POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Design and implementation of a data-driven system for bus crowding analysis

Supervisor
prof. Giovanni Malnati

Candidate
Francesco Palma

Academic year 2017-2018

Summary

Public transport planning is a crucial aspect for companies, operating in the public transport, which seeks to improve their quality of service. In the last years, the trend of using IT solutions has been increasing due to the advancement of IoT technologies. Indoor localisation is one of those technologies allowing to locate the position of devices, such as smart phones, in closed environments. Locating and counting each device in an area allows estimating the crowing level of the area. If the area is a bus, then the count will indicate the number of people on board. This thesis work aims at designing and implementing a software solution built on top of a passive device detection system installed on several buses that collects and transmits the data of each device detected on board to a remote server.
The built solution analyses the data on the remote server doing two main operations that involve classification. The first operation is recognising the line the bus is following based on the path followed by the bus where the sensors are installed. The second operation consists in determining for each valid device detected on the bus the ascent and descent time. Combining the results of this two operations what is obtained is the utilization of a line. In addition to the calculation of metric, it has also been developed a graphical representation of the bus utilization for each line followed by the buses where the system is deployed.
The error of classification has been calculated for both operations resulting in low error levels that prove the metric is reliable and ready to be used to improve and support the creation of a public transport plan.

Contents

List of Tables 6

List of Figures 7

1 Introduction 9

2 Problem description 11
2.1 Public transport planning 11
2.1.1 Transport costs 11
2.1.2 Demand and offer 12
2.1.3 Counting techniques 13
2.2 Context of the project 14
2.2.1 Starting point 14

3 Architecture 16
3.1 Existing system architecture 16
3.1.1 Overview 16
3.1.2 Endpoint data model devices 18
3.2 Objectives and requirements for the software extension 21
3.2.1 Requirements of the software extension 22
3.3 Software architecture 23
3.3.1 Functional components 24
3.3.2 Functional component: Classifier 25
3.3.3 Functional component: Data visualizer 35

4 Implementation 37
4.1 Environment choice 37
4.1.1 Existing system solution 37
4.1.2 Extension solution 37
4.2 Classifier implementation 39
4.2.1 Main loop 39
4.2.2 Acquiring the data 40

4.2.3 Preprocessing input data 42
4.2.4 Device analysis implementation 44
4.2.5 Itinerary Matching 48
4.2.6 Metric computation and storage 54
4.2.7 Configuration 55
4.3 Data visualizer 56
4.3.1 UI 56
4.3.2 Data access layer 63

5 Results 67
5.1 Requirement validation 67
5.1.1 R1 - Device Filtering 67
5.1.2 R2 - Path classification 69
5.1.3 R3 - Data aggregation 72
5.1.4 R4 - Readability 73

6 Conclusion 74

Bibliography 76

List of Tables

3.1 Content of the stop table before grouping. 36
3.2 Grouped content ready to be shown. 36

4.1 Configuration parameters for the Classifier component. 66

List of Figures

3.1 Functional schema of the MQTT protocol. 18
3.2 Class diagram representing the coordinate and speed of the bus at a certain point of time. 19
3.3 Class diagram representing a device detected by a sensor having identifier equal to "IMEI" 19
3.4 Parameters section of MyMoby UI. 21
3.5 Device list section of MyMoby UI, by clicking on a device ID is possible to show it on the map. 21
3.6 Positions of a device shown on the map section of the UI. 22
3.7 Picture showing on of the roads that the device passed by. 23
3.8 One of the bus stop this devices passed by. 24
3.9 Software architecture representation. 25
3.10 Class diagram of the "StopTable". 26
3.11 Class diagram representing the bus line data. 28
3.12 Class diagram of a "BusPath". 29
3.13 Class diagram of a "Segment". 30
3.14 Class diagram of SegmentClassification. 30
3.15 Class diagram of a "LineProbability". 31
3.16 Mathematical model for the P function, indicating the probability that the bus is serving a specific itinerary. 32
3.17 Conceptual representation of the metric's graph. 35
3.18 Class diagram of metric in a period of time for a specific itinerary of a bus line. 36

4.1 Http request made using HttpUrlConnection. 41
4.2 Retrofit library utilisation example. 41
4.3 Plots of the speed of the bus in two different days. 43
4.4 Example of a case where the bus is waiting at the last stop before starting a new itinerary. 44
4.5 Behaviour of a device not belonging to a human. 45
4.6 Device detection probably coming from an user device. 46
4.7 Device detection probably coming from a fixed IP device. 46

4.8 Structure of a list element returned by the devices endpoint. 47
4.9 Code for the enlarging time window. 49
4.10 Mapping between a router ID and a sequence of bus line identifiers. On top is specified the router ID while on the bottom there is the sequence of line ids. 50
4.11 Representation of a itinerary's point inside MongoDB. 52
4.12 Example of use of the "GeographicLib" to select the middle point of the line connecting the first point (lat1, lng1) and the second point(lat2, lng2). 52
4.13 Data structure used in the "LineProbability" class. 53
4.14 Interval of validity where the metric will be calculated. 54
4.15 StopsTable entry in the db. 55
4.16 Graph showing the metric between 3 bus stops. 58
4.17 Map section of the UI. 59
4.18 Dinamic transformation of the font-size of several paragraph depending on the data bound to the paragraphs. 59
4.19 Creating a rectangle SVG element using D3 library. 60
4.20 Node data structure. 60
4.21 Creation of a Leaflet's map object and introduction of a tile layer coming from OpenStreetMap. This operation will display a map in a div having as id 'mapdiv'. 61
4.22 Function to represent a bus stop on the graph. 62
4.23 Function to draw the path connecting two stops on the map. 63
4.24 MongoDB aggregation syntax example. 64

5.1 Quick search on MyMoby devices' list. 67
5.2 Console's output of the filter. 68
5.3 Device belonging to an human. 68
5.4 Device not belonging to an human, probably filtered because all its detections are concentrated at the end of a road. 69
5.5 Chart showing the error rate on the device classification in the time period corresponding to a month of detections. 70
5.6 Validation data structure for itinerary matching. 70
5.7 Graphical user interface of the website of the transport society used to validate the itinerary matching algorithm. 71
5.8 Chart showing the error rate on itinerary matching in the time period corresponding to two weeks. 72

Chapter 1

Introduction

In the last decade, the trend of utilising IT solutions in the scope of public transport has been increasing. The motivations behind this increase have to be searched in the technological advancement and new solutions coming from the IoT world that integrates well with the transport services.
Some of the solutions implemented in the public transport scope are real-time vehicle tracking, unexpected event notifications, personalised travel solutions based on the typical route a person does. All these solutions account for the needs of the users of the transport system. Other solutions provide to transport societies tools to monitor and extract metrics on their services. This metrics can be utilised to support decision making. This thesis work focuses on the solutions for transport societies, in particular on Automatic Passenger Counting systems which are systems used to count the number of people on board of the bus, without any user interaction and are based on sensors of a different kind.
Automatic Passenger Counting provides useful data to improve schedules, plan new itineraries or update the existing ones, allocate more buses at specific times of the day, and etcetera. This kind of systems are already in use by several transport societies, but there is still interest in improving them due to the return of investment they provide to transport societies.
The project associated with this thesis aimed at researching the potential offered by this systems by focusing on a particular typology of them which uses the technology of passive device detection. This technology relies on the signals personal devices emit to locate them in a closed environment.
This document will detail more the scope of public transport planning, introducing the advantages that systems of passive device detection have in respect to the classic methods in chapter 2. The following chapters are structured as follows: first the developed solution will be described abstractly in chapter 3 describing the main points of the existing system of data collection and how on top of it the design has been extended to reach the thesis's objectives. In chapter 4 the methods followed to implement the design along with their technical details will be discussed, also

showing some examples from the software developed.
In the end, in chapter 5 the results of this project will be discussed, proving the grade at which the objectives of this thesis have been reached.

