We loaded each and every file and grouped them by start date, which created a significant delay until a day could be processed. To resolve this problem, we used regular expressions to check the first date found on a file, without needing to parse and create the necessary structures for a track. Once this solution was in place, we only loaded the tracks that are going to be processed and the startup time became negligible.

ProcessMySteps exposes a simple REST API with 16 endpoints. To implement our endpoints, we

use the *Flask*[1] web framework. *Flask* allows for simple, yet powerful implementations of web servers and web APIs; with just a few lines of code, it is possible to receive and respond to API calls. Its extensive documentation and popularity are also good bonuses. Right from the start we aimed for simple, yet understandable API endpoints. Some of the most important are:

- `/previous` and `/next`: which receive a track/trip and applies the required processing and actions for each step, and returns the state of the new step;
- `/current`: responds with the state of the current step;
- `/completeTrip`: receives two points and returns a set of trips, that have been learned, and that pass between those points;
- `/location`: receives geographic coordinates, latitude and longitude and responds with possible locations for those coordinates (either learned locations, or from *Google Maps* [2], or *Foursquare API* [3]);
- `/transportation`: receives two points from a trip in the current state and predicts the transportation modes.

We also use request methods according with their function and side effect. Endpoints that have side effects are only accessible through `POST` method, for instance; to update a configuration, a call needs to be made to `POST:/config`. Endpoints that have no side effects are available through `GET` methods; for instance the current configuration of the system is available on `GET:/config`.

Since our application is to be used in a personal context, we have chosen to have a single state server. This means that there can only be one user at the time and the state is the continuous. This simplified development, as there were no concerns regarding data isolation, or user login.

To exchange information, both client and server use JSON formatted messages. At the beginning, we also had to decide what kind of client we wanted: a thin or fat client. We opted for a fat client, because current web browsers are more than capable of executing what we had planned, such as track editing: moving, adding or removing points or tracks. This option also kept the *ProcessMySteps* module simpler and with unnecessary information exchange.

Alongside our goal of usability, we also had an important consideration for the problems of a lifelogger. One of those problems is information black holes. It is incredibly frustrating when personal data, data that recorded and represents our lives becomes trapped inside an application that has lost support, functionality or utility. With that in mind, besides using a database to store data, we also save the processed tracks, backup the original tracks and save the semantic information.

The database is powered by *PostgreSQL* and *PostGIS*. In fact, we have chosen *PostgreSQL* because of *PostGIS*, as it is the most popular and powerful geographic extension for open-sourced databases. It allows us to store the building blocks of our system: tracks, and also allows us to perform powerful and complex queries, such as knowing places within a certain distance, all through SQL queries.

To help us communicate with the database, we use the python libraries: *psycopg2*[4] and *ppygis*[5]. They allow a great deal of flexibility, and avoid boilerplate code, such as type conversions and connections with the database.

The data model used by our server is represented in Figure 5.2. Until it reached its final version we made several changes. For instance, at the start of this whole process we considered a location

[1] `http://flask.pocoo.org/`, last accessed October 15th, 2016
[2] `https://developers.google.com/places/`, last accessed on October 15th, 2016
[3] `https://developer.foursquare.com/`, last accessed on October 15th, 2016
[4] `http://initd.org/psycopg/`, last accessed October 15th
[5] `http://www.fabianowski.eu/projects/ppygis/`, last accessed October 15th

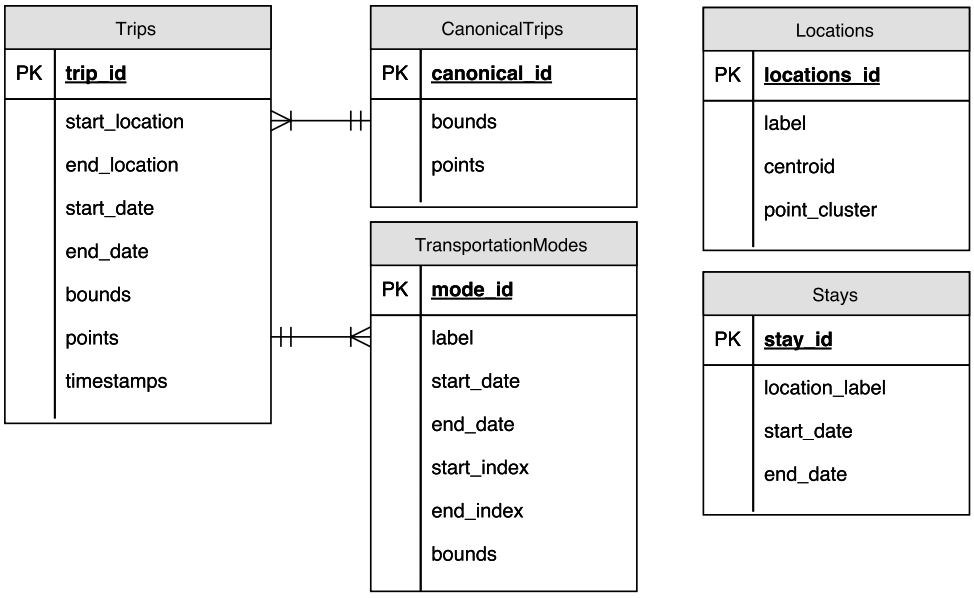


Figure 5.2: Entity-Relation — using crow's foot notation — of the data model for the database

name to be unique. As we have figured out, in a personal context, locations with the same name do not necessarily refer to the same geographic location.

Besides the database, the *LIFE* representation of one's life also needs to be updated. To update the *LIFE* file we append the annotation made by the user to the current *LIFE* representation. It is from that annotation that we also extract the names of places (to update the database) and transportation modes. We do that by matching the time (of trips and stays) in *LIFE* annotations, to the time of the points in the track.

5.1 Flow outline

The *ProcessMySteps* module imports the *TrackToTrip* library. The processing is divided throughout three stages (discussed in more detail on Chapter 6): preview, adjust and annotate. All throughout, *ProcessMySteps* controls the processing in different ways, as we can see in Figure 5.3. In the preview stage, the server sends the client its current state. The current state is comprised of the tracks (or trips) and their points, the current stage of processing and the days available for processing.

By advancing, the client calls the `continue` endpoint sending tracks to the server. These tracks may be the ones that were sent previously, or they may have been altered by the user. Those tracks are then fed to the *TrackToTrip* library and are transformed into trips. Those trips are sent back to client. After making changes or accepting the trips, users can proceed. Again, the trips are sent back to the server.

At this stage, we use the *TrackToTrip* library to infer the semantically relevant locations and transportation modes of each trip. This information is put together as a *LIFE* formatted annotation and sent back to the client. At this point, in the client, users can annotate their days.

To help with the annotation process, the client is able to request suggestions regarding locations and transportation modes. In the first case, it sends the coordinates of the point to infer the location, and a list of possible locations, ordered by relevance, is returned. In the second case, the client sends an array of points that is fed into the transportation mode classifier.

When the annotation is finished and the user clicks to save the current annotation, we save all the information (tracks, trips and *LIFE* annotations) in the database and in a *LIFE* file. The final step is to learn the new transportation modes, before loading another set of tracks.

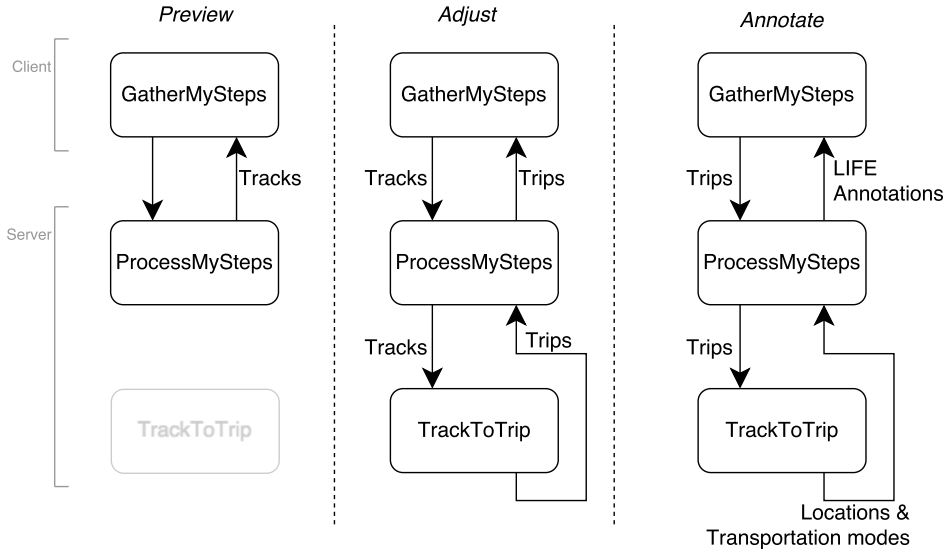


Figure 5.3: High-level diagram of the management made by *ProcessMySteps*

In addition to the current set of tracks and trips, we keep a history of the state of the system at each stage. This allows users to unwind to a previous stage. We always save the last tracks and trips sent by the client. This means that if users want to go back to a stage where they changed tracks, those modifications are not lost. This frees users to experiment and err without consequences.

Chapter 6

GatherMySteps

Our main goal was to have a fast and intuitive User Interface (UI) to allow for the processing and annotation of personal GPS tracks. Back in Section 3.3, we have seen what an application that aims to process personal GPS tracks must provide; most of those requirements are related with the UI.

One of the problems that we face with geographic information is how to make sense of it. Often, the best way to grasp what one has done, using GPS tracks, is to draw those tracks onto a map. It was with that intention that we decided to make the map our focal point, always visible and reactive to the actions of the user.

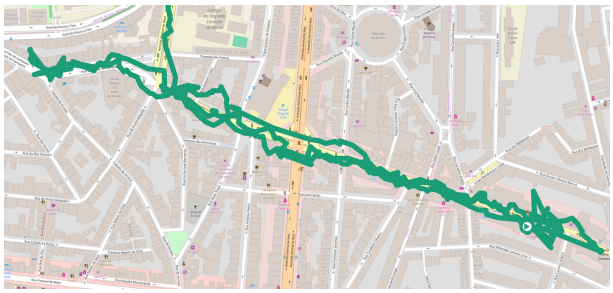


Figure 6.1: Raw GPS track

Still, GPS tracks, as they were recorded are hard to understand, even if with a map. This is shown in Figure 6.1. One can roughly see the trajectories, but it is hard to pinpoint at what time we left one place and to where. An option could be to put the user to work, but within the lifelogging domain it is not feasible. Users already spent their time and energy recording their trips.

Besides showing tracks, we use the *TrackToTrip* library to help us transform tracks into trips. This way the cleaning process is automatic; the user is left to decide if the decisions made were good enough. Another result of the automatic processing phase is that tracks — now trips — become easier to understand.

A map, even though essential, should not be the only way to present the information to the user. Users may be seeing a region of the map with some, but not all tracks, and those not visible might never be analyzed by them. It is important to list the tracks being processed, alongside their geographic representation. This also provides us with a place to show times and metrics (such as duration and distance), as well as actions, without cluttering the map or hiding information behind menus.

These are all components of the track editor. A *GatherMySteps* screen dedicated to the editing of GPS tracks and trips. It allows to inspect individual track points, remove tracks, join two temporally continuous tracks, split a track into two, edit track points individually (add, move and remove) and filter points by time.

GatherMySteps is dedicated to the lifelogging domain; as such the amount of tracks to process may range from days to weeks, or months. It would be burdensome to clean and annotate those tracks, which may contain thousands of trajectories. To simplify this process and provide the user with a moment to reflect upon the entire day, instead of a single trajectory, we group tracks by days. This way, users have an easier time recollecting their personal trajectories and what they did those days.

The processing of GPS tracks culminates by annotating each track, or in our specific case, by annotating the entire day. To do this, we have developed a semantic editor. Besides the interactive map, where the trips of the day are shown, the user is presented with a *LIFE* editor, where users can annotate their days. This editor is a text editor that is able to provide syntax highlight and relevant suggestions (which are retrieved from the *ProcessMySteps* module, using *TrackToTrip*). It works in such a way that it is possible to change locations and transportation modes like a drop-down form.

Figure 6.2: Three stages of processing

As shown in Figure 6.2, the processing is divided into three stages: preview, adjust and annotate. The *ProcessMySteps* module is responsible for coordinating the stages, the automatic process and the suggestions. In the preview stage, users can view the raw GPS tracks. In the adjust stage, tracks have become trips and users can make corrections or adjustments to those. While in the annotate stage, users annotate their days.

During development, we realized that the editor screen was a useful tool by itself, even outside *GatherMySteps*. With that in mind, we forked it as a parallel project and published it online [1].

The rest of this Chapter is dedicated to explain *GatherMySteps*. In Section 6.1, we present some the challenges and problems when developing the UI and all its components. Section 6.2 details the usage of *GatherMySteps* and how it fits most geographic lifelogging needs.

6.1 Implementation

As we have seen in Chapter 5, *GatherMySteps* is the client in our client-server architecture. We choose to implement the client as a web application, using HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and the JavaScript (JS) stack. We opted for a web application, instead of a desktop or mobile application because — although we only aim to support the desktop environment — web applications can run, with little change, in desktops, smartphones, or tablets. There is also the added benefit of being able to use the rich pool of libraries to speed up development, and to keep our attention focused on making the best UI possible.

We settled with the **preview**, **adjust** and **annotate** stages after some iterations. We started by outlining what the user should be able to do: make corrections to the automatic processing phase, complete trips where there is missing data and annotate transportation modes and locations. The result is represented in Figure 6.3.

Figure 6.3: Initial set of stages to process tracks

[1] `http://ruipgil.com/GatherMySteps`, last accessed on October 15th, 2016

After some coordination we realized that our concern to offer an efficient way to process personal GPS tracks required too many steps, and subsequently too many clicks. When looking for ways to reduce the number of steps, we realized that we had two stages dedicated to annotate transportation modes. We could reduce those two stages into one annotate stage, where users could annotate the locations and mark the transportation modes.

Furthermore, we realized that we could include trip completion into the adjust stage. Using the join functionality, we could offer suggestions similar to those planned in the completion stage. This way, users would only need to go through two stages to process a day's worth of tracks. This was what we were looking for, a simple, striped down approach to process personal GPS tracks.

There was a problem though. How could we be sure that the automatic processing phase was not doing something wrong? Or, what if the user could make some changes to improve the decisions of the automatic processing phase? The answer to our questions was to introduce a preview step, that would show the raw GPS recordings, like the adjust stage showed trips. By doing so, users can use the preview stage to proof-check the automatic processing phase.

Before moving to implementation, we also needed to decide what *GatherMySteps* would do and what the server (*ProcessMySteps*) would be responsible for. After some initial experimenting, and given how powerful and full fledged browsers are nowadays, we decide to keep the server as lean as possible — orchestrating the control flow of our application — while *GatherMySteps* was responsible for editing tracks without needing to communicate with the server. This way, the UI only asks the server for the current day and respective tracks (and the current server state), sending to the server the modified tracks, trips and annotations.

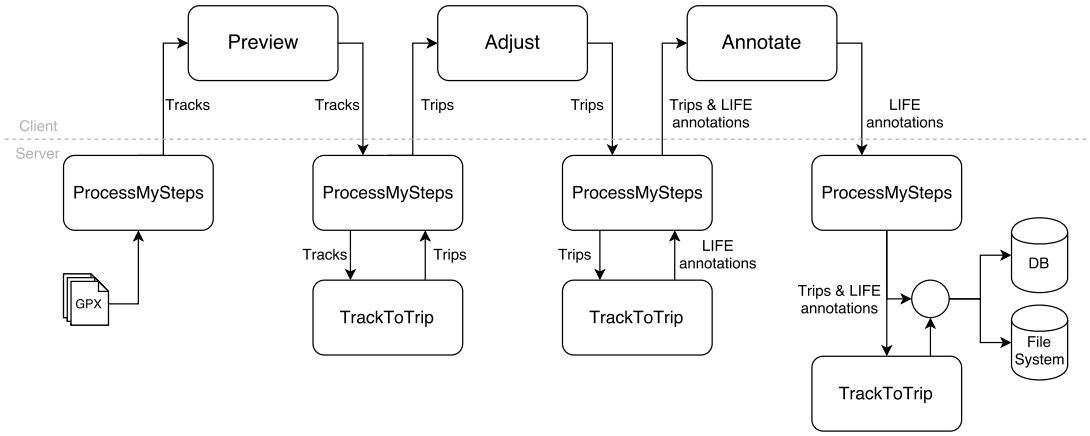


Figure 6.4: High-level data and processing flow

Figure 6.4 shows how data and processing flow throughout our solution. The client asks for the current tracks being processed. In the preview stage, users have the option of making changes to those tracks; regardless, the client sends the current tracks back to the server, which will use *TrackToTrip* to transform tracks into trips. The server responds back to the client, with the trips. Again, but this time in the adjust stage, users can edit, remove or add trips (all in the client). The result will be sent to the server.

At this point, the server builds a *LIFE* annotation — based on its previously acquired knowledge — and sends it back to the client. At the annotation stage, users annotate their tracks. The client asks for location and transportation mode suggestions when the user is annotating. When the user is done annotating, the client sends back the *LIFE* annotations, and the server creates the resulting files (moves original GPX files to a backup folder, saves trips as individual GPX files, saves *LIFE* annotations, etc.,) save the information to the database and learns locations as well as transportation modes.

According to the stages described above (preview, adjust and annotate), their descriptions and intended functionalities, we divided our system into two main screens: the track editor and the semantic annotation screen.

The track editor screen, which allows one to destructively edit tracks and trips, is used by the preview and adjust stages. While tracks are semantically different from trips, structure-wise they are indistinguishable. This allows us to use the same screen, with the same functionalities for both stages. This is good for the user, since he only needs to learn the functionalities of one screen to use both.

The semantic editor screen provides a text editor that allows for editing *LIFE* formatted annotations, it is used in the annotate stage. To help with annotations, users have access to syntax highlighting and suggestions to speed up the annotation process.

Both screens, however, are not completely different. We have mentioned that the map is one of the best ways to convey geographic information. With that in mind, we soon realized that both screens would share the same code to display information on the map. Both screens also have a side pane where all the information is presented.

The rest of this Section covers the development of the map module (in Section 6.1.1), track editor (in Section 6.1.2) and the semantic editor (in Section 6.1.3). Section 6.1.4 is dedicated to explaining the need for certain features that benefit geographic lifelogging.

6.1.1 Map

With the screens planned and with a rough idea of what we wanted to achieve, we started by looking at the technologies that we could use to help us realize our goals, with minimum development effort. At the time, web development was being taken by storm by the reactive paradigm, more specifically through *React* [2].

React is a JS library that eases the development of UIs, by reacting to the changes on the current data model. Our previous experience with *React*, allied to the fact that it was one of the most popular ways to develop web applications, made us decide to use *React* to implement the UI.

React, however, is only used to specify how to render the UI. To control state modifications and propagation, we needed another paradigm. A popular architecture that solves this problem is Flux [3], which was created by the same creators of *React*. Other popular libraries have sprung around Flux, one of the most popular being *Redux* [4].

Redux allows us to have stores that contain representations of our current state. With *Redux*, we can bind data stores to *React* components, which will react to stage changes signaled when there are changes in those data stores.

As we have seen previously, we decided that a map with tracks and trips was the focal point of *GatherMySteps*. As so, it was the natural starting point for implementation. We started looking for ways to present maps and tracks using HTML and JS. We quickly realized that the most flexible and powerful library to achieve that was *Leaflet* [5].

Leaflet is an open-source library that uses OSM [6] maps to display digital maps, where information can be overlaid. It supports the drawing of points, polygonal lines (polylines), polygons and markers. We also found out about *react-leaflet* [7], which is a library that provides *React* bindings to the *Leaflet* library. Since we decided to use *React*, it made sense to use *react-leaflet*.

[2] `https://facebook.github.io/react/`, last accessed on October 15th, 2016
[3] `https://facebook.github.io/flux/`, last accessed on October 15th, 2016
[4] `http://redux.js.org/`, last accessed on October 15th, 2016
[5] `http://leafletjs.com`, last accessed on October 15th, 2016
[6] `https://www.openstreetmap.org`, last accessed on October 15th, 2016
[7] `https://github.com/PaulLeCam/react-leaflet/`, last accessed on October 15th, 2016

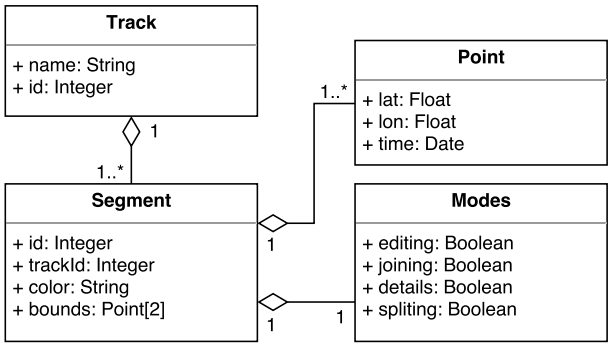


Figure 6.5: High-level UML diagram representing the data model of *GatherMySteps*

Our initial iterations consisted in loading a track and displaying it on screen as polygon lines. The state of our application, represented in Figure 6.5, consisted in immutable structures, implemented using *immutablejs* [8]. Immutable structures are one of the easiest and fastest ways to check whether data has changed, and if a *React* component needs to be re-rendered. In our architecture, like in Chapter 4, a track and a trip have the same structure, but different semantic meaning.