Chapter 2

Problem description

Public transport planning is a crucial aspect for companies, operating in the public transport, which seeks to improve their quality of service. The deployment of a transport plan requires the collection of feedback on its performance. Acquiring feedback is difficult and not always accurate with manual methods because they rely on human analysis over substantial data sets. For this reason, more transport societies are integrating innovative technologies coming from the IT domain to simplify this process.
This chapter will first introduce some concepts about public transport planning and then the context of the project.

2.1 Public transport planning

2.1.1 Transport costs

Every task of transport has a cost because it implies consumption of energy needed to win the floor, air and water attrition; also time is consumed which is another critical resource. In general, transports cost are classifiable as the cost for the users and cost of the providers of the transport service.
The price for the transport providers can be divided into the cost of investment (vehicle fleet, deposits) and the cost of operation (employee cost, fuel). This costs should be covered by the revenue coming from the service selling, but not always that the case. This happens, in some measure, because the prices appliance is subjected to limitations from public administrations to guarantee service to most of the community.
Moreover, transport societies are obliged to provide transport services also in zones where the demand level is not high enough to justify an investment, but the service has to be provided in any case for reasons of social equity. In this cases, the community intervenes financing in part, filling any deficit coming between the difference among cost and revenue. This costs mentioned before are increasing over

time, so transport society seeks new ways of improving their service. Nowadays, it is more and more spread the utilisation of ITS[1], that are software systems supporting the public transportation sector, allowing for real-time monitoring of vehicles, passengers that provide useful information in short time to the users.

2.1.2 Demand and offer

Public transportation planning means to choose a series of actions to be performed to reach some objectives aimed at improving the quality of services offered to increase user satisfaction and revenues. For public transport societies, this will imply updating their offer, which is constituted by all offered services (itineraries, allocation of vehicles on them and number of rides).
Before defining the offer and measure its effectiveness it is necessary to study the demand for mobility, that is the common interest in having two points (origin and destination) connected through any means of transport. There are two ways of estimating the demand for mobility:

Mathematical models Mathematical models can estimate the current demand of mobility and its evolution in function of socio-economic characteristics of the geographic area of study and on the transport system deployed in it.

Direct investigations Direct investigations consist in interviewing users to identify their needs of mobility.

Other methods may involve analysing the flux of vehicular traffic in entrance and exit of a specific zone.
Once the demand is defined it is possible through evaluations, supported by modern software decision systems, to define the mobility offer of the transport society.
What is crucial for transport society is to have feedback about the performance of a transport plan which can be done evaluating several metrics. One important metric that is of particular interest for transport societies is the utilisation of their bus lines. The utilisation of a bus line can be calculated by counting the number of people present on the bus while the bus is serving that line. Several methods can be used to produce the count of people on board, some of them use manual techniques while more recent IT systems, called APC[2], automate the process and give higher quality data with less effort.

[1]Intelligent Transport Systems

[2]Automatic Passenger Counting

2.1.3 Counting techniques

As said before, the counting techniques can be divided in manual and automated. The most common manual method is:

Direct counting With this method, the passengers are counted by sight by an operator present on the bus. This method suffers mainly in case of bus overcrowding due to sight obstruction. Also there are other sources of error, like lack of attention or tiredness of the operator.

The automated methods rely on the use of sensors, which can be of different nature, or machines installed on the bus and do not rely on human interaction. There are several methods for automated counting:

Direct counting This method which has also been classified as manual has a respective automated version when video cameras are installed on board of the bus which run image recognition algorithms able to detect the human shapes and track their movements on the bus. This method has an high accuracy, but high costs from an implementation and computational perspective.

Counting based on ticket This counting method relies on accessing the data contained in ticket validations machine to count the number of tickets validated, which give an insight on how many people were on a vehicle at a given time. There is no problem of counting a person twice as each ticket is uniquely identified in the validation machine registry. This technique is useful in contexts where the way in and the way out are accessed through ticket validation, like for example metro' stations. This technique presents instead several limitations, that affect the count and the computable metric, in the context of a bus for several reasons:

- The access to the vehicle is not bound to the validation of the ticket.
- Impossibility of reaching the ticket machine in situations of overcrowding.
- In a bus the ticket is not revalidated on exit, so the information on where a person left the bus is unknown.

Also this technique always requires an interaction of the user. This method can provide real time processing if the validation machines are able to transmit over a network the validations events.

Access counting Using sensors to detect a person physically passing through a door, which can be the entrance door and the exit door. The main limitation is that sometimes due to crowding a person can be accounted several times and also the sensors tend to be quite expensive.

Passive detection This kind of method relies on a different kind of sensors and approach. The entity counted is not any more a person, but the device which the person owns, because it emits different kinds of radio signals associated to technologies like Bluetooth or WiFi which in their processes of discovery allow the device to be tracked by sensors. This method among the one presented so far has the lowest implementation cost which can be divided into the cost of the sensors and the cost of deployment. One of the main problems in mobility context is that devices external to the bus can be detected (nearby cars, IP cameras).

Passive detections methods have little implementation costs, produces a high volume of accurate data and it provides the transport society with a broader sample of people to perform statistic analysis, meaning that if further developed and improved these methods can become a good investment for the transport societies seeking an improvement in their offer.

2.2 Context of the project

In the last section, the counting methods have been introduced highlighting the advantages of the passive device detection.
APC systems based on passive detections are already existent in different varieties, but typically they are based on proprietary solutions that have a strong implementation cost because this systems need to integrate with the transport society already existing IT systems and databases and also require training for employees to use them.
In the scope of another master of degree thesis work it has been developed a system for passive device detection on board of a bus characterised by the following features:

- minimal set up on board of the bus (sensor and router).
- web portal for accessing and consulting raw data.
- REST API.

The volume of data produced by this system daily is very high, meaning that after some time of its deployment the manual consultation of the web portal became time-consuming for any operator trying to extract statistics.

2.2.1 Starting point

The system of passive device detection described before its the base where this thesis work started. The existing system has been first observed and analysed in

order to understand its behaviour and from it build initially a design that uses its functionalities to develop an automated way to do three operations:

- Calculate the number of devices on board.
- Calculate the bus lines served by the bus.
- Combing the two results together to obtain the wanted metric.

A second phase of the design brought also a the conception of design for a graphical representation of the metric which will be described in further chapters.

Chapter 3

Architecture

In this chapter, it will be introduced the architecture of the software system that has been developed.
In the first sections, it will be discussed in detail the architecture of the system that has been extended with this thesis work, also going into the details of the data that this system produces.
The last sections of the chapter will talk about the system requirements, functional description and software architecture of the developed system.

3.1 Existing system architecture

In this section, it will be described the key components, from the perspective of the developed software, of the already existing system of passive device detection.

3.1.1 Overview

The existing system focuses mainly on how to collect and transmit the information about detected devices on board a bus, putting the greatest effort in the hardware implementation. The solution implemented by this system relies on Wi-Fi technology to detect devices on board of buses using the approach of monitoring and grouping together what is known as "probe request". A probe request is a mechanism that Wi-Fi technology uses to implement the possibility of discovery, that is to make a device know which Wi-Fi's AP [1] are present in the area. So a device that wants to discover the AP present nearby will transmit probe requests encapsulated in Wi-Fi frames[2], however, in doing that is possible for a device that is monitoring the traffic of Wi-Fi frames nearby to identify each device uniquely.

[1] Access Point.

[2] Message format for the Wi-Fi protocol.

3.1.1.1 Devices' sensor

The first component of the system is a microcontroller installed on the bus which has the capability of monitoring the Wi-Fi frames transmitted nearby to identify uniquely each device that is sending a probe request. In the scope of this thesis, the microcontroller, from now on, will be defined as a devices' sensor.
The devices' sensor can track devices around itself, but still misses the capability of associating each device to the bus position and transmitting the results of the computation to a remote processing server. In order to provide this functionality, a second hardware component has been added, which is a router.

3.1.1.2 Router

The router provides two essential services to the devices' sensor:

- Internet connectivity provided through the use of a 4G module.
- GPS position of the bus provided using a specific module installed on the router.

The router provides access to this services creating a wireless LAN[3] where the devices' sensor connects to.
With the services provided by the router, the devices' sensor has enough information to describe a device's detection, and in that case, it is ready to transmit it to the remote server. The remote transmission is handled using a protocol called MQTT.

3.1.1.3 MQTT

MQTT is a solution based on TCP/IP protocol used typically for M2M[4] in IoT contexts and uses a publish/subscribe pattern where there is a central entity called broker which handles the reception of messages coming from the entities known as publishers and forward this message to other entities called subscribers. Each message is associated to a label called topic which identifies the recipients, which are called subscribers, at which the message has to be forwarded to. Figure 3.1 shows a schema of the protocol.

[3]Local area network.

[4]Machine to Machine communication.

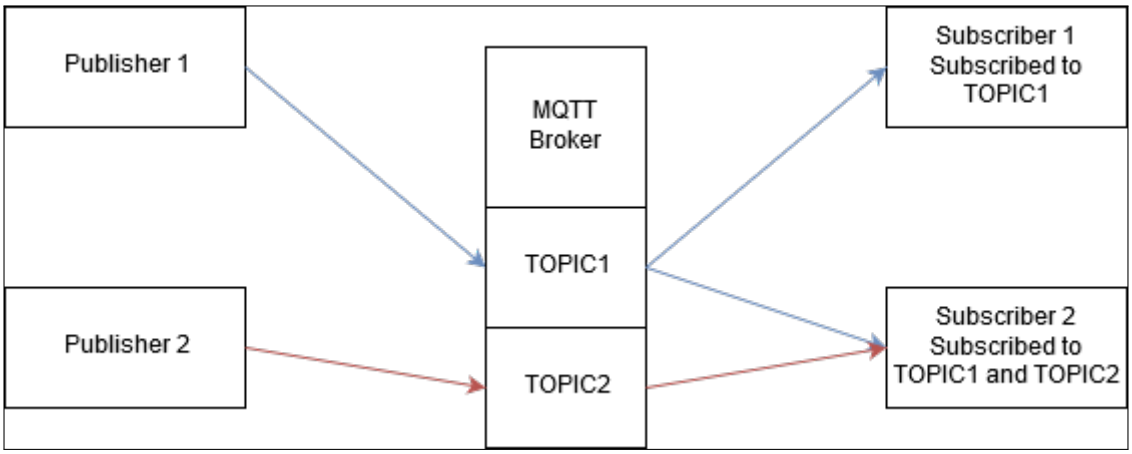


Figure 3.1: Functional schema of the MQTT protocol.

3.1.1.4 Remote server

The device's detections are transmitted to an MQTT broker under a specific topic. The remote server, which is the last key component of the existing system, runs an applicative which uses an MQTT client to subscribe to the topic where the device's sensors send data. Every new device's detection published on the topic is acquired and stored in a DB[5] engine.
REST web services are used to retrieve the data, stored on the DB, about positions and devices. Accessing the data stored on the bus requires specifying a key code in the HTTP request. This code that is called either "IMEI" or "router ID" and identifies the data coming from the devices' sensor installed on a bus.

3.1.1.5 Deploying the system

To deploy the system on top a bus is necessary to install the devices' sensor and the router hardware components on board the bus and configure them.

3.1.2 Endpoint data model devices

This section will detail the data entities exposed by the existing system including the description and the path of the web resources. The system exposes more REST endpoints than the one described in this section, but the others are not relevant for this thesis work.

[5]Database.

3.1.2.1 Devices and Positions

The REST endpoints exposed by the existing system are:

Positions (/collect_data/positions) Is an endpoint exposing a set of entities named "Position". The coordinates and speed of the bus at a certain point of time are the attributes of a "Position". In the next figure, it is reported the class diagram. 3.2.

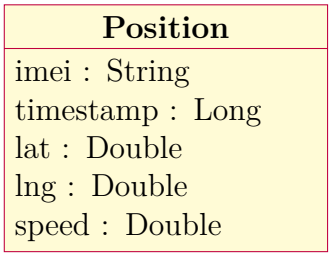


Figure 3.2: Class diagram representing the coordinate and speed of the bus at a certain point of time.

Grouped devices (/collect_data/devices) Is an endpoint exposing a set of entities named "Device". The attributes of a "Device" are the set of information that can be extracted from the Wi-Fi's probe request. In figure 3.3 the class diagram.