6.1.1.1 Scalability

The technology stack that we detailed was good enough for most situations, however, when a large amount of points were displayed on the map, the system would lag in simple actions such as panning or zooming in and out. Moreover, with the introduction of features that allow the destructive edit of tracks and trips, we started running into situations where the system would become unusable.

Reactive UIs are a bliss to implement. In most cases, we do not need to think about how the information will be updated when our state changes. In our case, however, we needed to be wary. After all, we need to be able to display at least thousands of points; furthermore, actions such as panning and zooming should be highly responsive, as they are the most intuitive way for users to explore their tracks and trips.

We opted to follow an iterative approach, reaching a feature complete solution first. When we reached that stage, we looked back at *GatherMySteps* and were not happy with the user experience, so we decided to improve it. The navigation, as we have mentioned, was sluggish, but most of all the features to edit tracks were unbearably slow. After looking through the code and profiling its execution, we concluded that the *react-leaflet* library was slowing down our application.

The *react-leaflet* library works just like any other react component: whenever there is a change in its properties or state, it renders itself again, with the new properties or state. This is a simple process that works wonders with small datasets. In our case, this means that every time a point is moved, the whole track or trip must be redrawn in the map. At our scale, with tens of thousands of points, it is not practical. To solve this issues, we implemented a custom react component to render tracks and trips in a leaflet map.

With a custom implementation, we can look to leverage the knowledge we have over our data model, fine tuning some of the features. It allowed us to make changes to the points of the map as needed, preventing unnecessary allocations and deallocations of objects (points and polylines). If a single point was changed, we can specifically adjust that point without re-rendering the whole track or trip.

Navigation and features like split, join and inspect greatly benefited from this new approach, but it was the point editing feature that saw the greatest performance increase. First, by allowing the user to move points, we can reuse the markers already allocated and used by the split and point inspecting

[8] `https://facebook.github.io/immutable-js/`, last accessed on October 15th, 2016

features. This meant that we only needed to allocate markers to create new points. Second, once a point is created, we can update the polyline and markers with minimal allocations, albeit using a complex process. Actions in tracks with thousands of points, such as panning and zooming, were now fluid.

6.1.2 Track Editor

The track editor, as we have discussed, is the screen where users can modify their tracks and trips, using actions like split, join or edit.

Before moving on to implement those actions, we also needed to think on how to present different tracks on the map. The quick answer was to represent them in different colors. The selection of colors, however, is not a simple process.

First of all, tracks are categorical data. Second of all, we needed to ensure that the colors selected were easily distinguishable between each other and from the background. Third of all, there might be up to ten or twenty tracks being processed at a time.

Figure 6.6: First color palette to represent tracks

To address some of these issues, we used the online tool *ColorBrewer2.0* [9] to generate a color palette, to identify each track. *ColorBrewer2.0* provides high quality color palettes for different data categories. In fact, it has become one of the main ways to generate consistent and user friendly colors for maps and cartography by the *InfoVis* community. Figure 6.6 shows the first color palette picked. Later on in development we realized that those colors were not appropriate, as they were soft and often confused with other elements of the OSM map. We decided to reduce the number of different colors, so that we could have distinctive and strong colors, that grab the attention of the users. The result of the color palette to represent tracks and trips is presented in Figure 6.7.

Figure 6.7: Final color palette to represent tracks

As we have seen before, it is not very efficient to just show the tracks or trips on the map. Users can quickly get lost, or not even realize that there are more tracks than those being shown. With that in mind, we set up a side pane where tracks were represented. That representation should include the color, name, date and time, and to further help the user remember what happened at that track or trip, some metrics, such as duration and distance.

We also realized that it would be a good place to put buttons to enable editing actions. During development, we kept iterating, improving the design and refining their functionality. Figure 6.8 shows the evolution of the representations.

Moreover, even with tracks and trips displayed and easily recognizable on the map, without further help, navigation can become a burdensome process. To solve those issues, we added three new features: hide or show tracks and trips, zoom to fit a track or trip into view and center the start and end points of a track or trip in the map. The problem was that we did not want to clutter the interface with four new buttons. Our solution was to add one button, which fits a track or trip. The action to center the map in the start or end of a trip could be accessed by clicking their time and date. Users could also hide or show the desired track or trip by clicking in respective representation.

[9] `http://colorbrewer2.org/`, last accessed on October 15th, 2016

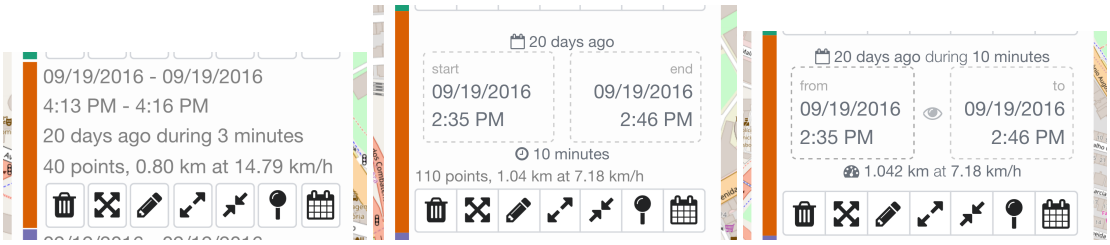


Figure 6.8: Evolution of the track representation design (first iteration on the left, final on the right)

Furthermore, since in our lifelogging scenario there is often more than one track or trip being represented, there are actions that are useful when used globally. One of those is the ability to hide all tracks or trips at once, so that users can toggle them, isolating the amount of information going their way. This can help identify and explore the trajectories of a day.

The next natural step was to start implementing the actions to manipulate tracks and trips: split, join and edit individual points. To execute these actions, users need to activate them, by clicking in their respective button. When they are activated, an overlay of point is displayed above the respective tracks and trips. Each track has its own modes, that are independent from each other. This way we can provide information to the user, in a way that is understandable, and such that they have precise control.

6.1.2.1 Split

The split action was the first to be implemented. From the start, we followed a simple approach to the problem: overlay a marker above every point of the track or trip. This approach worked well from the beginning. The fact that points represent where the user can split a track or trip, makes it simple to use and gives users fine control the process. Figure 6.9a shows an early prototype of the split action, while Figure 6.9b shows the final iteration. During the process we tried to make the points more appealing and make them feel like they were part of the track or trip.

(a) Prototype (b) Final version

Figure 6.9: Splitting a track/trip

Splitting a track or trip into two is a simple process. During development, however, we had to make a decision of whether the point that the user clicked to split the track or trip, would belong to only one track (Figure 6.10a), or both tracks (Figure 6.10b). We opted for the later option, since it gives the impression that we are cutting the track or trip into two. With the other option, users might be unsure of what would happen, which point would belong where. This way, the result is always the same, the spliced tracks meet in the point where they clicked.

(a) Track/trip from the left contains split point

(b) Both tracks/trips contain split point

Figure 6.10: Different results from splitting

When splitting, we create a new track or trip that contains the points from the one clicked to the last; a new color will be attributed to it. The original track will remain with the points between the start and the one where the user decided to split it, keeping the same color representation.

6.1.2.2 Join

The join action is essentially the reverse of the split action. With the split action, we had to decide whether to include the splitting point in both splits; the join action required users to select tracks that they wanted to join. We knew that we wanted to have points that users could click to join two actions, but this raised a problem: what if the user decided to join two tracks, when there is a third one happening (temporally) between them? This could produce an inconsistent state, where the user had two trips that overlap temporally.

To prevent this, we decided that only tracks that were temporally sequential could be joined. This means less liberty to the user, but in a lifelogging scenario it makes more sense.

(a) Spatiotemporal contiguous tracks/trips

(b) Spatiotemporally separated tracks/trips

Figure 6.11: Joining two tracks/trips

The joining action is capable of joining two tracks or trips that share the same spatiotemporal point (Figure 6.11a) and tracks or trips that are farther apart (Figure 6.11b). By default, the user has one line or point between the two tracks or trips that can be joined; in this case, the tracks are joined, and no points are interpolated.

If the tracks or trips to join are farther apart, we request the server the trip already learned between the two points to be joined. If there are valid suggestions, the straight line will be replaced by them. Various suggestions can be displayed at once; the most common suggestions are displayed with thicker lines. Figure 6.12 shows the suggestions to join two trips.

Figure 6.12: Suggestions when joining two tracks or trips

When joining two tracks or trips, we merge the points of both tracks, maintaining their temporal sequence. If the ends have equal spatiotemporal points, we discard one of them to avoid redundant points.

6.1.2.3 Edit

Technically, the edit mode was one of the most challenging features of our system. We started by using *leaflet-editable-polyline* [10] plug-in for *Leaflet*, that allows the editing of polylines — the *Leaflet* structures that we were using to display tracks and trips on the map.

The plug-in, however, did not offer flexibility and a finer grain of control. We wanted to have a marker for each point (similar to those for the split action), a marker between each point (to add a point) and markers at the ends to extend the track or trip (same as adding a point). Since we already had a way to display half of the points (a marker per point), which we use in the split action, we just needed to implement ways to add points.

Figure 6.13a, shows what the edit action looked like. Points could be moved by dragging the markers and the same goes to create points. The time of each new point is the linear interpolation between the two adjacent points' time.

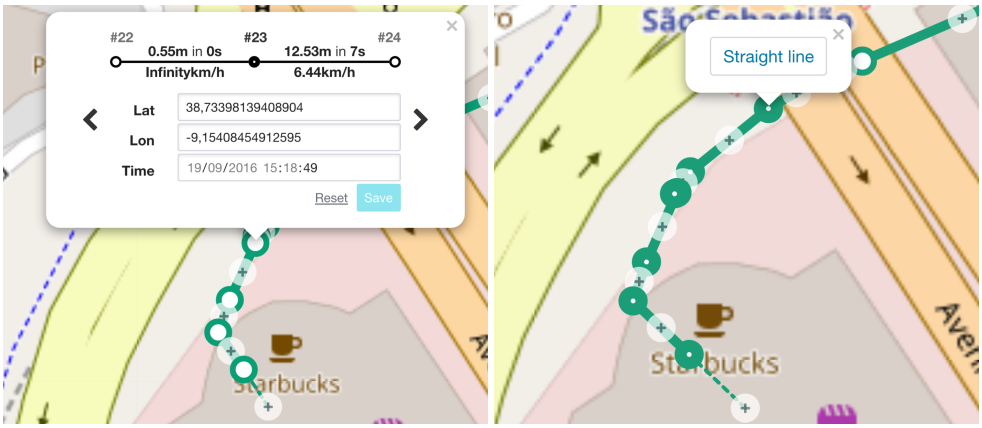
(a) Using plugin (b) Final iteration

Figure 6.13: Point edit feature

Our custom implementation in Figure 6.13b, consisted in reusing the markers layer to display the points, like those in the split action and markers that allow adding new points. The markers that represent points are draggable, which allow the users to move the points, while a right click on a point removes

[10] `https://github.com/tkrajina/leaflet-editable-polyline`, last accessed on October 15th, 2016

that point. Between each point we added a new marker, to add a point between those two points. To add a point, the marker can be clicked or dragged to the desired place. The ends of a track or trip also had a marker that allowed them to be extended. To help visualize the changes and provide feedback to the user, we use auxiliary lines that appear while moving markers.



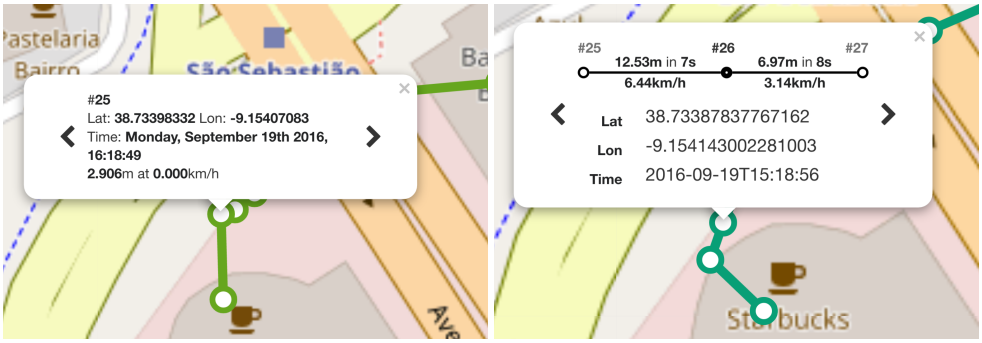
(a) Details of a single point (b) Multiple points

Figure 6.14: Editing points

In further iterations, we introduced a popup to edit each point coordinates and time (Figure 6.14a). To maintain time consistency of the track or trip, each point can only be attributed a time that is between its two consecutive points. Moreover, we implemented a way to edit multiple points at a time. By shift-clicking in one point and then in another (selecting all points in between) it is possible to straighten those points. We had other actions planned, such as to move those selected points temporally, but those actions were not implemented due to schedule restrictions.

6.1.2.4 Inspect

As lifeloggers, we found ourselves wanting to inspect a myriad of minor details about our trajectories. Those might be at what time a point was recorded, the velocity or time interval of one point against its neighbors or the GPS coordinates. For that we implemented an action to inspect points.



(a) Prototype (b) Final version

Figure 6.15: Inspecting a point

We decided to use the same markers over tracks and trips, as it was already a familiar way to interact with tracks and trips, avoiding the need to come up with new metaphors. When inspecting, instead of a destructive change, a popup appears with the information about that point. Figure 6.15a shows an early implementation of this feature.

To simplify the navigation between neighboring points, we added clickable arrows that move the popup to the next point (right arrow), or to the previous point (left arrow). Albeit simple, the arrows can induce error within the first contact, since they point to a certain direction; users may find themselves confused. When an arrow points left, users may expect the popup to advance to the neighboring point at its left. In reality, it may advance to the point to its right, since temporally that is the next point. We know this may be confusing at first, but it can be learnt quickly.

Figure 6.15b shows the final version of the popup, with more information, such as velocity, distance and time between the two consecutive points.

6.1.2.5 Undo & Redo

As with any editor, whether a text, photo or video editor, there is a feature that is transverses all of them. That feature is the ability to undo a previous action, or redo the action that was undone. Since we have the ability to edit tracks and trips destructively, it is natural to provide undo and redo mechanisms. This allows users to recover from mistakes, while given them the ease and liberty to experiment with destructive features.

As we have mentioned before, we are using immutable structures. This means that every time there is an action applied to tracks or trips, whether it is to join, split, remove or edit to the points, we are producing a new state and removing the previous one. *Redux* offers a simple way to implement undo/redo around these principles. Our first iterations started by using that approach.

The simplistic approach, however, turned out to be unfeasible. After each action — even the simplest one — we would store the entire representation of tracks and trips. This resulted in a linear memory growth at each action. Within a few changes, the browser would crash, lacking the necessary memory to run.

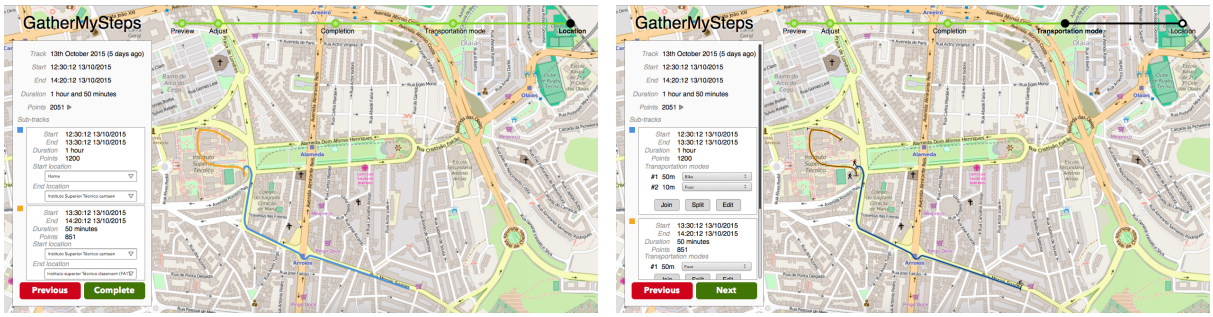
To solve this problem, we looked back at the actions, specifically join and split, and realized that those two were opposite. To revert one, all we needed to do was to execute the other with similar parameters. We applied a similar process, not only to join and split, but also to edit and remove actions. Instead of storing the whole state, we stored a way to reverse the executed action, using the minimum amount of memory. The result was impressive, with crashes becoming a thing of the past.

6.1.3 Semantic Editor

Our first mockups of the system, Figure 6.16a and 6.16b, annotations consisted of simple drop-down forms, that offered little flexibility and required two steps to annotate the transportation modes and locations. As we have seen in Chapter 3, there is a format in which users can annotate their locations, transportation modes and other personally relevant information they may wish to record, as different lifeloggers have different needs. That is the aim of the *LIFE* format (specification in Appendix D).

One of the main benefits of the *LIFE* format is its flexibility and simplicity to understand and edit the data represented. Being a text format is an important factor for it to be so easy to use and understand. Since we decided to use the *LIFE* format to annotate days, we needed create a text editor to have the same benefits as the *LIFE* format. A plain text editor, however, is too simple. As so, inspired by modern Integrated Development Environments (IDEs), we decided to create a *LIFE* editor that resembled an IDE.

After some planning, we realized that, at least, we were going to need to help the user identify the important *LIFE* element, such as locations, time spans, transportation modes, etc. We could also give a list of appropriate suggestions to the users, such as possible alternative locations, or transportation modes users might have used. Besides helping the user annotate faster and with less clicks, it also



(a) Location

(b) Transportation mode

Figure 6.16: Mockups screens for the annotation stage

provides a way to keep consistency within names. For instance, when annotating a grocery store, if the user has annotated the name of that store before he or she can use it and keep consistency, instead of using a generic abbreviation.

Moreover, it supports different user profiles. The feature described above allows the editor to work like a drop-down form, which is expected from experienced users, while expert users are able to annotate a single day without leaving the keyboard, also having the benefits of suggestion. With this settled, it was time to build the semantic editor.

We first looked at *React* and whether we could use it to manage the state of *contenteditable* components — the only way to edit and decorate text in HTML. It turns out that react is incapable of such. Next up, we looked at how we could use vanilla JS to implement such interactive editor. Our research showed that the amount of work to achieve an interactive editor, reliable enough to provide a good experience, was going to take a lot of work.

It was around that time that one library, Draftjs [11] (developed by Facebook), was released to the public. It allowed using *React* and vanilla JS to decorate a *contenteditable* HTML element. Upon reading about it and looking at their documentation, we set up to build the first iteration of the semantic editor.

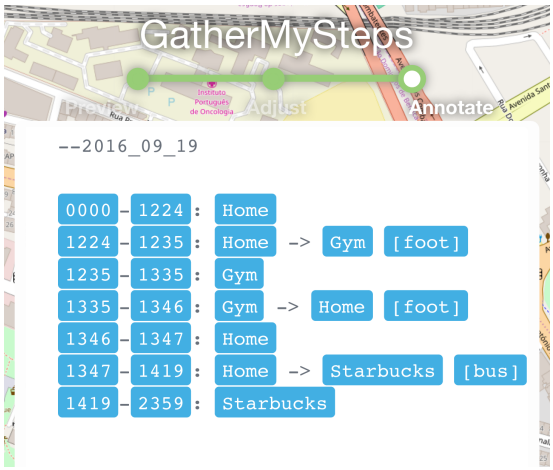


Figure 6.17: Semantic editor early prototype

The first iteration, as shown in Figure 6.17, made use of regular expressions to extract the time spans, locations and other information from the *LIFE* formatted text. It was not very fast and reliable, but still, good enough and gave us hope for the future. Around that time, we also started to implement

[11] `https://facebook.github.io/draft-js/`, last accessed on October 15th, 2016

ways to provide suggestions to users, initially, by retrieving from the server all the suggestions for each location and transportation mode. Those suggestions were retrieved all at once, when the client asked for the current state.

Besides the lack of reliability, it would not recognize a different layout of trips other than the original and there was no feedback in the map, which made the edition a tedious and inflexible guessing game.

To tackle those problems, we decided to take a fresh look at the problem. We changed the way we identified and decorated text. Instead of using regular expressions, which were long and with limited expressiveness, we created a Parsing Expression Grammar (PEG) parser (using *PEGjs* [12]) that would build an Abstract Syntax Tree (AST) after each keystroke.

A PEG parser is program that is defined by a set of well defined rules (grammar) and is able to recognize strings that represent those rules. Besides recognizing strings, we can build a tree that represents a hierarchy of the rules recognized. This hierarchy is the AST. This required the implementation of a grammar that defines the rules for the *LIFE* specification. The result is an AST, where each branch is a *LIFE* element, such as: trip, stay, location, time span, tag, etc.

This alone made the semantic editor much more reliable, however, slower. In most cases, it was unnoticeable, but when noticed, users could be typing and some text that they insert would not register. We opted to provide reliability in favor of speed. With that in mind, we introduced a timer, so that only after the user stopped typing, we would parse the text and build the AST.

With an AST of the *LIFE* annotations, we could also associate trips and points to their respective point in the map. For that, we extract the time of a location or transportation mode and looked for a point with that time. The *LIFE* AST element is marked with the id of the trip and the geographic position of that point.