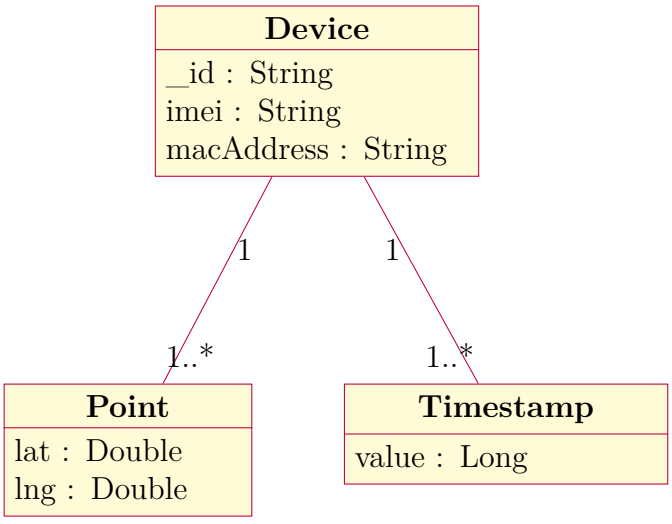


Figure 3.3: Class diagram representing a device detected by a sensor having identifier equal to "IMEI"

3.1.2.2 MyMoby client

Before the development of the thesis work, another development team implemented a basic client of the REST's API described in the previous section. This client allows having a visual representation of the path the bus follows during the day and the positions over time of each device detected on a day.
The UI[6] presents three main sections:

1. **Parameters** This section of the UI allows to select data applying some filters based on the following parameters:

   - Date of detection.
   - Time of day, specified through a select box and a time bar.
   - "IMEI" which as said before identifies the data collected on a bus.

   The time bar does not serve only the purpose of specifying a time of the day to select, but also is used as an indication of time. Also there is an informative counter which tells the total number of detections saved during the selected date and also how many of the them are meaningful[7].

2. **Device list** It is a list of identifiers of all devices which have been detected in the date selected in the parameters section. Clicking on a identifier on the list allows to show its data on the map.

3. **Map** An interactive map which shows two kind of information:

   - The position of the bus at the time of the day selected with the parameters section.
   - Circles positioned at coordinates where a device has been detected by a sensor.

The graphical appearance of each component is shown in figures 3.4, 3.5 and 3.6.

The map section allows different zoom levels allowing to see precisely the roads followed by a device and also the bus stops intersected by the device. Examples in figures 3.7 and 3.8.

[6] User Interface.

[7] A device is meaningful if the are at least more than two detections of it.

Figure 3.4: Parameters section of MyMoby UI.

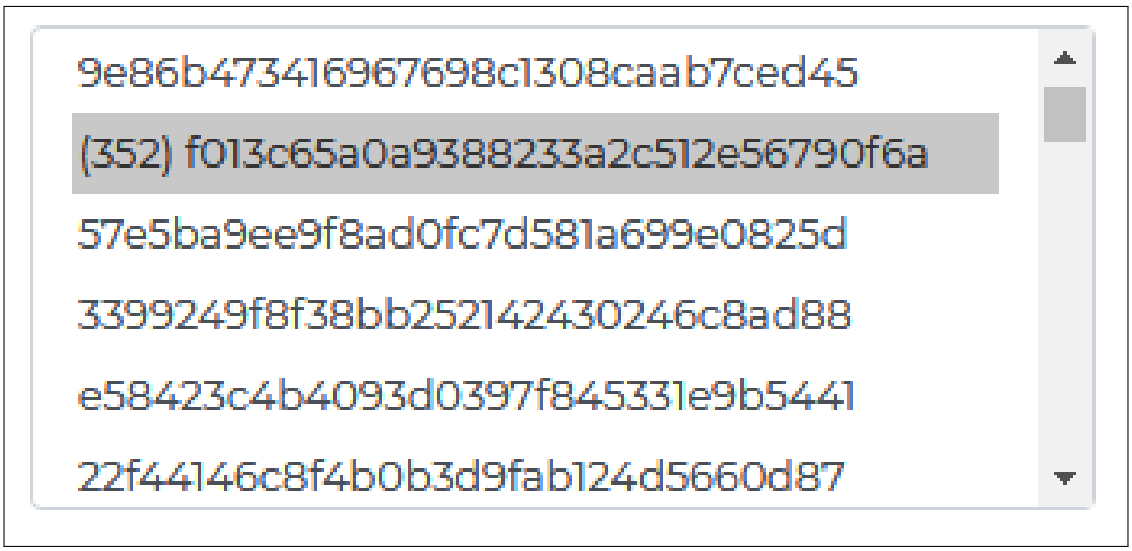


Figure 3.5: Device list section of MyMoby UI, by clicking on a device ID is possible to show it on the map.

3.2 Objectives and requirements for the software extension

The existing software accomplish data collection and storing, but it has no functionality of data analysis, leaving this job to an operator that needs to use the MyMoby portal.
Several useful metrics can be extrapolated from the analysis of data coming from the sensors, and the project associated to this thesis work aimed at extracting the statistic about the line utilisation for each line served by the buses where the sensors are installed. The statistic objective of the thesis to be calculated requires to know for each time instant how many devices are on board the bus and which bus line the bus is serving. Those two kinds of information are deducted from the positions and devices' detection.
Observing the devices' data set it can be observed the presence of detections which are not belonging to a human user or are relevant to the analysis. This phenomenon is correlated to the fact that devices' sensors installed on board the bus are capable

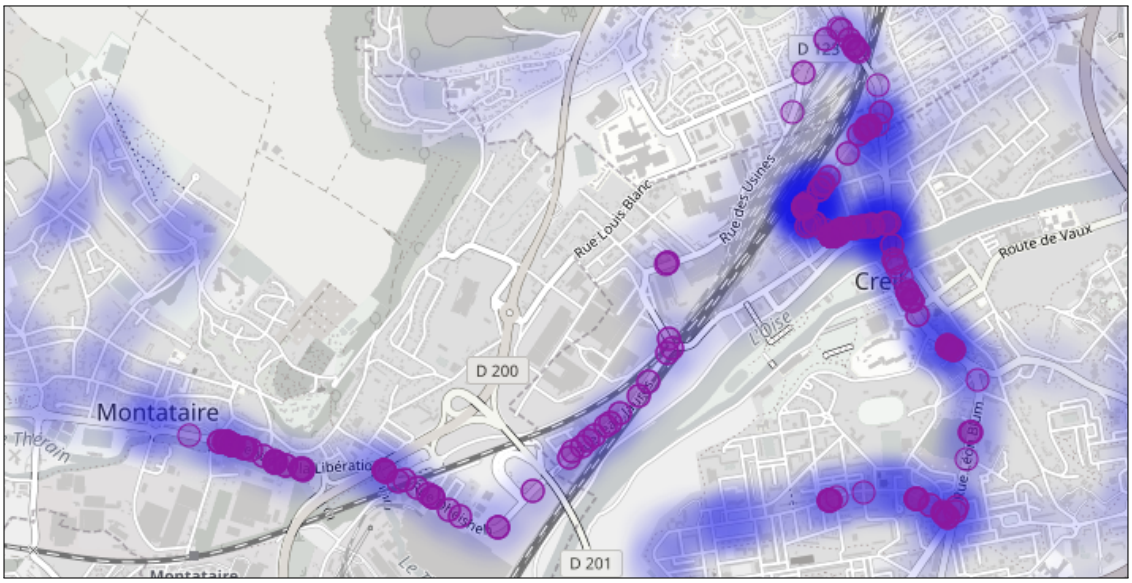


Figure 3.6: Positions of a device shown on the map section of the UI.

of sensing devices that are outside of the bus like people on the road or IP cameras. Those types of devices are characterised by having a short detection period for the case of people outside the bus or to be detected in fixed locations for the case of IP cameras.
In both cases, those devices are not representative of the number of people on the bus and must be ignored.
Another objective followed in the project is the need of showing the result in some intuitive way. So it has been decided to build a graphic representation of the metric that shows for each line the sequence of its bus stops connected by a line with width proportional to the number of devices detected between each pair of bus stops.

3.2.1 Requirements of the software extension

After the choice of the statistic and the data set observation a list of requirements has been deducted:

R1 Data filtering: The system must ignore the devices which are not representative of the devices on board of a bus.

R2 Path classification: The system must know for each time instant which is the most probable bus line which the bus was following.

R3 Data aggregation: The system must know for each time instant the count of devices on top of the bus.

Figure 3.7: Picture showing on of the roads that the device passed by.

R4 Readability: The system should be able to show a human legible visualization of the metric.

3.3 Software architecture

This section will present each software component's functional description, data model and its interaction in the architecture. In figure 3.9 it is shown the high level view of the software components.

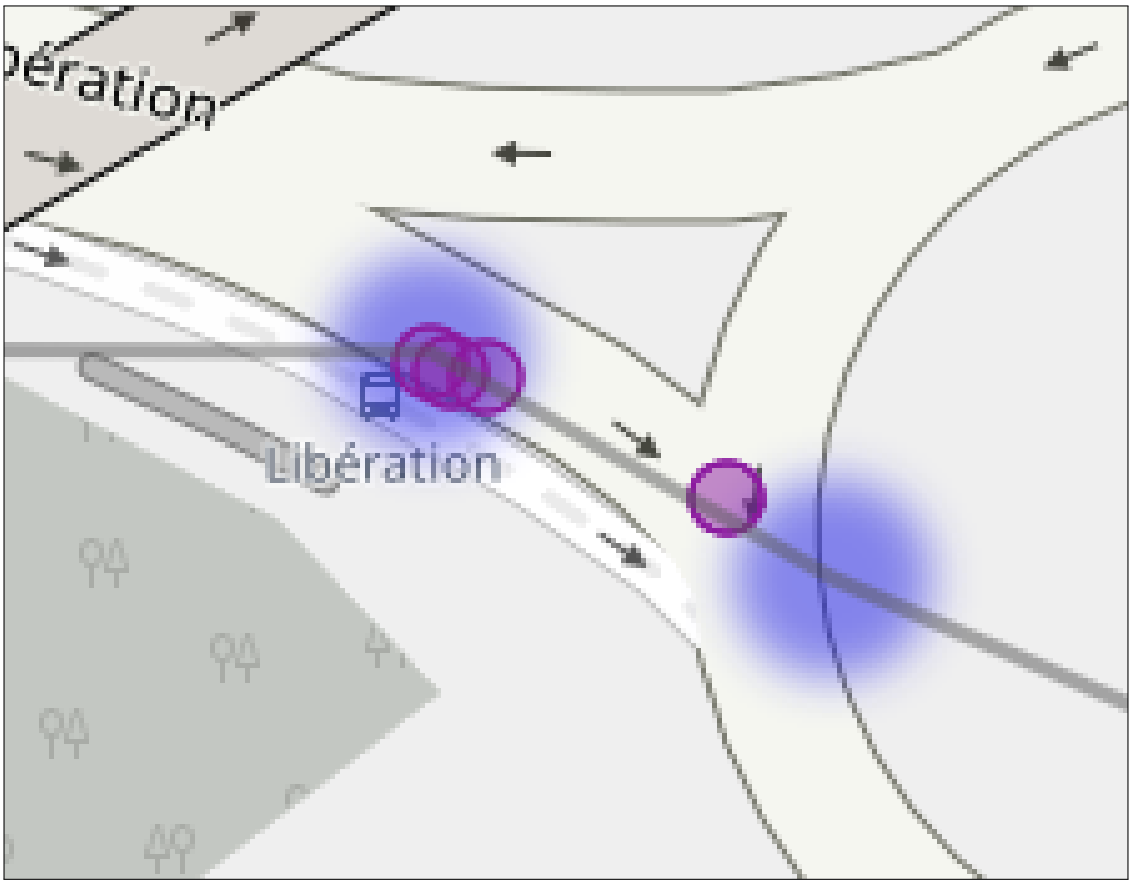


Figure 3.8: One of the bus stop this devices passed by.

3.3.1 Functional components

The system is divided into two independent software components, one responsible for data analysis and the other one responsible for building graphic representation out of the result of the analysis.
The result of the data analysis is a data structure called "StopsTable" that represent the value over time of the statistic objective of this thesis work.
Every day the first component produces new values for this data structure while the second component can select a subset of this data structure's values to build a graphic representation of the metric. In figure 3.10 it is reported the class diagram of the "StopsTable".
The following section talks in detail of the two components which are called, respectively in order of introduction, the Classifier and the Data Visualizer.

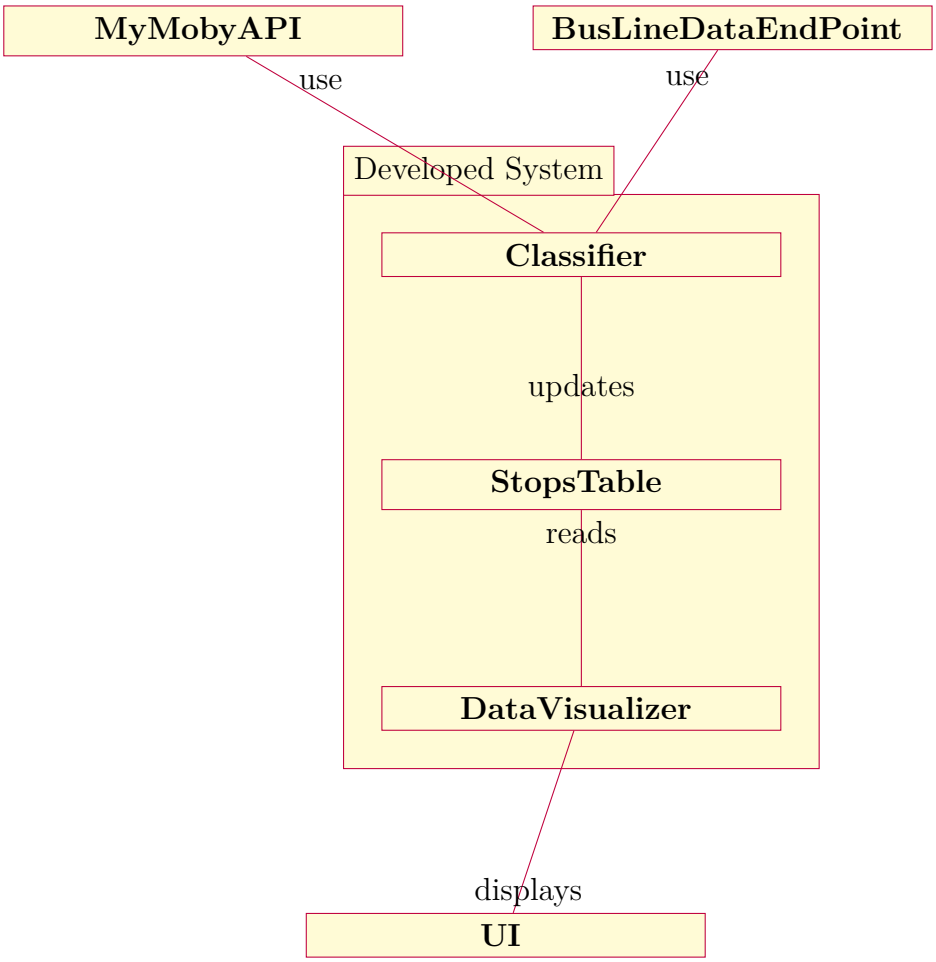


Figure 3.9: Software architecture representation.