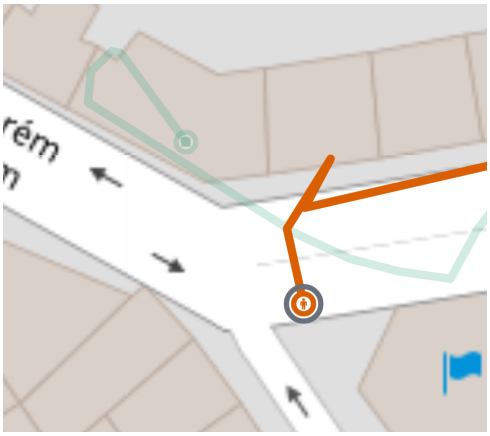


Figure 6.18: Segment and point highlight, caused by semantic editor

With that information, we were able to highlight the segments and points on the map. For that, users only need to hover with the mouse above one element and its position, and segments are highlighted on the map. As shown in Figure 6.18, points have a circle around a certain location, and segments that are not the ones being highlighted are faded.

Furthermore, if we know the geographic location, we are able to make better predictions about the locations, or transportation modes. Figure 6.19, shows the predictions for the transportation modes of a trip. Predictions are ordered by their relevance and decided by the server.

The first few iterations featured boxes, to give the user the impression that the editor could be used as a form. After a few iterations, we felt that the design was poor. There were too many boxes, which were not distinct, and they could deter the user from using the text capabilities. Again, we looked back at

[12] `http://pegjs.org/`, last accessed on October 15th, 2016

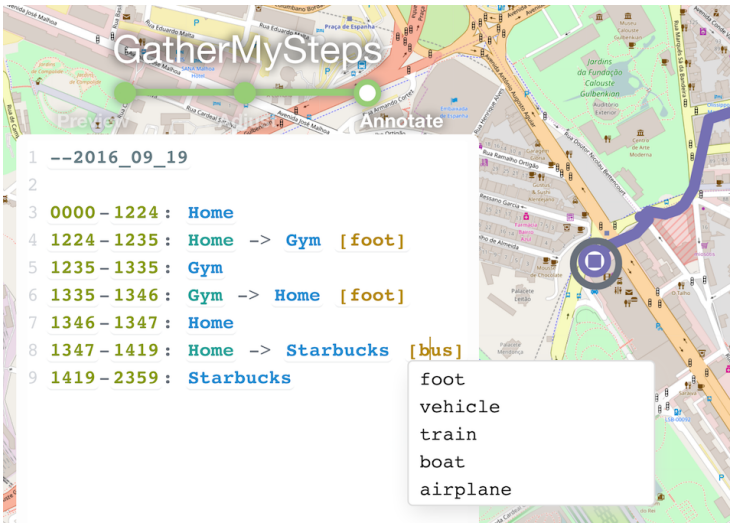


Figure 6.19: Final iteration of the semantic edior, suggesting transportation modes

our IDE inspirations and changed the decorations to a subtler style, that resembles syntax highlighting more accurately.

Figure 6.19, show the final version of the semantic editor, with small touches, such as lines under locations, transportation modes and time spans, that give the impression of a text box.

6.1.4 Other features

Besides all the essential features presented in the sections above, to provide a good experience that adapts to the different types of geographic lifeloggers we had to consider and implement other small, but nonetheless important features.

As we have mentioned, when processing personal GPS tracks, it makes sense to group all the tracks of a day together. This reduces the time lifeloggers spend processing their tracks and allows them to construct a better picture of their day.

In development, when we were testing the system with our own data, we realized that sometimes we wanted to change the day that we were processing. Either because we had little to no recollection about a certain day — exploring adjacent days might give some clues and help us recollect — or we just did not want to process that day at that time.

With that in mind we have created a drawer, as in Figure 6.20, that lists the different days available to process. We also saw the opportunity to add an option to remove a day from processing (without removing the files on the system) and a refresh button that requests the server to rescan the input folder for GPX files. In the same spirit, when on the preview stage, we decided to add a *Skip Day* button, which previously was a disabled *Previous* button, like in Figure 6.22.

Figure 6.22: Navigation button, with and without *Skip Day* button

Furthermore, if the system learns enough about users and the tracks to process are common trajectories, with learned locations, users may want to bulk process every remaining day. The drawer with the days left to process has a button that allows that.

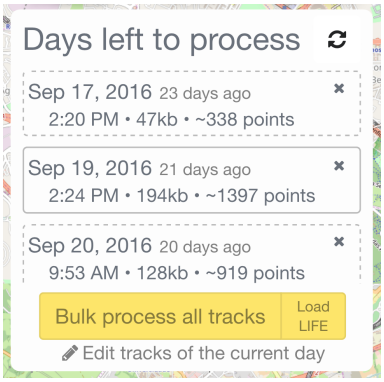


Figure 6.20: Days left to process

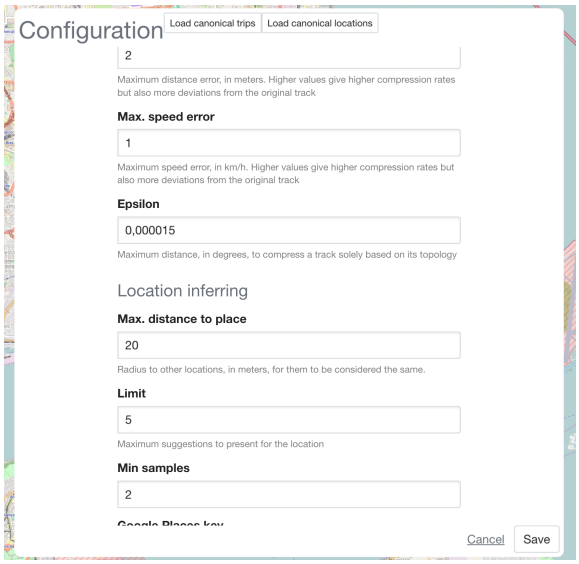


Figure 6.21: Configuration panel

To adapt to different types of users, besides having a thoughtful UI, we needed to provide the users with a way to easily change the parameters of the server and of the *TrackToTrip* library. This is crucial, since users use different equipment to record their trajectories that have different accuracies, which may require different parameters. Moreover, users may want to skip any step of transforming a track to a trip (as described in Chapter 4).

Since our domain is lifelogging, we have to accommodate those cases, as each user may have a slightly different approach or needs. To solve this issue, we implemented a configuration panel, as shown in Figure 6.21, that has all the parameters that can be changed, with a brief description of its impact in the system and what the values mean.

6.2 GatherMySteps User Interface

In this section, we are going to present the final version of *GatherMySteps*, exemplifying the steps and functionalities that can be used to process a day's worth of tracks.

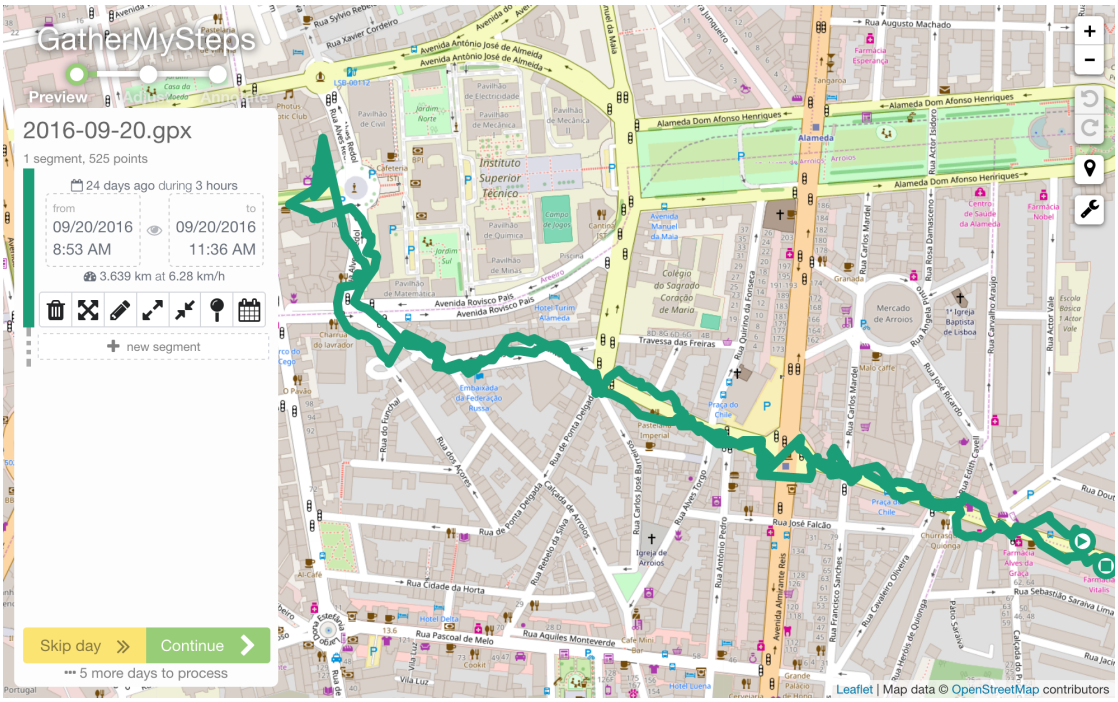


Figure 6.23: Preview stage

Figure 6.23 represents the typical screen that is presented to the user when *GatherMySteps* (and *ProcessMySteps*) is started and there are days to process. Right away we can see that we have at least one day to process, which is composed of one track. This is evident because there is only one track represented on the left pane. That same track is also represented in the map.

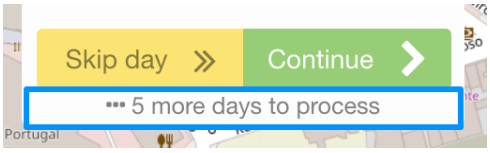


Figure 6.24: Indication of days left to process

Besides this day, we can see that there are five more days left to process, highlighted by the blue rectangle in Figure 6.24. We can click that region to view the days left to process. From there we change to any of those other days, by clicking in any of them.

Figure 6.25 is a close-up of the left pane with the days left to process. From there, we can rescan the input folder for GPS files, using the button highlighted by in orange and remove any of those days from the processing queue using the crosses buttons, highlighted in green. Furthermore, using the big yellow buttons at the bottom, we can automatically process every day queued. To go back to where we were previously, we have to click in the region highlighted in blue.

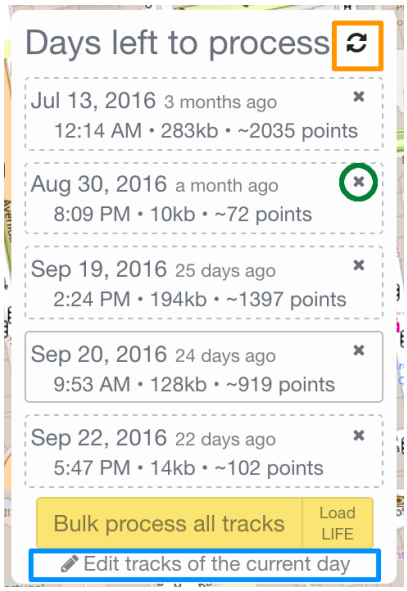


Figure 6.25: Close-up of the side pane, showing the days left to process

The **preview** step is useful to compare the before and after of the automatic processing phase. In our case, we are going to advance by clicking *Continue* button.

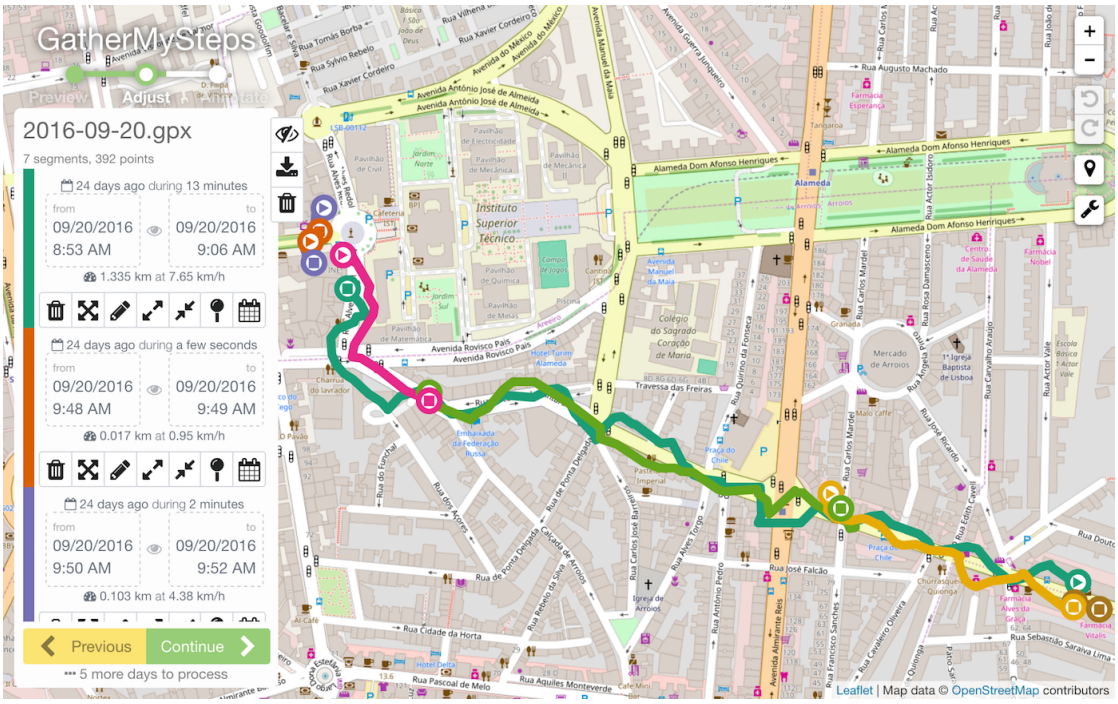


Figure 6.26: Adjust stage

We have now reached the **adjust** step, presented in Figure 6.26. Here we can see that the automatic processing phase split and organized our track into multiple trips. We can explore the trips. Each trip is represented by a color on the left pane, that corresponds to the color on the map.

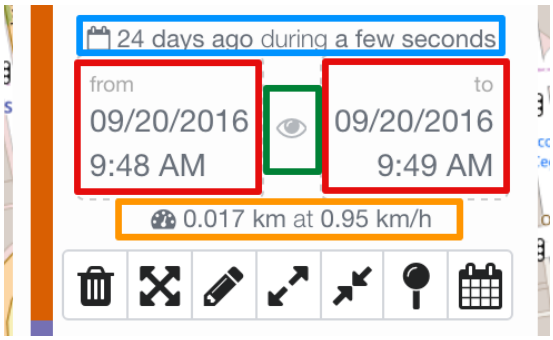


Figure 6.27: Trip representation

Figure 6.27 focuses on the side pane's representation of the red trip. There, we can see information about that trip: how long ago and the duration of that trip (highlighted by the blue rectangle), how long it was and the average velocity (highlighted in orange). Highlighted in red rectangle, we can also see the start and end date and time. Those are clickable buttons, that center the start or end point on the map. We can also click in the eye (highlighted in green) to hide or show the trip. When hidden, both the representation on the side pane and on the map fades. Moreover, we can edit and inspect a trip. To do so, we use the toolbar for each track. The buttons provided by toolbar provide a different set of functionalities:

- *the trash* deletes a trip;
- *the four arrows* icon focuses the trip in the map (fitting it into view)
- *the pencil* icon enables the edit mode
- the icon with *two arrows pointing in opposite directions* enables the split mode
- *the two arrows pointing to each other* enable the joining mode
- *the pin* icon enables the point inspector mode
- *the calendar* allows us to filter points by their temporal occurrence

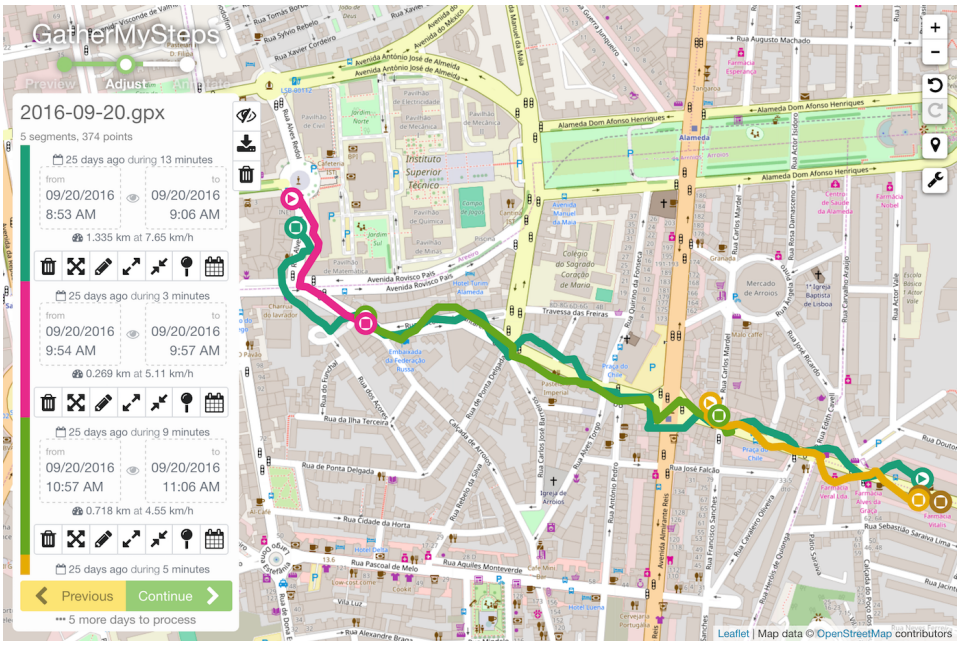


Figure 6.28: Trips of the day, after removing bad segmentations

We can use the functionalities that we have mentioned to help us explore the entire day, trip by trip. From Figure 6.26, it is evident that the automatic processing did a bad segmentation and created two trips (violet and orange) too small, that should be deleted. As we have seen, to delete them we just have to click in the trash button of each of those trips. The result of removing those two trips is presented in Figure 6.28.

In this day we started our day at home and ended at home. We can improve the first trip (green), making it start at the right place. To do it, we have to enable the point editing mode (pencil button). When enabled, points appear on the map and over the trip we want to edit. As shown in Figure 6.29.

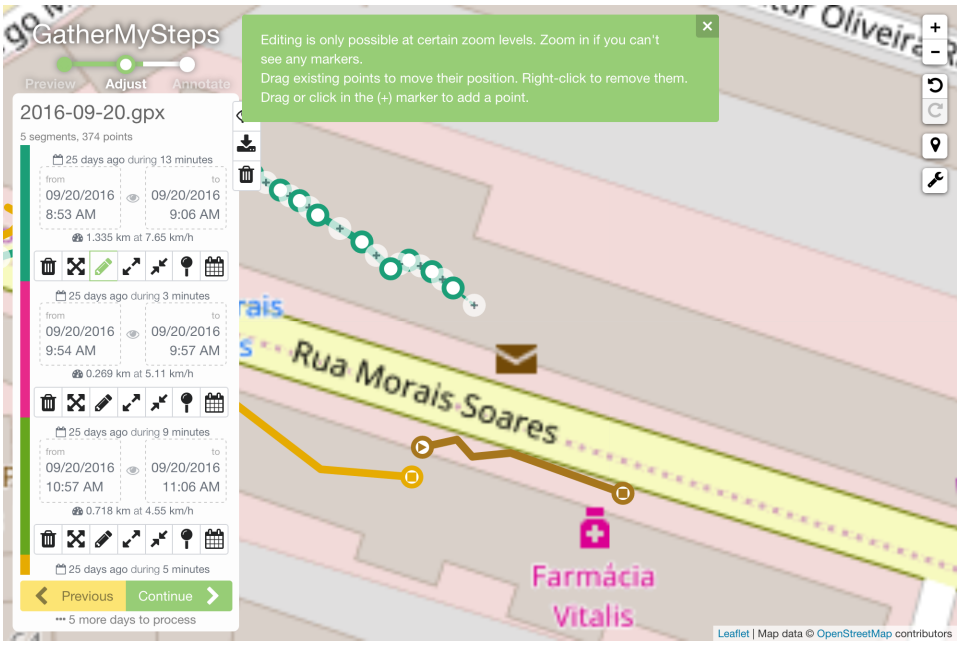


Figure 6.29: Edit mode enabled

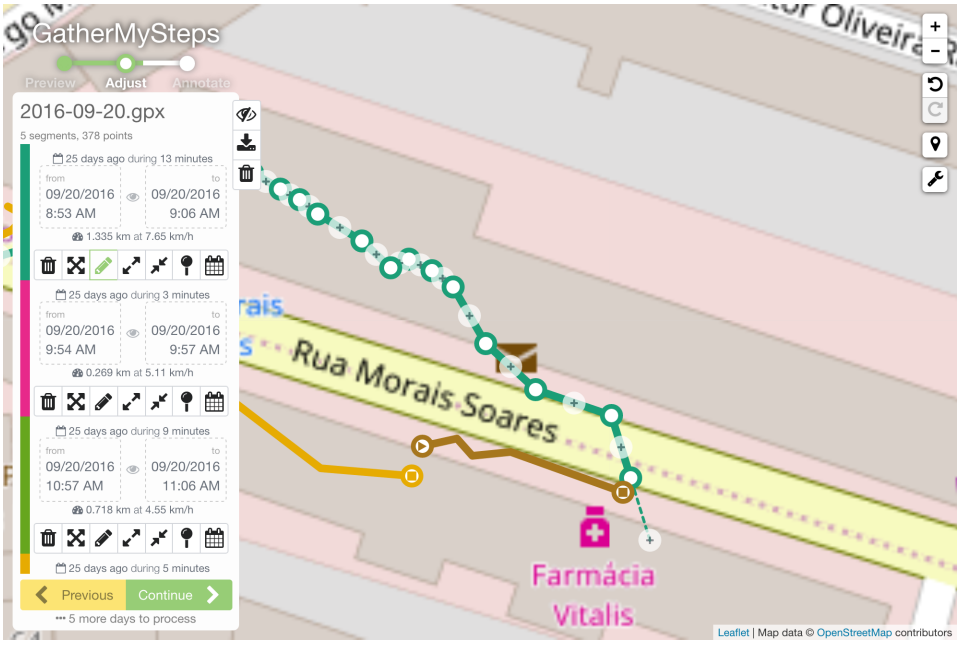


Figure 6.30: Point added and moved, using the edit mode

We can zoom and edit the points or add or remove new points. Above the trip that is being edited,

there are two types of markers: circles with a white center, which represent the existing points of a track, and markers with the "+" symbol, which allow the creation of a new points. To remove a point, we have to right-click in the desired point.

In our scenario, we add a few points to the beginning of the green trip, so that it starts closer to home. Note that we do not want to adjust the whole trip, that would be too time consuming. A small adjust to the start and end position of a trip may, however, improve the location inference engine. The result is presented in Figure 6.30.

The automatic processing phase also detected a pause at a location, before we returned home. Upon initial inspection, we do not remember that happening, so we remove that segmentation. To remove this segmentation, we can join the yellow and brown trips. To join two trips, we identify one of the trips that we want to join and click the button to join (two arrows pointing at each other).

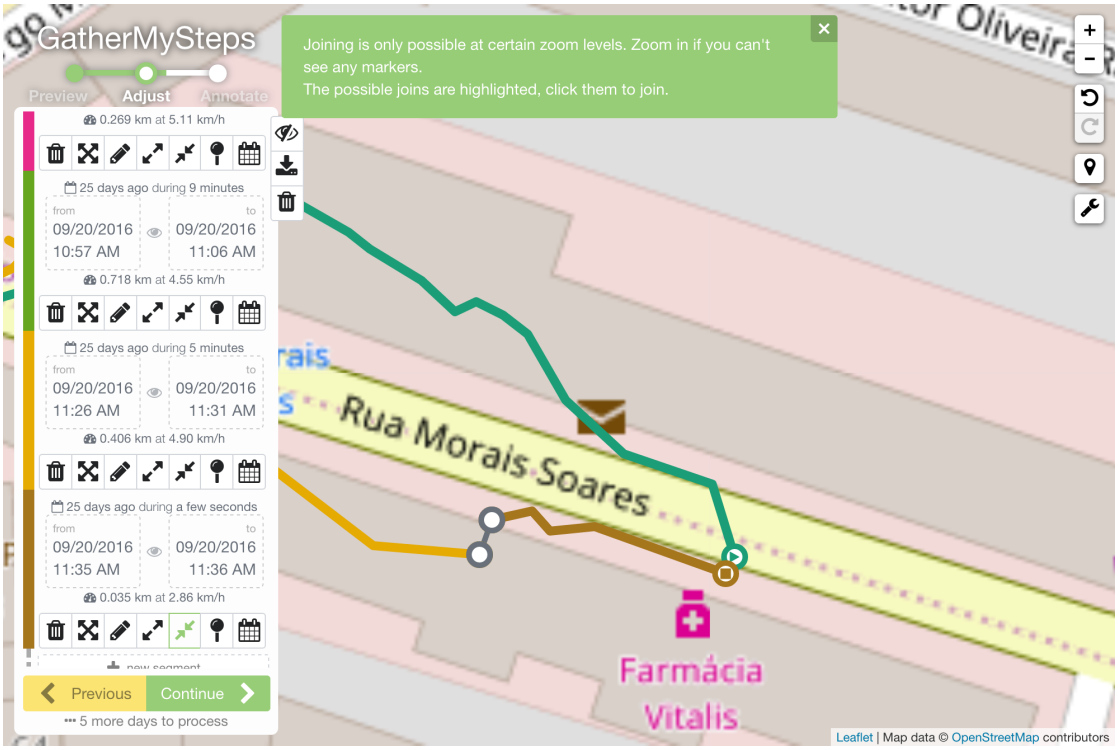


Figure 6.31: Join mode enabled

Figure 6.31 shows the join mode enabled for the brown trip. The yellow trip is the only possible trip to join with the brown trip. To join the end of the yellow trip and the start of the brown trip there is only one suggestion. If we click the suggestion (or the grey markers) the yellow trip is joined with the brown trip. The result is shown in Figure 6.32.

A few moments later we remembered that we had, in fact, gone to grocery store in that location. This means that the segmentation made by the automatic processing was right. Fortunately, we can undo and redo destructive actions, such as join a track. Using the buttons at top right of the map, we can undo (curved button to the left) or redo (curved button to the right) previous actions. To return to the previous state we just need to click the undo button. Users could also use common shortcut keys to undo and redo, such as *CMD-Z* (on a *macOS*) or *CTRL-Z* (in other operating systems).

For instance, if after leaving *home* we went to the *post office*, we could enable the splitting mode for the green trip (Figure 6.33) and click on the point near the *post office*. The result, in Figure 6.34, would be two trips: a green trip, between *home* and the *post office*, and a gray trip, between the *post office* and *INESC* — the destination of the previously green trip.

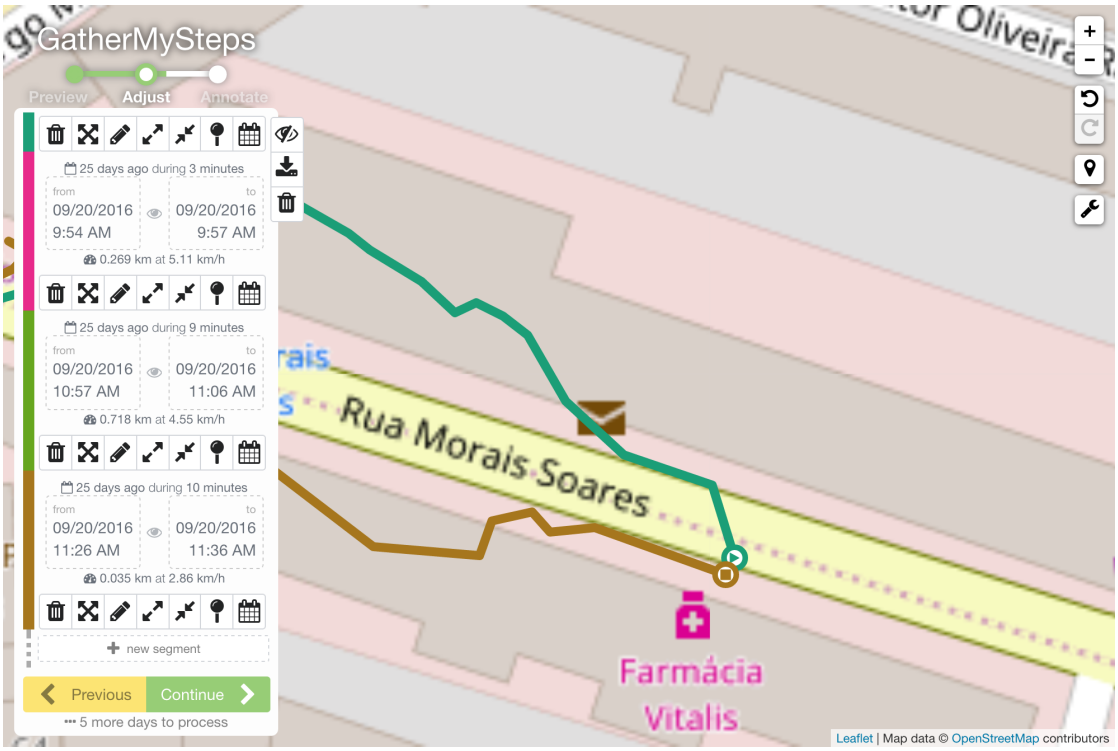


Figure 6.32: Result of joining two trips

We can also inspect a specific point to know, for instance, at what time we passed by the bus stop on our way to *INESC*. To do it, we have to enable the point inspecting mode for the green trip. Figure 6.35 shows the point inspecting interface. If we click in a point, in our case, close to the bus stop, a popup appears with a myriad of information about that specific point, as shown in Figure 6.36.

We have reviewed the trips of the current day and are happy with the current representation. As so, we can proceed to the **annotate** stage by clicking the *Continue* button.

In the annotate stage we can annotate our day, the interface is shown by Figure 6.37. We can see that the representations of trips on the left pane, gave place to the semantic editor.

If, for any reason, we wanted to go back and make changes in the adjust stage, we could click the *Previous* button. If we decided to go back, we would obtain the trips as like we left them, before clicking the *Continue* button. When we are done annotating the current day, we can save the annotations and tracks by clicking the *Save* button. After *ProcessMySteps* saves them, the next queued is loaded to be processed.

With the semantic editor we can annotate a day using the *LIFE* format. By default, we give some suggestions according to what we already know about the user. Right away we can see the trips and stays of that day. If we hover the mouse above them, their respective trips and geographic locations are highlighted in the map, as shown in Figure 6.38.

If the initial suggestions are not good enough, users can click the location or transportation mode and a drop-down appears with relevant suggestions. Figure 6.38, 6.39 and 6.40 show examples of that. In case the suggestions are not good enough, uses can edit the name of the locations and transportation modes like they would when editing text.

Moreover, experienced users can annotate their days without leaving the keyboard. If the *TAB* key is pressed, the next *LIFE* element will be selected. In fact, from Figure 6.38 to Figure 6.39 we only use the tab key to move and select the next location.

Users can also add more information about their days, that it is not represented by the trips. This is

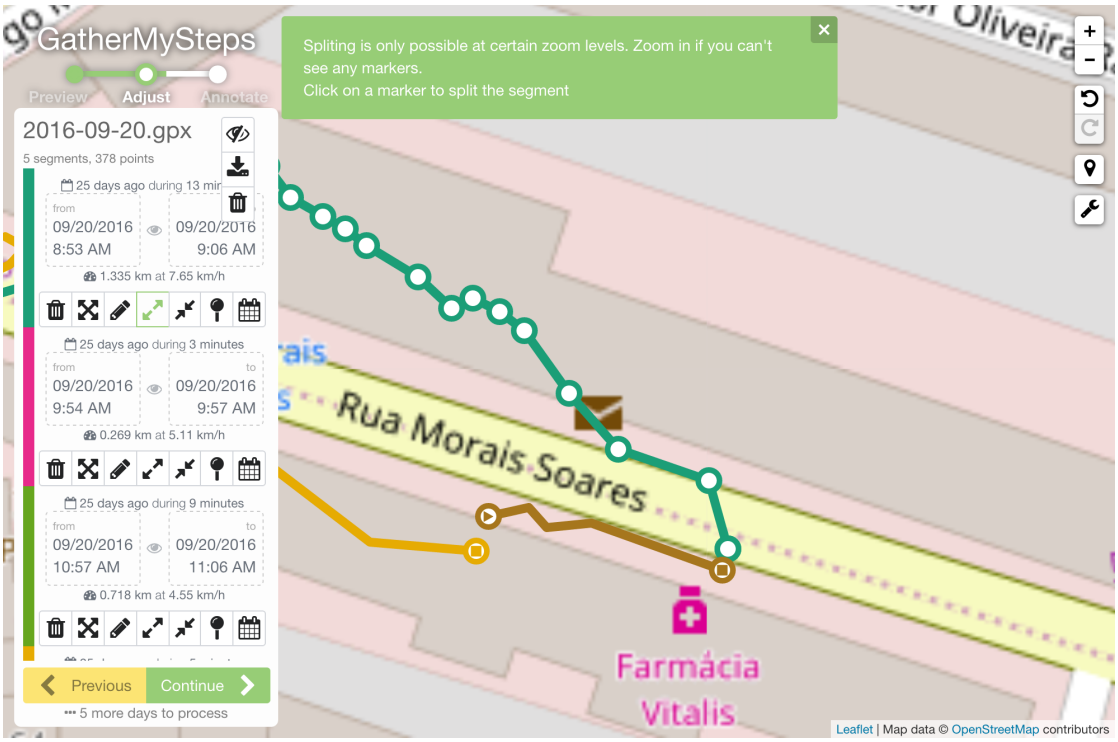


Figure 6.33: Split mode enabled

useful when users forget to record certain tracks, but know that they traveled certain trajectories.

After we have corrected and validated the annotations, we finish the processing by clicking the *Save* button and another day is loaded to be processed.

To further adapt to the needs and different profiles of users, we also provide a configuration panel, as shown in Figure 6.41. In there, at any time, users can tweak various settings and parameters related to the automatic processing phase and general definitions such as input and destiny paths.