3.3.2 Functional component: Classifier

The Classifier component has the role of computing daily the metric, objective of the thesis, using the data of the day before coming from the existing system's endpoints and save the result of the computation inside a "StopTable".
Two main algorithms are executed to obtain the line utilization and these algorithms are:

- Device analysis.
- Itinerary matching.

Those tasks process independently the data coming from the two endpoints and once they are finished the result of the operations are joined together.
The next sections will described each operation in details.

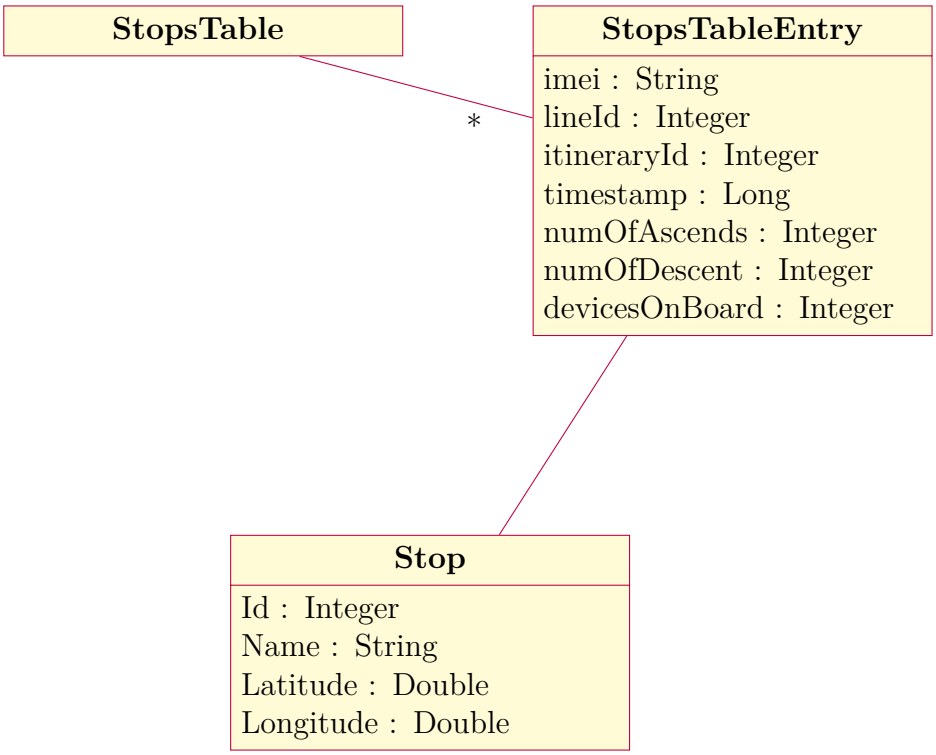


Figure 3.10: Class diagram of the "StopTable".

3.3.2.1 Device analysis

The device analysis works on the device endpoint to estimate for each device belonging to a person at what time they entered the bus and at which time they left it. This operation is divided into two sub-operations; one dealing with finding invalid devices, the ones who do not belong to a person, and the second sub-operation determines the ascent and descent time.

3.3.2.2 Device filtering

The devices' data set presents common patterns which are listed here:

- **User device on board** device detected in several positions along the bus path for a interval of time below 2 hours.
- **User device not on board** device detection with low detection duration.
- **Cameras** device's detections heard in fixed points.
- **Bus driver** device detection with a duration longer than 2 hours (2, 3 per day).

- **Bus maintenance** device detection heard before 6 a.m. and before the bus has even accelerated indicating that these detections are made at the bus deposit.

Among this patterns only the first one is relevant for analysis purposes, all other types of device detection must be filtered from the set of devices to analyse.
This software component applies a rule-based classification for each device in the devices list and removes from the list all the devices which are not classified as user device on board. The rules on which the classification are made are:

- The number of device detections is below 5 -> User device not on board
- The positions of a device are concentrated around 1 point -> Camera
- The time period on board of the bus is more than 2 hours -> Bus driver
- The bus has not accelerated yet -> Bus maintenance

3.3.2.3 Ascent descent analysis

The devices not filtered by the filter, are analysed one by one to determine the ascent and descent times for each device. The calculation is made considering the set of time instants that the device has been detected, forming groups of nearby time instants and considering, for each group, as ascent time the first time instant of the group and as descent time the last time instant. An ascent (or descent), from now on, will be now defined as "Variation" which is a pair of values where the first value is the instant when the ascent (or descent) happened in time and the second value is either 1 or -1, and it is used to distinguish, respectively, an ascent or a descent.

3.3.2.4 Itinerary Matcher

The bus during the day may follow different paths meaning it serves more than one line at day. The Itinerary Matcher algorithm has the role of determining, throughout the day examined, the set of bus lines followed by the bus and the bus stops where the bus stopped. A probabilistic approach is used to determine the bus lines served by the bus, and it consists in determining for each possible itinerary, a function of probability. Comparing the calculated functions of probability, it is possible through the comparison of them to know which line the bus was following. The probability functions are calculated, to simplify the study, in a time window where the bus is likely to be serving only one itinerary of a bus line. This time windows are called "Segments". Before going into the functional description of the algorithm, it will be introduced the model of bus line data and some abstract entities used in the itinerary matching.

3.3.2.5 Bus line data definition

The transport society created an abstract model able to represent a line. The main concept in this model is the notion of "Itinerary" that represents a path composed by a set of "Geometries" which describe the path the bus follow departing from a bus stop and arriving at the next one. A bus line can follow several itineraries.
In figure 3.11 the class diagram.