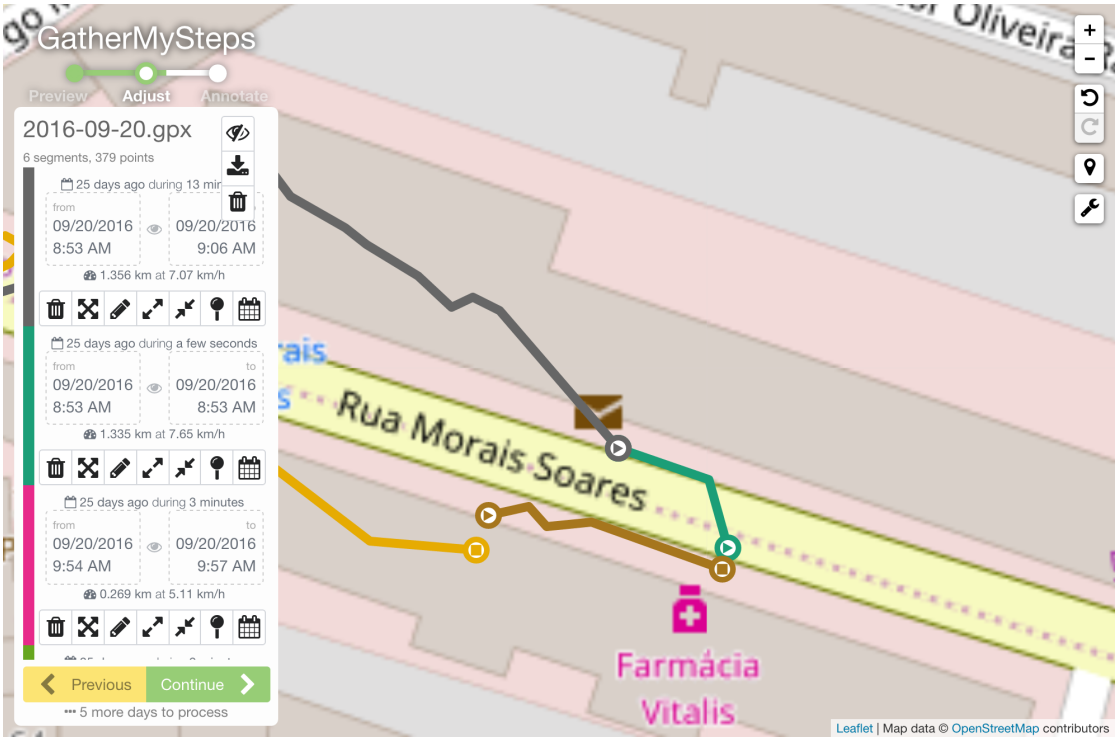


Figure 6.34: Result of split mode

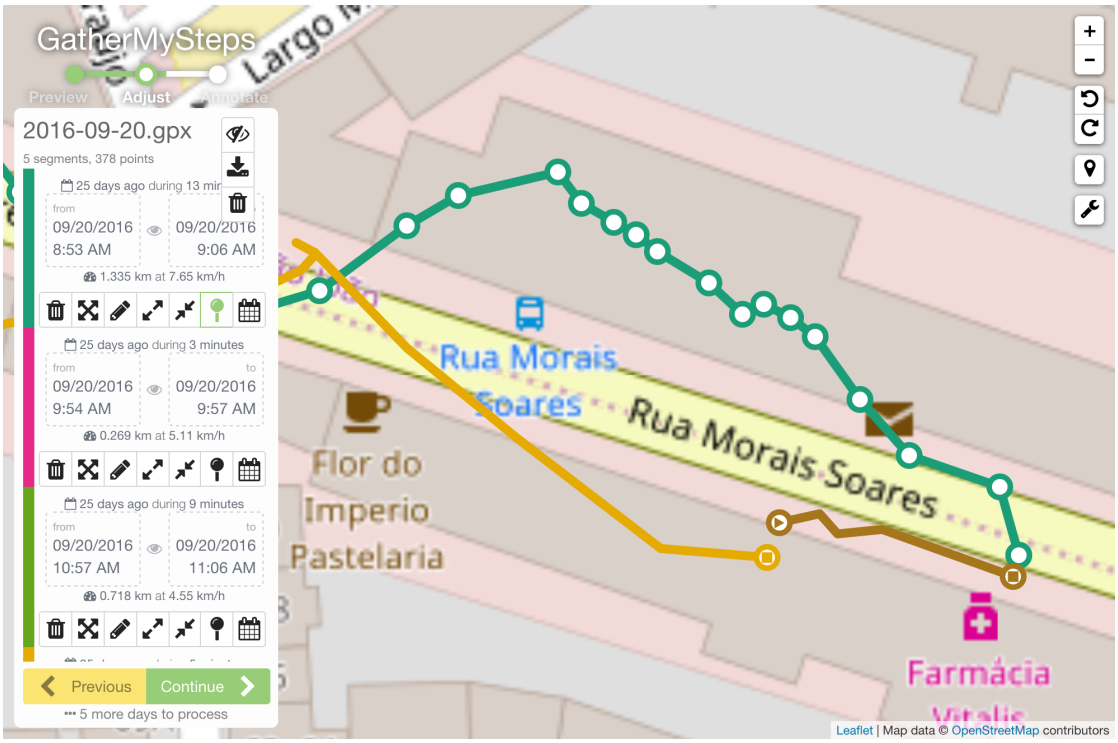


Figure 6.35: Inspect mode enabled

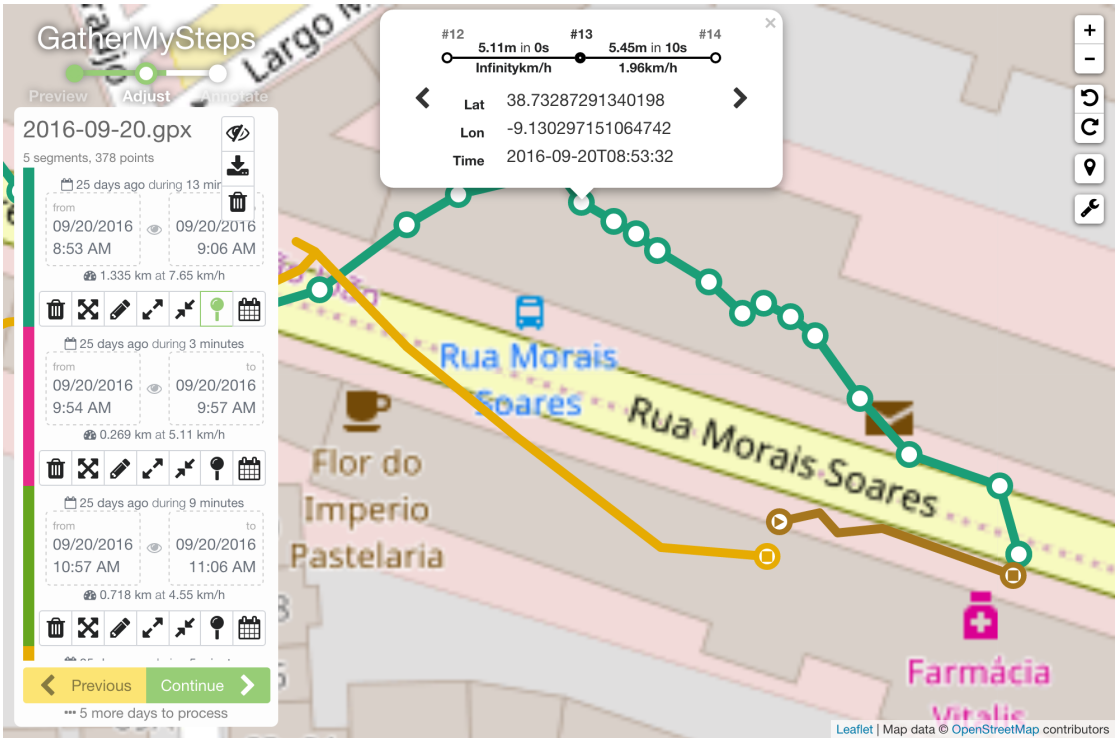


Figure 6.36: Inspecting a point

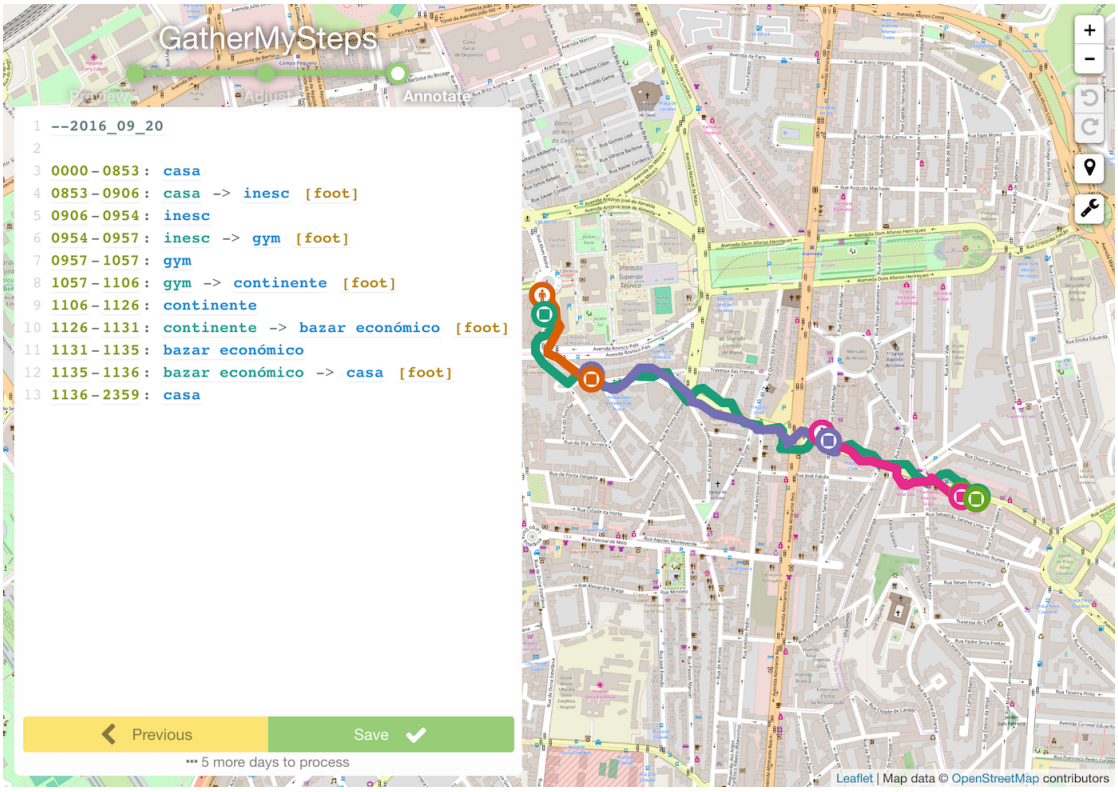


Figure 6.37: Annotate stage

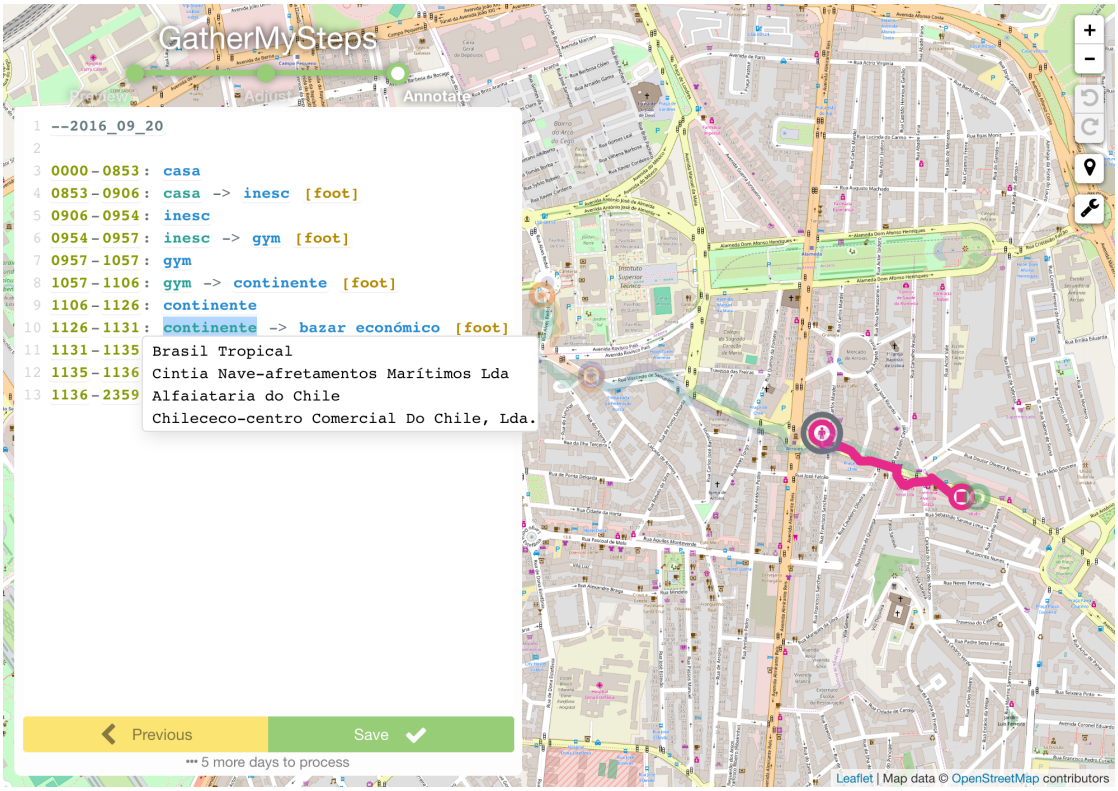


Figure 6.38: Annotating a location

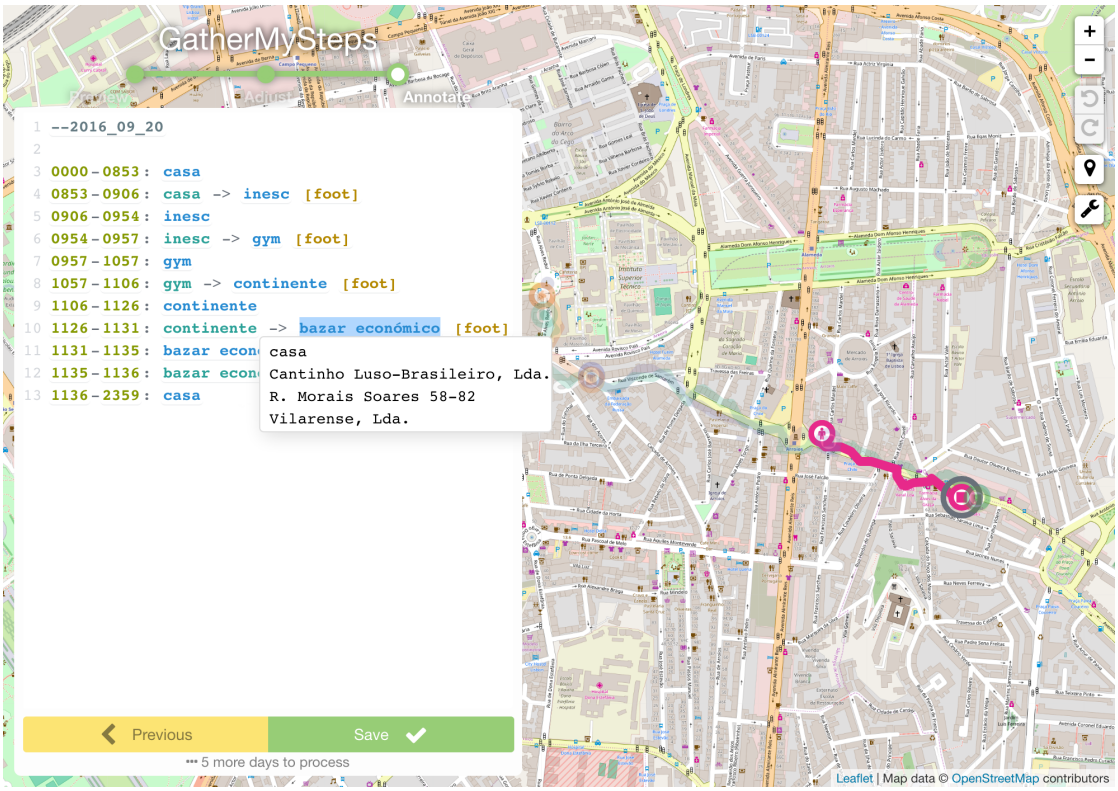


Figure 6.39: Annotating another location

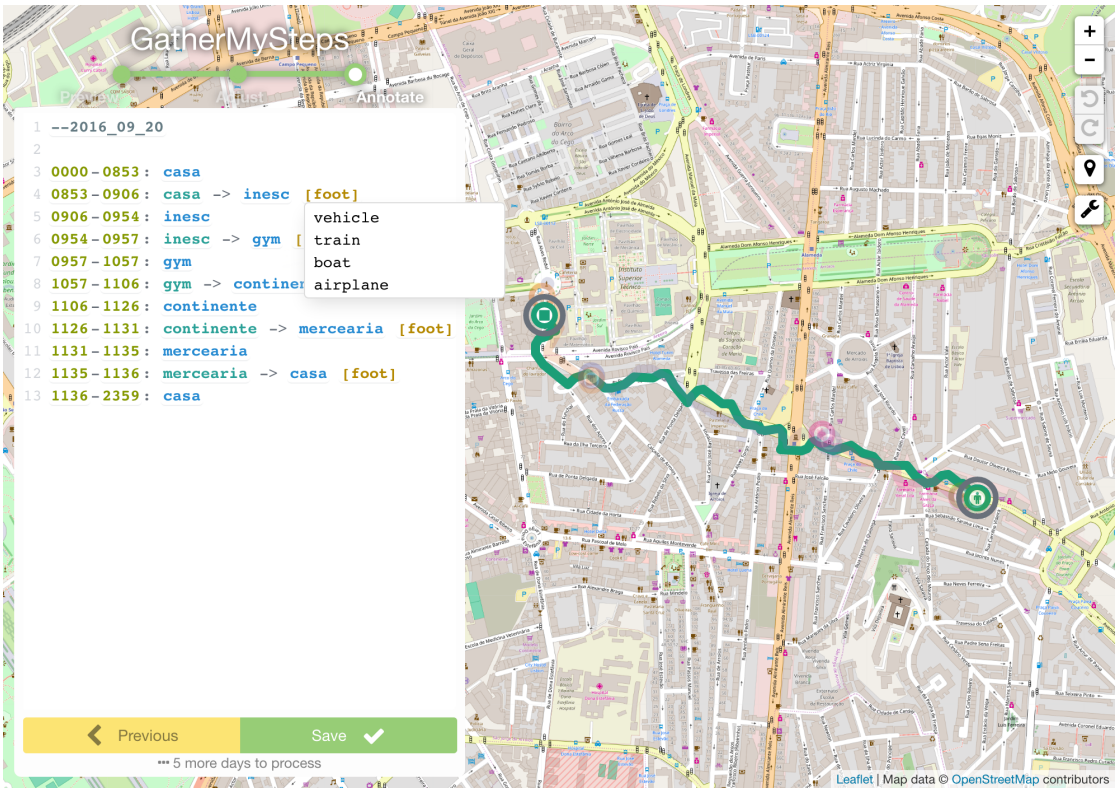


Figure 6.40: Transportation mode annotation

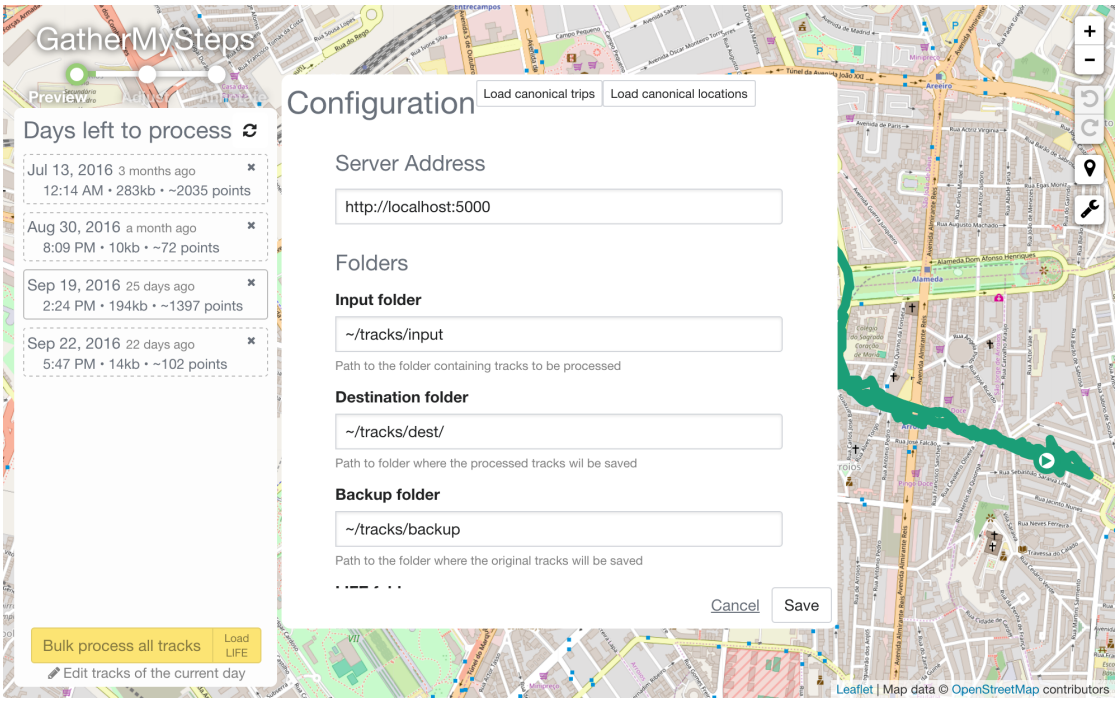


Figure 6.41: Configuration panel

Chapter 7

Evaluation

In this chapter, we are going to detail how we devised usability and utility tests for *GatherMySteps*. In usability tests, described in Section 7.1, participants were observed using our application with a fixed dataset, collected by us. Throughout the tests, we collected metrics and registered observations about the usage of our system, so that we can derive how user-friendly, understandable and how easy to learn our UI is.

In the utility tests, described in Section 7.2, we have asked a set of five participants to collect their trajectories and use our system with it. While they were using the system, we collected reactions, comments and suggestions. This allows us to check whether our application is useful and appropriate for the lifelogging domain.

We discuss the tests and results, then analyze if our solution is appropriate, as well as where it falls short in Section 7.3. In Section 7.3.1, we present an improved version of *GatherMySteps*, based on what we learned during the tests.

7.1 Usability User Tests

GatherMySteps is a UI created to process personal GPS tracks and the trajectories they represent. Since users may have several trajectories recorded and lifelogging is a everyday effort, our aim was to trim down the processing of a day, by either using automatic processing, or by suggesting places and transportation modes.

To have a clearer picture of how efficient the features are — such as track editing and annotating a day — we have created usability user tests.

7.1.1 Methodology & Setup

We selected 20 users, amongst friends and family, to test our application. Each of them was submitted to the same procedure, using the same dataset. Our goal was to see how user-friendly our solution was to users without any experience using our system.

Every user, individually and in isolation, was subjected to a preparation period. During that period, they read an handout (Appendix F) containing a brief description of *GatherMySteps* and the goals of the application. It also listed the tasks to perform, as well as a map marked with relevant places and trajectories (necessary since the users were not using their own data and, thus, would otherwise have no reason to assign particular meanings to any of the locations referred to in the dataset).

We did a four-minute demonstration of the application. The demonstration consisted of processing a day, from start to finish, where most features were used. Furthermore, we let them explore the applica-

tion for four minutes. In that time, they would replicate the demo, asking questions along the way. These phases were crucial to avoid periods of unnecessary frustration and exploration caused by initial contact with the application.

After a final period of clarification about the system, users started executing the tasks that were proposed. Each user had a unique, randomized task order.

The dataset was composed of six days' worth of tracks, recorded by us, with a total of 16 trips spread across eight different GPX files. The tracks were recorded by us, with an Alcatel Idol 3 Android device, with the highest sampling rate possible. The median resulting sampling rate of all tracks, was one point recorded every five seconds, with an average of 567.6 points per file.

Due to the complexity of our system, we devised two types of tasks: complex tasks, that simulate the processing of a day; and focused tasks, that require the user to perform a concrete action.

In total, there were three tasks that simulated the processing of a day, and where users were given information about the trajectories and transportation modes of that day. For each of those tasks, the maximum time to complete was 20 minutes. Tasks were described as follows:

- **Task 6**: "September 17, 2016: I left *home*, at 1:20pm, to the *gym*, where I was until 2:27pm. Then, I left to go to the *Pavilhão de Civil* and from there I left, at 5:19pm, to go *home*."

- **Task 7**: "September 19, 2016: I left *home*, at 1:24pm, to go to the *gym*. I finished training at 2:35pm and went back *home*. At 2:37pm I left *home* to to *Starbucks*, by bus."

- **Task 8**: "September 20, 2016: I left *home*, at 8:53pm, to go to *INESC*, where I was until I went to the *gym*, at 9:54pm. After I was done with my training, I went shopping to *Continente*, at 11:26pm I left to go shopping, again, at a *grocery store*. I ended my morning *home*."

The tasks described above were designed to make the user explore the trips and semantic locations — identifying start and end points —, removing segmentation leftovers and annotating trips. If the users so desired, they could edit points, specifically the start and end points, making trips more accurate.

The more focused tasks, five in total, required the users to execute a concrete feature of the system. They were prepared by the supervisor. For each track, the maximum time to complete was five minutes and the tasks were:

- **Task 1**: "August 30, 2016: Identify, by speaking, the coordinates of the point at 7:10pm."

  In this task the user has to use the inspect point feature.

- **Task 2**: "August 30, 2016: Add two points, to the beginning of the track, so that it begins at home."

  This task was designed to make the user activate the point editing feature, and to add two points with that feature.

- **Task 3**: "September 22, 2016: Split a track near the grocery store."

  Users had to use the split feature, to complete each track.

- **Task 4**: "September 22, 2016: Join the two tracks by the most common path."

  In this task users had to activate the join feature, and then select one of the various options.

- **Task 5**: "Change the day to process to July 13, 2016."

  This task was designed to make users change the current day to process.

Before each task, the system was initialized at the same state, that included one trip and the already learned locations, for the *gym*, *home* and the *grocery store*.

During the tests we collected metrics about the usage of the system, they were: time to complete task, clicks to complete task and number of errors during the task. For each task, the time started counting after the user had read the task. Each task had a time limit, that if reached, the task was deemed to have failed.

We registered every click to execute an action, ignoring clicks used to navigate the map, such as pan and zoom. When on the semantic screen, only clicks that lead to changes were counted, for instance when the user selected a suggestion for a point.

Every time the user used an unexpected feature that would not contribute to the completion of the task, we registered an error. There were cases, specifically when editing points, where users changed a wrong point, that were not considered errors if they could recover using the undo functionality.

Along the tests we also collected reactions that users would have while using, for instance, if the user was expressing boredom, frustration or any other comments that they would do.

When users finished performing the tests, they did an informal review, where they were prompted to give feedback and suggestions to improve the system. The test was concluded after filling a questionnaire, in Appendix E, to understand if they had ever used similar systems, if they were familiar with lifelogging and geographic lifelogging and what they thought of our solution.

7.1.2 Results

Users that tested our system averaged 22.5 years old. According to their answers, 10% of them never use the GPS, 50% use it once a week, 20% of them use it one to three days a week, 5% use it four to six days a week, and the remaining 15% use the GPS everyday. When asked about what they used the GPS for, accessing maps and localization services were the main usage scenarios. We also wanted to know how many users had already done geographic lifelogging. From the 20 users, only two had any kind of experience with geographic lifelogging.

As we have discussed before, we collected the time to complete a task, clicks to complete a task, the number of errors during a task and comments during the tests.

Below we present the results for each metric, as well as the computed statistics.

7.1.2.1 Time to Complete Task

Figure 7.1 shows a Tukey box plot with the resulting time to complete each task and Table 7.1 shows computed statistics: median, average, standard deviation and confidence interval.

From Figure 7.1 we can see that, as expected, the complex tasks took more time. Moreover, tasks that required the user to explore (Task 1 and 2) and that required more precision (Task 2) take more time and their standard deviation (as per Table 7.1) are higher than Tasks 3, 4 and 5.

We also need to take into account that most users were not familiar with the personally relevant locations presented during the test, as so, users often had to consult the handout to make sense of those places and the respective trips.

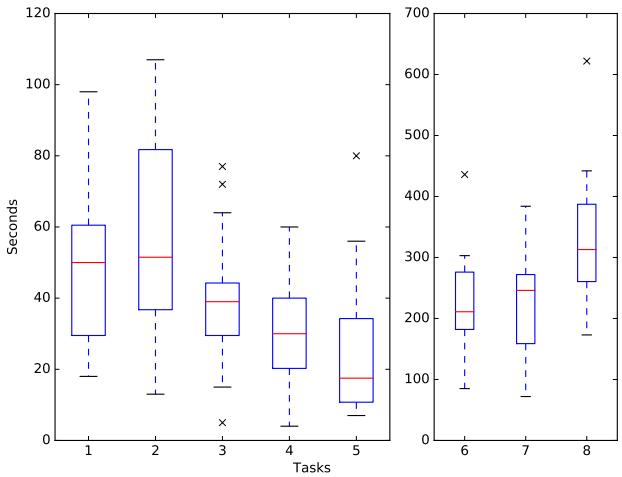


Figure 7.1: Time to complete each task

	Task							
	1	2	3	4	5	6	7	8
Theorical Maximum	420.00	300.00	240.00	120.00	120.00	900.00	1200.00	900.00
Median	50.00	51.50	39.00	30.00	17.50	211.00	246.00	313.00
Average	47.45	58.20	38.60	31.05	25.85	223.15	226.55	322.75
Standard Deviation	20.90	27.08	17.14	14.64	18.94	79.45	81.75	101.91
Confidence Interval (95%)	37.42 57.48	45.20 71.20	30.37 46.83	24.02 38.08	16.75 34.95	185.00 261.30	187.29 265.81	273.82 371.68

Table 7.1: Time to complete statistics for each task

7.1.2.2 Clicks to Complete Task

Figure 7.2 shows a Tukey box plot with the resulting clicks to complete each task and Table 7.2 shows the computed statistics: median, average, standard deviation and confidence interval.

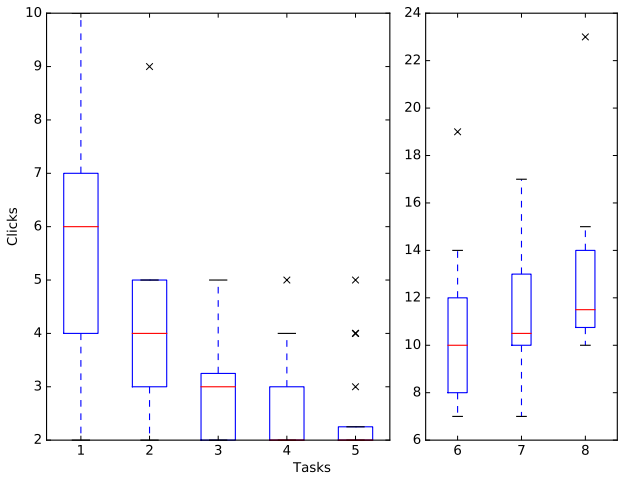


Figure 7.2: Clicks to complete each task

Figure 7.2 describes a similar scenario to that of the previous Section 7.1.2.1, to Tasks 6, 7 and 8. Task 1, however, presents an abnormal distribution compared with tasks 2, 3, 4 and 5. This is because, while some users were lucky to identify the first point immediately, most users had to inspect multiple points.

	Task							
	1	2	3	4	5	6	7	8
Theorical Minimum	2	2	2	2	2	7	7	10
Median	6.00	4.00	3.00	2.00	2.00	10.00	10.50	11.50
Average	5.65	4.20	2.85	2.55	2.50	10.35	11.40	12.50
Standard Deviation	2.08	1.40	0.91	0.80	0.92	2.76	2.58	2.96
Confidence Interval (95%)	4.65 6.65	3.53 4.87	2.41 3.29	2.16 2.94	2.06 2.94	9.02 11.68	10.16 12.64	11.08 13.92

Table 7.2: Click to complete statistics for each task

7.1.2.3 Errors During Task

Figure 7.3 shows a Tukey box plot with the resulting errors during each task and Table 7.3 shows the computed statistics: median, average, standard deviation and confidence interval.

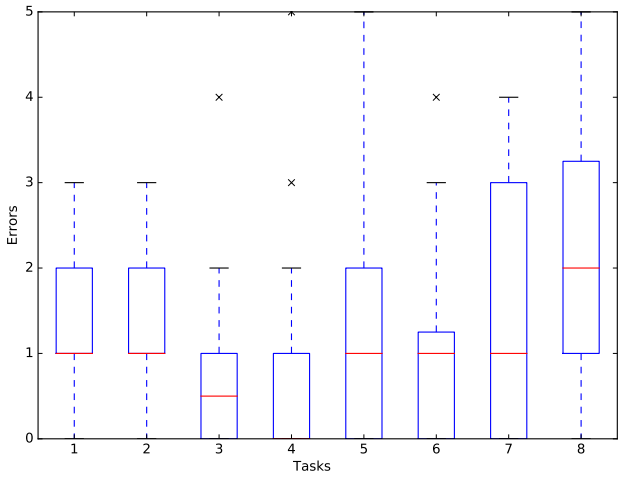


Figure 7.3: Errors during each task

When it comes to errors, we can see that all our tasks are comparable. Regarding this metric, task 5, however, presents a strange distribution. As we can see, using the Figure 7.3, the highest datum still within 1.5 Interquartile range (IQR) of the upper quartile is reasonably high. This is because some users were unable to find the button to change the day to process; most of the errors were made by clicking on the region to edit the name to give the current day.

	Task							
	1	2	3	4	5	6	7	8
Median	1.00	1.00	0.50	0.00	1.00	1.00	1.00	2.00
Average	1.40	1.35	0.80	0.80	1.20	1.05	1.45	2.20
Standard Deviation	0.92	0.85	1.03	1.29	1.36	1.07	1.40	1.54
Confidence Interval (95%)	0.96 1.84	0.94 1.76	0.31 1.29	0.18 1.42	0.55 1.85	0.54 1.56	0.78 2.12	1.46 2.94

Table 7.3: Errors statistics for each task

Even though the complexity of tasks 6, 7 and 8 is higher, and exploratory tasks (tasks 1 and 2) require lots of clicks, we can see that the median and averages are very comparable.

7.1.2.4 Time Per Click for Task

With the time and clicks per task, we can compute the time per click metric, for each task. Figure 7.4 shows a Tukey box plot with that metric and Table 7.4 shows the computed statistics: median, average, standard deviation and confidence interval.

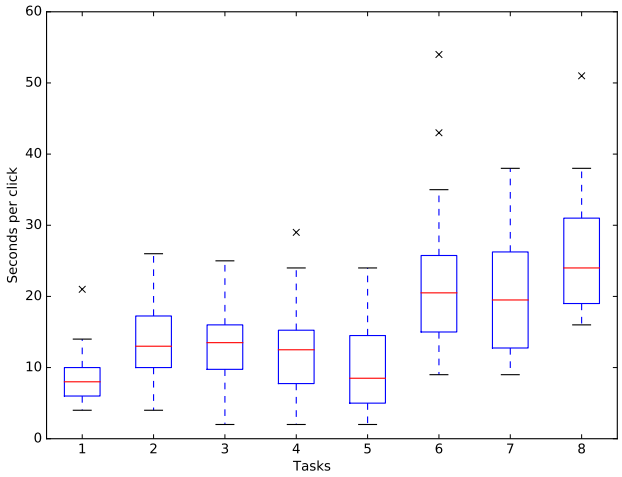


Figure 7.4: Time per click for each task

Figure 7.4 allows us to understand that even though tasks 6, 7 and 8 require more clicks and time (as we have seen previously), those clicks are more time efficient. This is because most clicks in those tasks are done when annotating the current day being processed. The same goes for task 1, where users inspected various points very fast, until they found the correct point.

	Task							
	1	2	3	4	5	6	7	8
Median	8.00	13.00	13.50	12.50	8.50	20.50	19.50	24.00
Average	8.55	13.90	13.60	12.50	10.10	22.60	19.95	26.05
Standard Deviation	3.85	6.02	5.89	6.50	6.26	11.11	8.01	8.69
Confidence Interval (95%)	6.70 10.40	11.01 16.79	10.77 16.43	9.38 15.62	7.09 13.11	17.27 27.93	16.10 23.80	21.88 30.22

Table 7.4: Time (in seconds) per click statistics for each task

7.1.2.5 User ratings and comments

Besides the usage metrics, users where asked questions (in the questionnaire) to help us understand what they thought about *GatherMySteps*' usability and all around usage. For each question, users had to choose between zero (very hard) and five (very easy).

Figure 7.5 shows the results to the question: "Is the system easy to understand?", which averaged a score of 4.1. To the question "Is the system easy to use?", we obtained an average score of 4.15, the distribution of scores is presented in Figure 7.6.

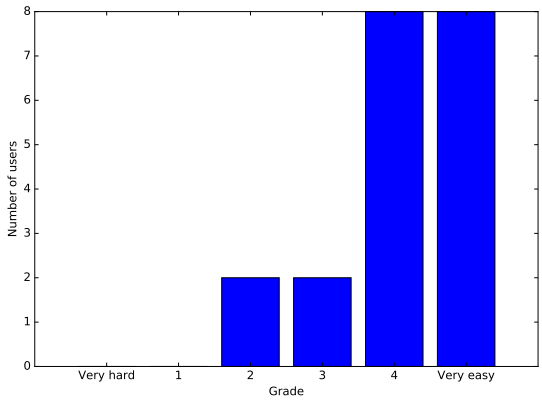


Figure 7.5: User's rating regarding understandability

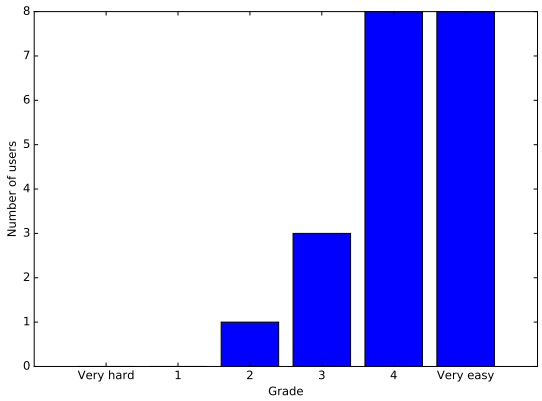


Figure 7.6: User's rating regarding usability

Figure 7.7 shows the results to the question: "Is it easy to process a day?", which had an average score of 4.3. Finally, the question "Is the system easy to learn?" yielded a score of 4.45.

During the tests, we also collected some reactions, comments and suggestions about the system. Users praised the design of the system, the ability to see the track on the map, the ease with which they could edit points individually and the ease of use of the semantic editor.

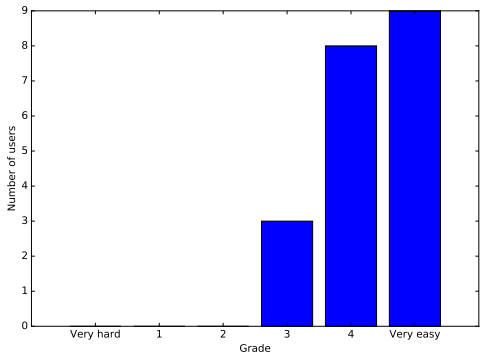


Figure 7.7: User's rating regarding the ease of day-to-day processing

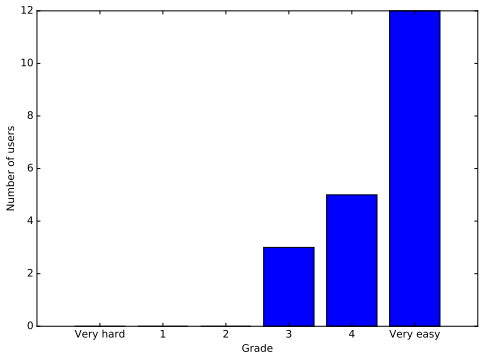


Figure 7.8: User's rating regarding learning

Five of them, however, found annotations to be burdensome, because for each location, they had to mark the span and the trip. One common suggestion (20% of the users) was to make the annotated locations available through suggestions, even before learning them.

Users, 15% of them, also suggested to make the scrollbar always visible in the side panel and the panel to change the day to process more intuitive. From our observations, in task 5, users tend to err by clicking in the name of the day.

7.1.3 Analysis

When we set up to develop usability user tests, we had as our main goal to check whether or not the various features that composed *GatherMySteps* were easy to understand, user-friendly — and since lifelogging is a continuous effort, — we wanted to check if our system was easy to learn. Moreover, we aimed to assess how efficient the processing of a day was.

Regarding the features, we experimented with track editor's features: inspect (task 1) and edit individual points (task 2), split (task 3) and join (task 4) trips.

When it comes to **inspecting points** explored in task 1, we can see that while it requires a median of six clicks to complete (from Table 7.2), it is time efficient (Figure 7.4) with a median of eight seconds per click, and with a small distribution. This results are expected, as this is an exploratory feature/task.

The feature to **edit individual points** was the goal of task 2. This feature is exploratory, but it is action-oriented too, like tasks 3, 4 and 5. From the time (Figure 7.1) and the errors (Figure 7.3), we can see that task 2 is similar to task 1 — boxes are similar — which makes sense since they are exploratory tasks. While time per clicks shows that the clicks are not as efficient as those to inspect points, task 1 has an average of 8.55 seconds per click, while task 2 has an average of 13.90. In that regard, it is similar to tasks 3, 4 and 5.

The **split** feature, which allows one to split a trip into two, was evaluated using task 3. Split a trip is an efficient process; as per Table 7.4, we can say that with 95% confidence that it takes less than 17 seconds per click to execute this task and less than 4 clicks (Table 7.2), values that make it efficient, specially since the number of clicks is not far off from the theoretical minimum of 2 clicks.

To evaluate the **join** feature we devised task 4. When joining the two trips, users were presented with two different options to join the tracks, with different popularities — with different weights. They all identified the correct suggestion, choosing the thicker option. This is translated by the results, requiring less than 3 clicks to execute, with 95% confidence (from Table 7.2) and less than 39 seconds (from Table 7.1). Users, however, were often confused by the display of the suggestions, clicking on the end of a track, which explains the high standard deviation in errors, which was 1.29 errors per task (from Table 7.3).

We also wanted to evaluate how visible it was to **change the current day to process**; to that end we devised task 5. From the results we can see that it is not clear enough how to perform such action. This is a fairly simple and straightforward task, but the results paint a different picture. The error (Figure 7.3) shows an abnormal distribution of errors, even compared to more complex tasks 6, 7 and 8. Most errors were from clicking in the name of the current day.

Auxiliary features like undo, redo and usage of navigation buttons to hide and display a trip, center a trip or the start and end points were not tested directly through tasks. They, however, were part of all tasks. When a mistake was made, users used the undo and redo functionalities and a click was recorded. Navigate buttons allowed users to quickly make sense of the scenarios from tasks 6, 7 and 8. Users that used these buttons tended to complete their tracks with lower times, but more clicks.

The more complex and broad tasks gave users the liberty to process an entire day, making decisions whether or not to edit points, approximate the start and end of a trip to the correct location and use split and join trips. By processing a day, we could also assess how our **semantic editor and the suggestions** fare in a processing scenario.

The liberty to use features such as individual point editing and the joining and splitting of tracks, is shown by the high standard deviation of clicks (between 2.5 and 3, from Table 7.2) and time (between 79 and 102 seconds, from Table 7.1), while some users deemed the results of the automatic processing phase good enough, some decided to make corrections. This usage is expected, since different users have different needs and different thresholds to what is acceptable or not.

To evaluate the **semantic editor**, we can take a look at the errors. If we compare the number of errors for each task (Figure 7.3) we can see that, while time and clicks are not comparable, the errors executed during tasks are comparable. More, tasks 6 and 7 present errors similar to those of focused tasks, which indicate that the semantic editor does not introduce a significant amount of errors.

Tasks 6, 7 and 8, were designed with an increasing degree of complexity. Task 8 requires more annotations and potential corrections than tasks 6 and 7. Task 7 is also more intricate than task 6. Even though there is this complexity involved, we can see that the differences in time and clicks are not significant, which bodes well to overall usability in a day-to-day scenario. The boxes, in Figure 7.4, for

Tasks 6, 7 and 8 are quite similar.

While there are things to improve, when users were asked to rate our system, we obtained a good rating regarding understandability (rating of 4.1), usability (rating of 4.15), day-to-day processing (rating of 4.3) and ease of learning, which scored 4.45.

7.2 Utility Use Cases

While usability tests allow us to assess whether our system is as easy to use as we thought and wanted it to be, they do not provide us with any insight whether it is particularly useful to process personal GPS tracks.

With that in mind, we conducted utility user tests, with five users, selected between friends and family. They were asked to record their everyday trajectories, with the *GPSLogger* app [1], during a period of ten days.

From the twenty users that took part in the usability tests, discussed in Section 7.1, five also participated in the utility. This way they were already familiar with the application, its functionalities and UI.

The test started by uploading their personal tracks to our solution and then they were prompted to process each and every day, making corrections as they saw fit and annotating transportation modes and locations.

The utility use cases were more informal than the usability tests, as the former are an exploratory and qualitative evaluation of our application. Throughout sessions, we asked the users questions about their day and asked them to think aloud, so that we could derive meaning from their usage of *GatherMySteps*. During the usage we collected reactions, comments and suggestions.

7.2.1 Results & Analysis

During the tests, users started by exploring their trips, understanding what they had done. During this initial phase, users skipped the preview stage, mostly because it was difficult to derive meaning from the information presented there.

After automatic processing, users tried to make sense of their day. For the users, this was one of the most interesting parts of the whole experiment. After the automatic processing phase, they spent some time recalling and reconstructing their days. They were excited to find out what they had done in the past. Besides identifying the locations, users were able to recollect the whole day. They also commented on how useful and incredible it was to be able to recollect events in such a detailed matter.

Three users were surprised when reconstructing their days. Since they had recordings — at least — ten days old, they did not remember early days. Some days, they were not sure of the reason behind their trajectories. All of them preferred to use the adjust stage to figure out what was the purpose of their trajectories. This behavior is expected, because users have more freedom and help from the interface to navigate the trips in the adjust stage.

When on the annotate stage, in rare cases, users went back to the previous stage to make more corrections. Either because they missed a trip, highlighted by the unexpected number of annotations to make, or because the annotation of a location was not correct, as they wanted the trip to start or end at the right place.

[1] Created by Mendhak, `https://play.google.com/store/apps/details?id=com.mendhak.gpslogger`, last accessed on October 15th, 2016

They remarked on the quality of the suggestions for locations. Most (60%) did not have to annotate their home and the most frequent places after the first day. The transportation modes, however, were less reliable. They were also impressed by the ease and flexibility of the *LIFE* format.

One user asked if she could specify the intent of her visit at a certain location. Our response stated that it was possible by using semantic descriptions (defined using curly braces); while we do not provide suggestions to those, our editor is able to highlight them. The user was glad that such feature was available, noting it could be very useful if one wanted to be meticulous while annotating their day. This also goes in line with our thought process during the development: to support different needs and usage profiles. Besides the semantic editor, the availability of the configuration panel offers another degree of control and personalization to each and every user. These are signals that our solution is robust and supports multiple user profiles.

They remarked on how useful the suggestions were to speed the annotating process, while pointing out some shortcomings, already discussed in Section 7.1.2.5.

When asked if they would use this application to process their personal GPS tracks, all of them said yes. Regarding geographic lifelogging, they felt that the process of collecting data was not the most convenient. While users would often forget to start and end their recordings at the right times, the alternative — to leave the application record more than one trip, or the whole day — consumed too much battery.

While our work does not focus on the recording of GPS tracks, with *GatherMySteps*, the recording part of geographic lifelogging has become the least user-friendly part of the process, more than the processing an annotation of personal GPS tracks — solved by us.

7.3 Discussion

Back in Chapter 1, we proposed to create interactive tools to efficiently process track data from personal GPS tracks, taking into consideration meaningful personal semantics regarding location and travel information. The result of this desire is *GatherMySteps* and all the works that support it (*ProcessMySteps* and *TrackToTrip*).

To achieve this objective, we needed an application that made it easy to process tracks, that offloaded most of the work from the users, putting them in the position to accept and review their personal GPS tracks. Moreover, such application should satisfy the different needs and frustrations from geographic lifeloggers.

Within this frame, to evaluate an application such as *GatherMySteps* we need to take into account the amount of clicks to execute features, the amount of time users spend processing their days and if the UI is prone to errors. Furthermore, since this is an application to be used frequently to process personal GPS tracks, it is important to understand if we have designed a UI easy to learn.

Overall, the results of the usability user tests show that our application is understandable and user-friendly. This is shown by the low amount of clicks — requiring less than 14 clicks, with a 95% confidence interval of $[11.08; 13.92]$ for the most complex task — and a low amount of time — taking less than seven minutes, with a 95% confidence interval of $[4m33s; 6m12s]$ — to process a single day. From the questionnaire, we also obtain similar results; when asked about the usability of our system, we obtained an averaged score of 4.15. Users also felt that our application was easy to learn (average score of 4.45) and easy to process in a day-to-day basis (average score of 4.45).

This also shows that our goal to simplify the processing of personal GPS tracks of a day was met, by implementing clear and simple features. Users new to the system are able to process days with ease, even though during the test they were working with data personally relevant to them.

In utility use cases, users fed *GatherMySteps* with their own personal information. They showed excitement with the possibility of reviewing their days and recollecting past memories, by processing personal tracks. Users were also able to identify and easily correct errors. All of them recognized the utility of our application, saying that they would use it if they were to continue recording their personal trajectories.