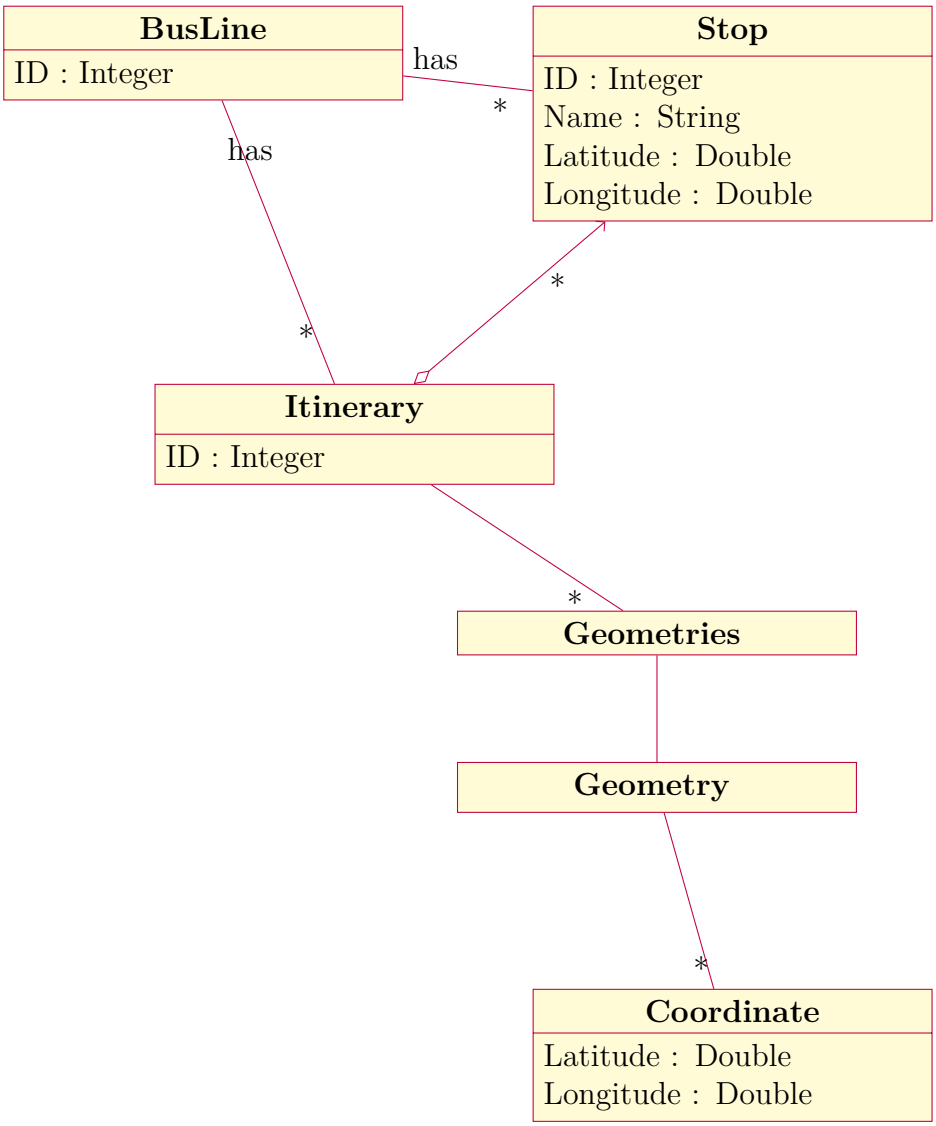


Figure 3.11: Class diagram representing the bus line data.

The transport society exposes data of bus paths for each line from the following endpoint:

- **api/map/GetLineMap** is the name of the endpoint that takes as a parameter a line identifier and returns the bus line data for that line.

The model of the data returned associates to a line identifier the following information:

- **Itineraries** set of paths the bus can follow. Each path carries information about the set of stop ids that the path includes and the set of coordinates that the bus follows from the first stop to the last stop.
- **Stops** set of all bus stops information for this line.

3.3.2.6 Itinerary matcher glossary

In this paragraph it will be explained and characterized the entities used in the itinerary matching algorithm:

BusPath From the position's data set through reordering it is possible to build an entity representing the set of positions the bus followed from the moment it first accelerated (speed > 5) to the last time instant of the day.
In figure 3.12 the class diagram.

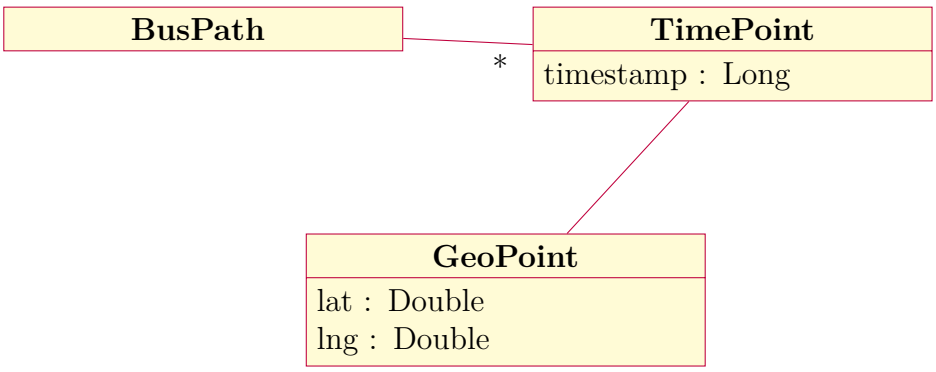


Figure 3.12: Class diagram of a "BusPath".

Segment A "Segment" represents a time window composed by a pair of time instants and a set of "TimePoint" having their time-stamp included in the interval defined by the pair of time instants.
In figure 3.13 the class diagram.

SegmentClassification Each "Segment" goes through a process of classification which ends in assign a label to the "Segment". The label assigned is a line identifier, meaning that in the time window defined by the "Segment" the bus was serving a specific itinerary of a bus line. Some information useful to process next segments are contained inside the Segment Classification entity.
In figure 3.14 the class diagram.

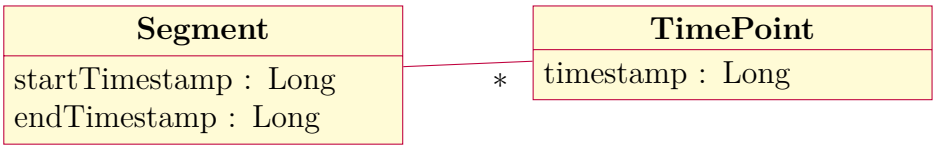


Figure 3.13: Class diagram of a "Segment".

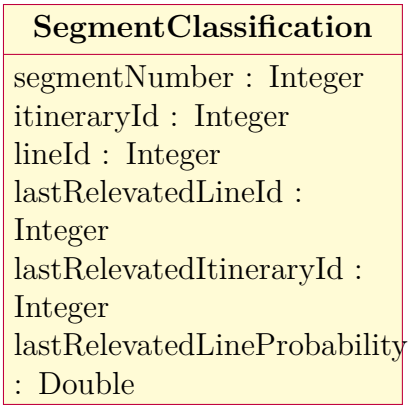


Figure 3.14: Class diagram of SegmentClassification.

LineProbability is a class that indicates for a specific time instant which is the probability that the bus is following an itinerary.
In figure 3.15 the class diagram.