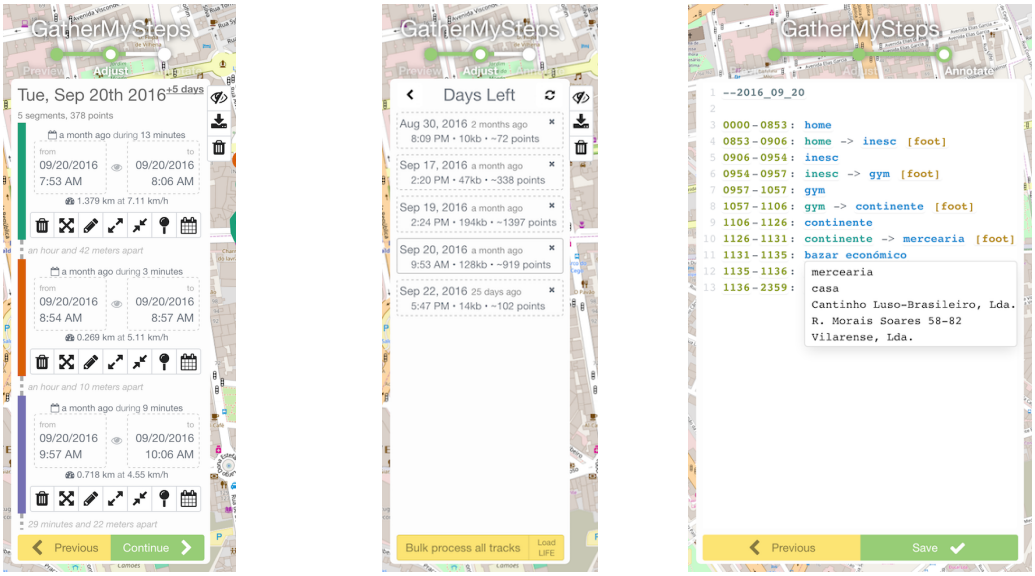
Even though they liked our system and the evaluation was positive, there are some improvements that can be made. Between remarks and suggestions by the users to improve our system and between the results, we can improve:

- Make “change day to process” more explicit.
- Make time difference between trips more explicit.
- In the semantic editor, new locations should be used in suggestions.

With this in mind, we have implemented a new version of *GatherMySteps*, that aims to correct some of the problems itemized above. We show these changes in the Section below.

7.3.1 GatherMySteps 1.1

After we have identified what we could improve in our interface, we have devised an improved version of *GatherMySteps*.



(a) New track/trip presentation (b) New drawer showing days left to process (c) Locally inferred locations

Figure 7.9

Per user suggestions for implementation, as shown in Figure 7.9a, a way to tell the spatiotemporal distance between each track/trip on the left pane. The indicator can also be clicked, to fit the ends of the two adjacent tracks into the map.

During the tests, we realized that the way to change the day to process was not intuitive. The first instinct of them was to click the name of the file. In the new version, we decided to build upon that. To change days to process, users can click the date displayed in bold and at the top of the left pane. This is exemplified by Figure 7.9a and the drawer of the days left, in Figure 7.9b.

One complaint that users had was the lack of suggestions for locations that they had just typed. Now, besides the location inferring made by ProcessMySteps, we infer a set of location suggestions locally. We do this by checking locations adjacent to the one that the user is changing.

Chapter 8

Conclusion

In this dissertation we explored geographic lifelogging, its problems and how to overcome them. The most prominent problems are: the lack of personal tools dedicated to process personal GPS tracks and the lack of accuracy and reliability of GPS recordings. Tools in this area need to have a balance between automatic processing and manual changes, to not overload the user with tasks, while not making decisions that could hinder personal information. Furthermore, lifelogging is a continuous effort, a few unnecessary minutes spent every day, may turn out to be hours wasted after a few months.

A glance over the state of the art unveils projects and systems to process and show GPS recordings, that are designed without introducing users in the loop, to make changes and validate results. For instance, in transportation mode extraction, the user could validate the results, making future predictions more accurate.

Our goal is to put users in the center of action, validating automatic decisions, and correcting annotations for better suggestions as the knowledge base is increasingly improved.

Furthermore, we felt the need to correct GPS recordings — tracks. By transforming a track into a trip, we try to approximate raw GPS recordings to the ground truth, the trajectory done by a user. To that end, we have created *TrackToTrip*, a *python* library that is focused on performing automatic actions on tracks. One of its main functionalities is to transform a track into a trip, by smoothing, segmenting and simplifying a trip. It also allows for information extraction and learning, for locations and transportation modes across trips.

Our main focus, however, was devoted to the UI — the *GatherMySteps* component. *GatherMySteps* streamlines the processing and annotation of personal tracks, in part helped by the automatic processing, and by relevant suggestions made by *TrackToTrip*. It takes three steps to process a day of tracks/trips: preview, adjust, and annotate. The first two steps use a track editor screen, that allows to visually edit a track/trip. The annotate step uses a semantic editor screen, that allows to seemingly annotate a track using drop-down menus or plain text, while displaying the results on the map.

The application is structured in a client-server architecture. The client is implemented by the *GatherMySteps* module, whereas the server is implemented by the *ProcessMySteps* module. *ProcessMySteps*, exposes a REST API to the client, and controls the flow of the application and executes the necessary processing, through the *TrackToTrip* library, for each step.

ProcessMySteps is also responsible to store the information, not only on a database, but also on files, creating backups of the original tracks, and saving the resulting trips and annotations to independent files. This ensures that the personal data of a user does not become inaccessible and hard to export.

We evaluated our work using usability user tests and utility use cases. The usability tests had the help of 20 participants that were unfamiliar with the system. They were asked to perform eight tasks, from where we collected metrics such as: time to complete, clicks to complete and errors during each

task. The results show that our application is efficient when executing small actions, such as: splitting a trip into two, and when processing an entire day.

Five of the participants in the usability user tests also participated in the utility use cases. For that they collected their personal trajectories — using their smartphones — during a period of a week. They had the opportunity to use their personal GPS tracks in *GatherMySteps*. Through a think aloud protocol, we collected their comments and reactions to the system. All of them were excited to remember their past trajectories and experiences. All of them acknowledge the utility of *GatherMySteps* and would use it in the future.

During the evaluation tests, some problems were detected, mainly regarding the lack of visibility or confusion of some features. With that in mind, and having into account suggestions made by the users, we developed a new and improved version of *GatherMySteps*.

To conclude, we have reached our goal to create tools to make the life of geographic lifeloggers easier, with less hassle and from where they can extract meaningful personally relevant information.

Although, if we were to look back and start the whole process again, we might have considered slight changes in our approach. The first one would be to prioritize a working version that entailed all the stages and start adding features from there. Another thing that we would have done differently would be to follow, right from the beginning, a Test-Driven-Development approach — which could save us time when we had to do major refactoring — and piece of mind that our application was doing was it was supposed to do.

Overall this was a challenging, but rewarding, thesis. Besides the obvious technical experience and learning that we obtained in *python*, JS, and the entire technological stack. We learned more about geographic information, lifelogging, and machine learning, their problems and how to mitigate them.

8.1 Future Work

Although our solution is complete and efficient, due to the large scope of our system, there are a few areas that could be improved.

We believe that it is still possible to further improve the automatic processing, mainly the learning and consequent prediction, of personal information. One example of such, is to complete incomplete tracks/trips, between the missing part and the most probable locations. We already have part of the necessary groundwork in place, with location and trip learning.

Further improvements when predicting personally relevant locations and transportation modes should be possible. Locations could also be time clustered, instead of just being spatially clustered. This could prove to be a better solution for people that frequently visit places close together, at different times of the day. It would also be interesting to introduce personal features when classifying the transportation modes of trips and at which time they occur, because usually, we use the same transportation modes for the same trajectories.

The UI could also be improved, to introduce more granular control and more actions. One such feature would be the ability to apply individual automatic processing methods. For instance, apply map-matching to a selected group of points, which could resolve some of the problems with current map-matching libraries. A nice addition would also be to add, remove, or alter learned locations and trips.

Also, it would be nice to introduce more interactive editing of tracks. For instance, in the semantic editor, it is impossible to edit the transportation modes and locations on the map. An interface that allowed that could even be more intuitive and mobile friendly.

GatherMySteps, and all the modules that support it, are a leap forward and with room to grow, regarding geographic lifelogging tools.

Bibliography

Ananthanarayanan, Ganesh, Haridasan, Maya, Mohomed, Iqbal, Terry, Doug, & Thekkath, Chandramohan a. 2009. StarTrack: a framework for enabling track-based applications. *Proceedings of the 7th international conference on Mobile systems, applications, and services - Mobisys '09*, 207.

Bhargava, Preeti, Gramsky, Nick, & Agrawala, Ashok. 2014a. SenseMe: A System for Continuous, On-Device, and Multi-dimensional Context and Activity Recognition. *Pages 40–49 of: Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ICST.

Bhargava, Preeti, Gramsky, Nick, & Agrawala, Ashok. 2014b. SenseMe: A System for Continuous, On-Device, and Multi-dimensional Context and Activity Recognition. *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 40–49.

Bottou, Léon. 1998. Online learning and stochastic approximations. *On-line learning in neural networks*, **17**(9), 142.

Brunauer, Richard, Hufnagl, Michael, Rehrl, Karl, & Wagner, Andreas. 2013. Motion pattern analysis enabling accurate travel mode detection from GPS data only. *Pages 404–411 of: 16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*. IEEE.

Brunsdon, Chris. 2007. Path estimation from GPS tracks. *In: Proceedings of the 9th International Conference on GeoComputation*. National Centre for Geocomputation, Maynooth University.

Chen, Minjie, Xu, Mantao, & Franti, Pasi. 2012. A fast multiresolution polygonal approximation algorithm for gps trajectory simplification. *IEEE Transactions on Image Processing*, **21**(5), 2770–2785.

Cheng, Yizong. 1995. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence*, **17**(8), 790–799.

Cortes, Corinna, & Vapnik, Vladimir. 1995. Support-vector networks. *Machine learning*, **20**(3), 273–297.

Del Moral, Pierre. 1996. Non-linear filtering: interacting particle resolution. *Markov processes and related fields*, **2**(4), 555–581.

Douglas, David H, & Peucker, Thomas K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, **10**(2), 112–122.

Efron, Bradley. 1981. Nonparametric estimates of standard error: the jackknife, the bootstrap and other methods. *Biometrika*, **68**(3), 589–599.

Ester, Martin, Kriegel, Hans-Peter, Sander, Jörg, Xu, Xiaowei, *et al.* . 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. *Pages 226–231 of: Kdd*, vol. 96.

Filipe, Jorge Miguel Saldanha. 2015. Where Have I Been - Visualizing Personal Geolocation Data.

Gil, Rui, & Gonçalves, Daniel. 2016. GatherMySteps. *In: Proceedings of the 2016 Encontro Português de Computaçao Gráfica e Interação*. GPCG.

Goodwin, Graham C, & Sin, Kwai Sang. 2014. *Adaptive filtering prediction and control*. Courier Corporation.

Guttman, Antonin. 1984. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. ACM.

Haridasan, Maya, Mohomed, Iqbal, Terry, Doug, Thekkath, Chandramohan A., & Zhang, Li. 2010. StarTrack next generation: a scalable infrastructure for track-based applications. oct, 1–6.

Hartigan, John A, & Wong, Manchek A. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **28**(1), 100–108.

Hastie, Trevor, & Stuetzle, Werner. 1989. Principal curves. *Journal of the American Statistical Association*, **84**(406), 502–516.

Huang, Yan, Shekhar, Shashi, & Xiong, Hui. 2004. Discovering colocation patterns from spatial data sets: a general approach. *IEEE Transactions on Knowledge and Data Engineering*, **16**(12), 1472–1485.

Javanmard, Adel, Haridasan, Maya, & Zhang, Li. 2012. Multi-track map matching. *Page 394 of: Proceedings of the 20th International Conference on Advances in Geographic Information Systems - SIGSPATIAL '12*. New York, New York, USA: ACM Press.

Killick, Rebecca, Fearnhead, Paul, & Eckley, IA. 2012. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, **107**(500), 1590–1598.

Kirmse, Andrew, Udeshi, Tushar, Bellver, Pablo, & Shuma, Jim. 2011. Extracting patterns from location history. *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '11*, 397.

Lee, Jae-Gil, Han, Jiawei, & Whang, Kyu-Young. 2007. Trajectory clustering: a partition-and-group framework. *Pages 593–604 of: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM.

Lemire, Daniel, & Maclachlan, Anna. 2005. Slope One Predictors for Online Rating-Based Collaborative Filtering. *Pages 1–5 of: SDM*, vol. 5. SIAM.

Meratnia, Nirvana, & Rolf, A. 2004. Spatiotemporal compression techniques for moving point objects. *Pages 765–782 of: International Conference on Extending Database Technology*. Springer.

Newson, Paul, & Krumm, John. 2009. Hidden Markov map matching through noise and sparseness. *Pages 336–343 of: Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. ACM.

Rockafellar, R Tyrrell, & Wets, Roger J-B. 2009. *Variational analysis*. Vol. 317. Springer Science & Business Media.

Scott, ANDREW JHON, & Knott, M. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 507–512.

Umair, Muhammad, Kim, Wan Seok, Choi, Byoung Chul, & Jung, Sung Young. 2014. Discovering personal places from location traces. *Pages 709–713 of: 16th International Conference on Advanced Communication Technology*. Global IT Research Institute (GIRI).

Voshell, Martin. 2004. High acceleration and the human body. *28th November*.

Waga, Karol, Tabarcea, Andrei, Chen, Minjie, & Fränti, Pasi. 2012a. Detecting Movement Type by Route Segmentation and Classification. *Conference on Collaborative Computing: Networking, Applications and Worksharing*, 508–513.

Waga, Karol, Tabarcea, Andrei, Mariescu-Istodor, Radu, & Fränti, Pasi. 2012b. System for real time storage, retrieval and visualization of GPS tracks. *In: International Conference on System Theory, Control and Computing. Sinaia, Romania*.

Waga, Karol, Tabarcea, Andrei, Mariescu-Istodor, Radu, & Fränti, Pasi. 2013a. Real Time Access to Multiple GPS Tracks. *Pages 293–299 of:* Krempels, Karl-Heinz, & Stocker, Alexander (eds), *WEBIST*. SciTePress.

Waga, Karol, Tabarcea, Andrei, Mariescu-Istodor, Radu, & Fränti, Pasi. 2013b. Real Time Access to Multiple GPS Tracks. *Pages 293–299 of: WEBIST*.

Wei, Ling-Yin, Zheng, Yu, & Peng, Wen-Chih. 2012. Constructing popular routes from uncertain trajectories. *Page 195 of: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*. New York, New York, USA: ACM Press.

Zheng, Yu. 2015. Trajectory Data Mining. *ACM Transactions on Intelligent Systems and Technology*, **6**(3), 1–41.

Zheng, Yu, & Xie, Xing. 2010. Learning Location Correlation from GPS Trajectories. *Pages 27–32 of: 2010 Eleventh International Conference on Mobile Data Management*. IEEE.

Zheng, Yu, Wang, Longhao, Zhang, Ruochi, Xie, Xing, & Ma, Wei-Ying. 2008a. GeoLife: Managing and Understanding Your Past Life over Maps. *Pages 211–212 of: The Ninth International Conference on Mobile Data Management (mdm 2008)*. IEEE.

Zheng, Yu, Liu, Like, Wang, Longhao, & Xie, Xing. 2008b. Learning transportation mode from raw gps data for geographic applications on the web. *Page 247 of: Proceeding of the 17th international conference on World Wide Web - WWW '08*. New York, New York, USA: ACM Press.

Zheng, Yu, Xie, Xing, Zhang, Ruochi, & Ma, Wei-Ying. 2008c. Searching your life on web maps. *In: SIGIR workshop on mobile information retrieval*.

Zheng, Yu, Li, Quannan, Chen, Yukun, Xie, Xing, & Ma, Wei-Ying. 2008d. Understanding mobility based on GPS data. *Page 312 of: Proceedings of the 10th international conference on Ubiquitous computing - UbiComp '08*. New York, New York, USA: ACM Press.

Appendix A

Transportation Mode Classifier Evaluation

```
Labels to use:
  foot (4113 samples) aliased from: run (4 samples), walk (4109 samples)
  airplane (14 samples)
  train (811 samples) aliased from: subway (620 samples)
  vehicle (3249 samples) aliased from: taxi (526 samples), bus (1897 samples),
                                       motorcycle (2 samples), car (824 samples)

Labels not used: bike (1582 samples), boat (7 samples)

========================================

Split A (train: #4095, test: #4092)
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='log', n_iter=2500, n_jobs=1,
       penalty='l2', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)

             precision    recall  f1-score   support

   airplane       0.88      1.00      0.93         7
       foot       0.88      0.95      0.91      2056
      train       0.31      0.02      0.04       405
    vehicle       0.76      0.86      0.81      1624

avg / total       0.78      0.82      0.79      4092

Training time: 3.559772s
Predicting time: 0.003526s
Mean squared error: 0.405914
Score: 0.822581
```

```
Split B (train: #4092, test: #4095)
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='log', n_iter=2500, n_jobs=1,
       penalty='l2', power_t=0.5, random_state=None, shuffle=True,
       verbose=0, warm_start=False)

             precision    recall  f1-score   support

   airplane       1.00      0.43      0.60         7
       foot       0.92      0.96      0.94      2057
      train       0.83      0.16      0.27       406
    vehicle       0.81      0.93      0.87      1625

avg / total       0.87      0.87      0.84      4095

Training time: 3.161249s
Predicting time: 0.003893s
Mean squared error: 0.263980
Score: 0.869841

Average score was 0.846211
========================================

Split A (train: #4095, test: #4092)
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')

             precision    recall  f1-score   support

   airplane       0.70      1.00      0.82         7
       foot       0.88      0.92      0.90      2056
      train       0.61      0.46      0.52       405
    vehicle       0.83      0.83      0.83      1624

avg / total       0.83      0.84      0.83      4092

Training time: 0.017156s
Predicting time: 0.002712s
Mean squared error: 0.401271
Score: 0.838465
```

```
Split B (train: #4092, test: #4095)
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')

             precision    recall  f1-score   support

   airplane       1.00      0.43      0.60         7
       foot       0.89      0.96      0.92      2057
      train       0.39      0.45      0.42       406
    vehicle       0.88      0.76      0.81      1625

avg / total       0.84      0.83      0.83      4095

Training time: 0.013183s
Predicting time: 0.002780s
Mean squared error: 0.312576
Score: 0.827839

Average score was 0.833152
```

Appendix B

Trajectory Hausdorff Ratio Pseudocode

```
function distance_similarity is
    input: line segment points (l1, l2) and a point p
    output: number

    d = distance_to_closest_point_in_line(l1, l2, p)
    r = (-1/distance_threshold) * abs(d) + 1

    if r > 0 do
        return r
    else do
        return 0

function line_distance is
    input: two pairs (pa1, pa2) and (pb1, pb2) of points and distance threshold
    output: number

    return abs(
        distance_similarity(pa1, pa2, pb1, distance_threshold),
        distance_similarity(pa1, pa2, pb2, distance_threshold)
    ) * 0.5


algorithm thratio is
    input: two pairs (pa1, pa2) and (pb1, pb2) of points and distance threshold
    output: number

    line_similarity = line_distance(pa1, pa2, pb1, pb2, distance_threshold)
    angle_similarity = abs(
        dot(
            normalize(pa1, pa2),
            normalize(pb1, pb2)
```

```
        )
    )


    return line_similarity * angle_similarity


algorithm thr is
    input: two segments (array of points) A, B, and distance threshold (number).
    output: number

    values := 0
    for each line_segment (pa1, pa2) in A do
        matches := intersects_bounding_box_in(pa1, pa2, B)

        m_values := array()
        for each line_segment (pb1, pb2) in matches do
            similarity := thratio(pa1, pa2, pb1, pb2, distance_threshold)
            m_values.add(similarity)

        if length of value is 0 do
            values.add(0)
        else do
            values.add(max(m_values))

    return average(values)
```

Appendix C

TrackToTrip Default Parameters

C.1 Smoothing

Parameter	Default	Possible values	Description
Algorithm	Inverse	No, Inverse, Extrapolate	Algorithm to use to compensate for Kalman Filter lack of initial predictions.
Noise	1000	$\mathbb{R}$	Expected noise of the track. Higher values for more noise.

C.2 Segmentation

Parameter	Default	Possible values	Description
Epsilon	0.01	$\mathbb{R}$	Density of location clusters. Higher values yield sparser clusters.
Min. time	60 seconds	$\mathbb{R}$	Minimum time at a location or between consecutive points to segment.

C.3 Simplification

Parameter	Default	Possible values	Description
Max. distance error	2.0 meters	$\mathbb{R}$	Maximum distance distance error when simplifying.
Max. speed error	1.0 m/s	$\mathbb{R}$	Maximum speed error when simplifying.

C.4 Location Inference & Learning

Parameter	Default	Possible values	Description
Max. distance	20 meters	$\mathbb{R}$	Maximum location radius.

C.5 Transportation Mode Inference & Learning

Parameter	Default	Possible values	Description
Remove stops	True	True or False	Remove stops from transportation mode identification.
Min. time	60 seconds	$\mathbb{R}$	Minimum time between each transportation modes.

Appendix D

LIFE Specification

D.1 LIFE File Format

Life 1.1 - 26 Jan 2016

D.1.1 Introduction

This document describes the syntax of the LIFE file format. This format is meant to record personal lifelogging information about a person's locations through time. It is intended to be both machine- and human-readable, easy to understand and easy to update.

D.1.2 Concepts

Lifelogging is the practice of recording information about oneself. There are many things that lifelogging practitioners record (food, health, mood, sleep, etc.). One of those are locations and activities performed throughout the day. The LIFE format gives us a standard way to record this information which is simple yet expressive, stemming from years of lifelogging practice.

The format is centered around *days*, a given calendar day. For each day, a number of *spans* describe where the user during a particular time interval. Intervals for which no span is specified are considered to be *travelling time*. Another way to look at it is that a span usually represents a time where you were "indoors", for some time at some place doing something meaningful, and their complement is when you were walking or driving somewhere.

Spans mention *places*. A place is somewhere that you were at and it is usually described in a personally relevant way. A place is assumed to be tied to a specific physical location. Hence, if there are two 'Tesco' supermarkets in two different streets, the place names should be difference (e.g.: 'Tesco-Foo St.' and 'Tesco-Bar St.').

An exception are what we call *trips*. A trip is a special kind of span where instead of just one place, two are mentioned. These are the origin and destination of a trip the user has made. They can be used to annotate useful information about that trip (see below).

Each span can be annotated with tags and a personally relevant comment string that can be used, for instance, for short descriptions of what was the purpose of that span (a movie that was seen, person that was met, etc.). Also, additional tags for specific periods of time inside a span can be specified.

Finally, LIFE gives support for several meta information regarding the categories of places, their physical location, place inclusion (a restaurant inside a shopping mall, for instance), etc.

D.1.3 Syntax

D.1.3.1 Day

One day is represented by a header indicating the date. The format for the date is `"yyyy_mm_dd"`, and it must be preceeded by two hyphens (`"--"`). After the header, there should be a line per span, as described below. All spans should be contiguous (no blank lines in between). There should be spans covering the entire day.

D.1.3.2 Span

A span, in its simplest form, is represented as follows:

```
<start>-<end>: <place>
```

`<start>` and `<end>` are the starting and finishing time of the span. Precision is down to the minute. They follow military time standard, `hhmm`: 3:32PM becomes `1532`. 8:21am is `0832` and so on. Times begin at `0000` and end at `2359`.

`<place>` can be any string, for a personally relevant place name ("home", "brother's home", "sam's school", etc.). A complete span could be:

```
1241-1343: mcdonalds close to work
```

A span must start at the beginning of a line (no whitespace).

Tags and Semantics Optionally, a set of tags can be specified by enclosing them im square brackets ("`[]`"). Different tags should be separated with a pipe character ("`|`"). Also optionally, a *semantic description* can be added inside curly braces ("`{}`"). An example:

```
2124-2349: neighborhood cinema [movies|leisure] {Star Wars: The Force Awakens}
```

Trips A trip is a special kind of span represented by including the departure and arrival place, separated by `->`. The span is assumed to represent actual travel time, not time spent at either place. If you need to specify time spent there, you should explicitly include a span for it:

```
1026-1032: saldanha station
1032-1045: saldanha station->rossio station
1045-1047: rossio station
```

Sub-Span Annotations Sometimes, it may be relevant to provide additional information about some of the things done in a particular span. For instance, a trip may have consisted of different stretches, using different modes of transportation. So, while the user may want to specify only one Trip, it may be interesting to specify the time intervals, inside it, where those different modes of transportation were used. To do that, we use *sub-span annotations*.

A sub-span annotation is written by indenting the line using at least one space character (`' '`). After, the user can specify either a time interval (`1254-1301`), in the same format as for a span, or a single time instant (`1643`). This will tell either the **interval** or the **instant** that is being annotated. Then, after a semicolon, a set of tags in the same format as for spans can be specified. For instance, for interval annotations:

```
1254-1335: work->home ; a trip from work to home that we'll annotate below
   1254-1303: [walk]
   1303-1311: [stop]
   1311-1328: [bus]
   1328-1335: [walk]
   ; annotations for mode of transportation
```

or, for instant annotations:

```
1823-2102: le bistro [dinner]
   1943: [ice cream]    ; an ice cream was eaten at that time
```

Interval Sub-Span annotations could be replaced by actual spans or trips for that interval, of course. They should be used when the actual place names do not matter. There is no obligation for the annotations to cover the entire time interval of a span or trip. For instance, if the user only wants to record bus travels:

```
1254-1335: work->home ; a trip from work to home that we'll annotate below
   1311-1328: [bus]
   ; annotations for bus travels only
```

D.1.3.3 Timezone

Sometimes, when moving, we change timezones. We can specify this by indicating the timezone directly in the file. Timezones are represented as offsets from UTC and take the for of `UTC(+/-)<offset>`. For instance, `UTC+2`, `UTC-5`, etc. are all valid. You can also use `UTC` alone, for that timezone. Timezone commands should appear inside a day and are in effect from that moment on, for all spans of that day and subsequent ones, until another timezone command is found. So, usually, in the first day of a file, you'd start with a line for the timezone where you usually are in.

Sometimes, a timezone changes as the result of an indoors trip. For instance, you can board a plane in one timezone and leave it in another. In that case, the correct choice is to see that the departure place of the indoors trip is in one timezone and the finishing place in another. We can specify this by starting an `UTC` command with an `@` sign. So, in the following case, `LIS Airport` would be in `UTC` and `CDG Airport` in `UTC+1`. All spans after the indoors trip are also in `UTC+1`.

```
--2011_05_23
UTC
0000-0512: home
0543-0735: LIS Airport
@UTC+1
0735-1049: LIS Airport->CDG Airport
1153-2359: Hotel Foo
```

Please note that daylight savings time will change your timezone. So, the day they start (or cease) being into effect you should explicitly note this change in the file:

```
--2011_03_29
UTC+1
0000-0812: home
0843-1235: work
1243-1330: restaurant [lunch]
1338-1749: work
1823-2359: home [dinner]
```

D.1.3.4 Meta Commands

There are several commands that, while not describing a span, convey information that helps understand them. They must be entered *outside* of a day. They all start with an at sign (`@`). Spaces are allowed as padding around the place names and other parameters. Unless otherwise noted, `@` commands should be global: regardless of where they appear in the file, they apply to all days/spans in it.

Place Inclusion Some places are within other, e.g. a particular store inside a shopping mall. How to record it? If you record the store as its own place, you wouldn't know you were in the mall at that time. If you record the mall, information about where you actually went in there would be lost. To circumvent this problem, we can specify place inclusion by connecting a sub and super-place with a < sign:

```
@a store in the mall<the mall
```

Canonical Locations It may be relevant, for some applications, to know exactly where (geographically) a place is located. You can use `@` to connect a place with its location, in decimal `lat,lon` format:

```
@work @ 32.4343534,-9.54353
```

Place Categories We can specify categories for places. This will allow you to see your life in broader categories (when you spent time shopping, or going to movies, etc.). This is done by entering the place name, a semicolon and the category name:

```
@Radio Shack:Commerce
```

Category commands are time-sensitive:the category applies for spans after it has appeared. A second category command for a same place replaces that category for that place for that moment onwards. This allows for categorical changes. For instance, a place could be a 'Restaurant' until you go work there, and then it becomes 'Work'.

Name Changes Over large periods of time, places can change names. A restaurant can close and another one open in the same place with another name. Name changes can be specified with a >> connecting the old and new place names. Unlike other `@` commands, this one is time-sensitive: the name change is supposed to occur at the time the `@` command appears.

```
@ye olde restaurant>>super-duper new bisto
```

D.1.3.5 Comments

Comments can be entered into the file by preceeding them with a semicolon (`;`). Everything after it will be considered to be a comment.

D.1.4 Usage

The point of a LIFE file is to record a person's live from their point of view. Things that are personally relevant, in the way they are so. It should usually begin with a first day, the first line of which will be the default timezone (don't forget to take into account the daylight savings, if relevant). Then we will find a succession of days. Each day should have a span for whenever the user was in some place. The names of the places, as stated above, aren't necessarily the "official" names, but rather what that place means for the user. Place Inclusion can be used to tie those names to the "official" ones if needed.

If done properly, from a LIFE file a user should be able to tell where s/he was at any given moment in time and (if annotated with semantics) what was the purpose of the visit. Categories and tags can also help with this: a place categorized as a restaurant will probably be visited for a meal. It is up to each user to decide what different tags, categories, etc. represent. One person could use tags to annotate the names of people met at a given place. Another could use it to note where dinner or lunch were had, etc.

One important thing is to maintain things consistent. The same place name should be used recurrently to represent the same place, for instance. Failure to do this may make it hard to compare across spans. This is easier than it appears since for most people there will be a somewhat small set of recurrently visited places (home, work, etc.) and a few that are seldom visited. The recurrent ones will quickly be mechanized. And, of course, tools can be developed to help maintain a LIFE file.

D.1.5 Examples

Below is a sample LIFE file. It shows several use cases for the format and, as such, is heterogeneous in representation level. Usually you would want to avoid this and choose an uniform amount of detail and conventions for your LIFE files.

```
; LIFE for John Doe

; First couple of days: the simples use case. Just record when you were somewhere
--2016_01_01
UTC                          ;setting the "default timezone"
0000-1231: home
1245-1356: McDonalds         ;quite sad, lunching on a mcdonalds on New Year's Day...
1413-2359: home

--2016_01_02
0000-1002: home
1139-1602: home              ;went for a walk and came back home
1703-1830: shopping mall
1830-2045: beef delight
2201-2359: home

@beef delight<shopping mall   ; the restaurant is inside the mall!
```

```
; now, we'll see some tagging and annotation. For instance, John decides to tag where he eats,
; and which movies he watches

--2016_01_03
0000-0721: home
0801-1202: work
1223-1303: Pizza Place [lunch]                  ; tagged as a lunch place
1321-1802: work
1827-2101: Stellar Cinemas [movies] {Deadpool}  ; tagged as movie theater, semantics tells which movi
2107-2143: burger king [dinner]                 ; tagged as a dinner place
2227-2359: home

--2016_01_04
0000-0725: home
0807-1215: work
1231-1330: Le Bistro [lunch]
1342-1812: work
1837-1845: supermarket
1855-2359: home   [dinner]


; next day, John will travel by plane to meet a client

--2016_01_05
0000-0810: home
0000-0912: LIS                 ; Lisbon airport
@UTC+2                         ; in the next span, the first time will be UTC, the second UTC+2!
1055-1655: LIS->ATH  [lunch]   ; The trip was actually only 4 hours, but we changed timezones. Had lu
1655-1732: ATH
1901-2359: hotel [dinner]


; next day, we're still in UTC+2
--2016_01_06
0000-0801: hotel
0831-1243: client
1301-1358: QuickFoods [lunch]
1403:1754: client
2003-2359: hotel [dinner]


; heading back home! New timezone change!
--2016_01_07
0000-1143: hotel
```

```
1232-1403: ATH [lunch]   ; had lunch at the airport
@UTC
1404-1632: ATH->LIS      ; 4 hour trip again, but appears to be two due to timezone change
1632-1743: LIS
1831-2359: home



; John now decides to annotate his mode of transportation. First, using tags on trips. Not relevant w
; changed transport, only that he used different mods
--2016_01_08
0000-0715: home
0715-0801: home->work [walk|bus]
0801-1216: work
1216-1229: work->Le Bistro [walk]
1229-1342: Le Bistro [lunch]
1342-1358: Le Bistro->work [walk]
1358-1812: work
1812-1837: work->home [walk|bus]
1837-2359: home   [dinner]



; Now using a different trip line for each leg of the trip, to know where and when he changed mode of
--2016_01_09
0000-0705: home
0705-0712: home->bus stop close to home [walk]
0712-0734: bus stop close to home
0734-0801: bus stop close to home -> bus stop close to work [bus|bus:767]  ; rode bus 767
0801-0811: bus stop close to work -> work [walk]
0811-1211: work
1211-1219: work->Le Bistro [walk]
1219-1331: Le Bistro [lunch]
1331-1350: Le Bistro->work [walk]
1350-1822: work
1822-1824: work->bus stop close to work [walk]
1824-1825: bus stop close to work
1825-1841: bus stop close to work->bus stop close to home [bus|bus:767]
1841-1848: bus stop close to home->home [walk]
1848-2359: home   [dinner]

; 'bus:767' is a normal tag. John decided that he would annotate all bus lines using the 'bus:<number



; John decided he doesn't need to know the places where he gets on buses, etc. after all.
; so he'll use simply the sub-span annotations
```

```
--2016_01_10
0000-0705: home
0705-0811: home->work
   0705-0712: [walk]
   0734-0801: [bus|bus:767]
   0801-0811: [walk]
0811-1211: work
1211-1219: work->Le Bistro [walk]
1219-1331: Le Bistro [lunch]
1331-1350: Le Bistro->work [walk]
1350-1822: work
1822-1848: work->home
   1822-1824: [walk]
   1825-1841: [bus|bus:767]
   1841-1848: [walk]
1848-2359: home   [dinner]
   2201: [water:33cl]          ; drank 33cl of water at 2201
```

D.1.6 Changelog

v1.1 - 26 Jan 2016

- Extended support for recording trips between places
- “Indoors trips” are now just trips
- Introduced “Span Annotations”, both interval and instant
- Added “Examples” section

Appendix E

Questionnaire

This chapter contains the questionnaire that the participants in the usability user tests answered, after executing the tasks.

GatherMySteps

* Required

1. **Idade** *

.....

2. **Sexo** *
Mark only one oval.

- ( ) Feminino
- ( ) Masculino

3. **Utilizas o GPS do teu smartphone regularmente?** *
Mark only one oval.

- ( ) Nunca
- ( ) Pelo menos uma vez por semana
- ( ) 1 a 3 dias por semana
- ( ) 4 a 6 dias por semana
- ( ) Todos os dias

4. **Para que usas o GPS do teu smartphone?**
Check all that apply.

- [ ] Aceder a mapas
- [ ] Lifelogging
- [ ] Encontrar amigos
- [ ] Other: .....

5. **Fazes, ou já fizeste, geo lifelogging?** *
Mark only one oval.

- ( ) Sim
- ( ) Não

6. **Se não, porquê?**
Check all that apply.

- [ ] Não sei o que é
- [ ] Não tenho tempo
- [ ] É dificil/aborrecido
- [ ] Não tenho o equipamento necessário
- [ ] Other:

7. **O sistema é fácil de entender?** *
Mark only one oval.

	1	2	3	4	5	
muito difícil	◯	◯	◯	◯	◯	muito fácil

8. **O sistema é fácil de usar?** *
Mark only one oval.

	1	2	3	4	5	
muito difícil	◯	◯	◯	◯	◯	muito fácil

9. **É fácil processar um dia?** *
Mark only one oval.

	1	2	3	4	5	
muito dificil	◯	◯	◯	◯	◯	muito fácil

10. **O sistema é fácil de aprender?** *
Mark only one oval.

	1	2	3	4	5	
muito dificil	◯	◯	◯	◯	◯	muito fácil

Appendix F

Usability User Test Handout

This chapter contains the handout that the participants, in the usability user tests, were given.

GatherMySteps

Testes de usabilidade com utilizadores

GatherMySteps é uma aplicação para facilitar o processamento de tracks GPS, no âmbito de lifelogging, permitindo um processamento semi-automático, dando liberdade ao utilizador de fazer alterações a decisões feitas pelo sistema.

As tracks são agrupadas por dias, processados um a um. O processamento de casa dia consiste em três passos: (1) pré-visualização das tracks do dia, (2) ajustar alterações automáticas, e (3) anotar o dia com informação semântica.

Nos dois primeiros passos é possível editar tracks de forma granular, por exemplo, dividindo cada track em duas, juntar duas tracks temporalmente contíguas, ou editar individualmente cada ponto de uma track. É também possível inspeccionar uma track, segmento a segmento.

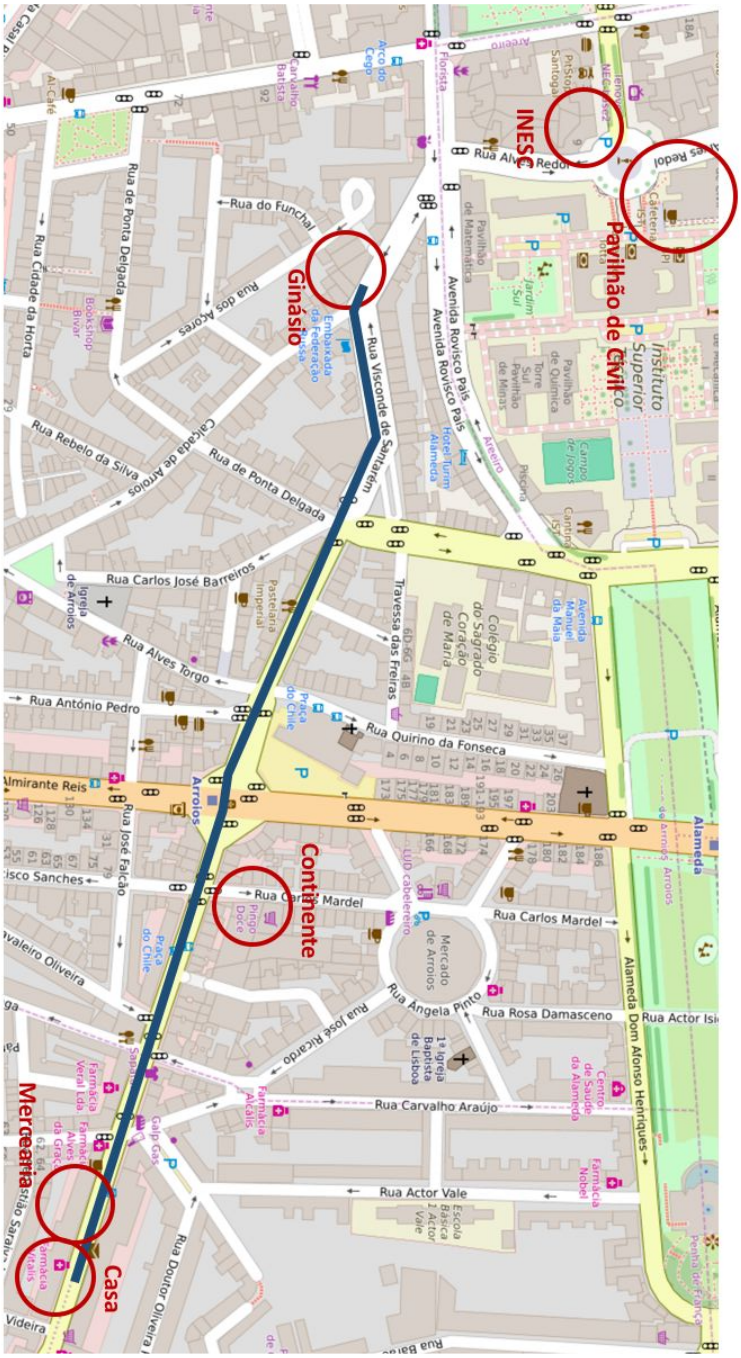
No último passo, é utilizado o formato LIFE para anotar localizações relevantes, e meios de transporte usados em viagens.

Abaixo tem um exemplo de um dia (10 de Setembro, de 2016) anotado com o formato LIFE, são apresentados também os trajectos e localizações mais frequentes, considerados no cenário que vai testar.

Exemplo do formato LIFE

```
--2016_09_10
0000-0712: Home    estadia em casa, da meia noite (0 horas, às 7h12 da manhã)
0712-0730: Home -> IST [foot]  viagem de casa para o IST, a pé
0730-1200: IST
1200-1215: IST -> Home [vehicle]
1215-2399: Home
```

Localizações e Trajetos comuns



Procedimento do teste

Depois de ler este documento por completo, e para se ambientar com o programa, o seu supervisor irá fazer uma série de pequenas demonstrações de funcionalidades. Antes de iniciar os testes, terá ainda a possibilidades, de você mesmo, de explorar a aplicação. Durante esta fase não está a ser testado.

Depois disto, e assim que desejar, ser-lhe-á dada a ordem (aleatória) de tarefas a cumprir, cada uma com tempo máximo predefinido. Durante a sessão de teste o supervisor irá tomar notas, para posterior análise.

No final ir-lhe-à ser entregue um pequeno questionário, com um tempo máximo de preenchimento de 3 minutos.

Tarefas

1. Dia 30 de Agosto, 2016: Identificar, oralmente, as coordenadas do ponto às 7:10pm.
2. Dia 30 de Agosto, 2016: Adicionar dois pontos, de forma a que a track tenha inicio em *casa*.
3. Dia 22 de Setembro, 2016: Separar track perto da *Mercearia*.
4. Dia 22 de Setembro, 2016: Juntar as duas tracks pelo caminho mais comum.
5. Mudar o dia a processar para 13 de Julho, 2016.
6. Dia 17 de Setembro, 2016: Saí de *casa*, à 1:20pm, para ir ao *ginásio*, onde estive até às 2:27pm. A essa hora saí para ir ao *Pavilhão de Civil*. Saí de lá às 5:19pm e voltei a casa.
7. Dia 19 de Setembro, 2016: Saí de *casa*, à 1:24pm, para ir ao *ginásio*. Acabei o treino às 2:35pm e voltei a *casa*. À 2:37 saí de casa e apanhei o *autocarro (bus)* até ao *Starbucks.*
8. Dia 20 de Setembro, 2016: Saí de *casa*, às 8:53pm, para ir ao *INESC*, onde estive até às 9:54pm. A essa hora saí para ir ao *ginásio*. Ao fim do treino fui às compras ao *Continente*, de onde saí às 11:26pm para ir a uma *mercearia*. Terminei a minha manhã em *casa*.

Obrigado pela sua colaboração!