3.3.2.7 Mathematical model

The itinerary matching is based on a mathematical model which allows calculating the probability associated to the fact the bus is following a specific itinerary.
Assuming there are "m" possible itineraries that the bus can follow, to find which is the most probable itinerary the bus followed in the segment it is required first to build a function of probability P. The latter describes what is the probability that the bus is following a specific itinerary for each time instant belonging to a segment. The function is computed recursively comparing the positions the bus occupied in a specific "Segment" and the points of all possible itineraries the bus can follow. The P function also keeps into consideration the direction the bus is going and if the bus is following the sequence of bus stops of an itinerary.
Figure 3.16 shows the mathematical model. The idea behind this mathematical model is that the P function at each time instant is made up of two weighted contributes[8]. The first contribute comes from the previous time instant's function

[8]The "a" constant defines the weight.

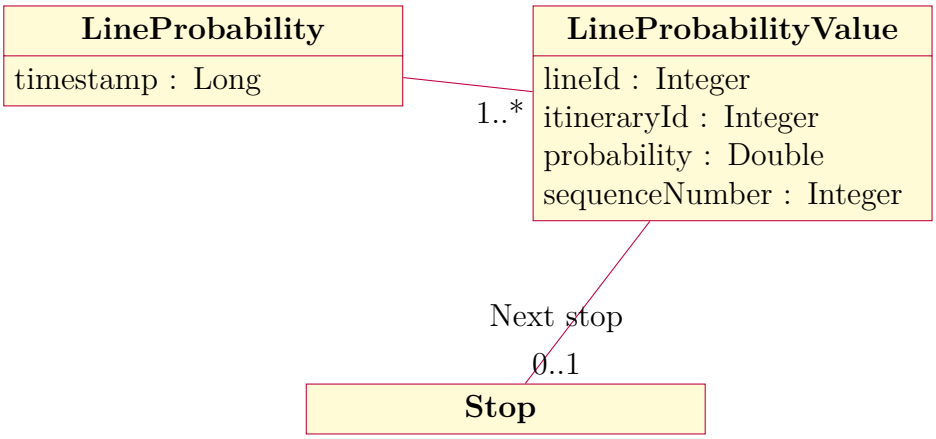


Figure 3.15: Class diagram of a "LineProbability".

value and the other comes from a function depending on the distance between the current point of the bus and the closest point to it of each itinerary.

3.3.2.8 D and I functions

To explain "D" and "I" functions, consider the bus position C(k), N_i set of points of the i^{th} itinerary, which can be restricted in cardinality by the I(k) function. Then the D function finds for every itinerary the minimum distance between the bus position C(k) and any point of N_i if this distance is lower than 50.0 meters. The index of the point of the itinerary at minimum distance to the current time point is the value of the "I" function at instant k. The "I" function indicates that all indexes, of the points of the itinerary, which are lesser than the current value of the function are irrelevant because if a bus is following the itinerary then it does not make sense to match the position of the bus to points of an itinerary which have already been travelled.

3.3.2.9 Stop sequence matching

The value of the P function, at instant k, for each itinerary, can have another multiplicative term if the bus is nearby a bus stop. The bus being nearby a bus stop alters the probability in the following ways:

- if the nearby stop is the first bus stop encountered in the scope of the segment, then no alteration is made to the value at instant k of the probability, but the value at time k + 1 may be altered.
- otherwise considering the last bus stop encountered as A:
  - if the nearby bus stop is the same or the next of A then that means that the bus is behaving as expected on that itinerary.

$$a \in [0,1], \tag{3.1}$$
$$k \in \text{set of time instant of a Segment}, \tag{3.2}$$
$$m = ||Itineraries||, \tag{3.3}$$
$$N_i = \text{ordered set of points of bus path of line i}, \tag{3.4}$$
$$n = ||setoftimeinstantsofaSegment||, \tag{3.5}$$
$$C\colon \mathbb{N} \to \mathbb{R}^2 \tag{3.6}$$
$$P\colon \mathbb{N} \to \mathbb{R}^m, \tag{3.7}$$
$$D\colon \mathbb{N} \to \mathbb{R}^m, \tag{3.8}$$
$$I\colon \mathbb{N} \to \mathbb{N}^m, \tag{3.9}$$
$$E\colon \mathbb{N} \to \mathbb{R}^m, \tag{3.10}$$
$$P_i(k) = \begin{cases} p_0 & k = 0 \\ P_i(k-1) & k > 0, D_i(k) \neq \emptyset \\ a \times P_i(k-1) + (1-a) \times \frac{1000}{999+E_i(k)} & k > 0, D_i(k) \in \mathbb{R} \end{cases} \tag{3.11}$$
$$E_i(k) = \exp \frac{D_i(k)}{3}, \tag{3.12}$$
$$D_i(k) = \begin{cases} min_{j=I_i(k)..n}\{||C(k) - N_{i,j}||\} & min_{j=I_i(k)..n}\{||C(k) - N_{i,j}||\} < 50.0 \\ \emptyset & \text{otherwise} \end{cases} \tag{3.13}$$
$$I_i(k) = \begin{cases} 0 & P_i(k-1) \leq \epsilon \\ b & P_i(k-1) > \epsilon, b \in \mathbb{N} \end{cases} \tag{3.14}$$
$$\tag{3.15}$$

Figure 3.16: Mathematical model for the P function, indicating the probability that the bus is serving a specific itinerary.

- if the nearby bus stop comes before of A in the sequence of bus stops, then the probability goes back to 0 for instant k.
- if the nearby bus stop comes more than a stop after A or A + 1 then the probability is multiplied by this factor:

$$\frac{P_i(k)}{0.588 * O} \tag{3.16}$$

  Where O is the indexes' distance between the index of bus stop A and the index of the nearby bus stop.

3.3.2.10 Internal and external aliasing

The functions of probability for each itinerary are not enough to choose which itinerary the bus followed in the "Segment", because of two phenomena called internal and external aliasing which are defined as follows:

Internal aliasing it occurs when two or more itineraries associated to the same line have an high probability.

External alasing it occurs when two or more different bus lines have itineraries which have an high probability.

So in order to choose the right itinerary another step is required, which will be discussed in the next section.

3.3.2.11 Choosing the itinerary

The selection of an itinerary as the most probable in a segment has to be done considering 2 possible cases:

1. it exist a time instant K where only one itinerary has an high probability.
2. internal aliasing and/or external aliasing.

In the first case, the solution is simple, and it consists in choosing the only itinerary with high probability as a result of the classification process.
The second case is more complicated than the first one because it requires the mitigation of the effects of aliasing and the solution for it is based on calculating a frequency function. This function, called F, can be computed from the P function in the following way, considering "m" the number of possible itineraries:

$$F_i(k) = \begin{cases} 1 & P_i(k) > P_j(k), j \neq i, j = 1..m, \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

This function indicates for each instant and itinerary if the value of probability is the greatest among all components of the P function. It can be used to mitigate both internal and external aliasing by performing two operations:

1. sum all values of the function for each itinerary.
2. from the result of the previous operation sum again this time grouping itineraries belonging to the same line.

The highest sum indicates which line is the most probable in the segment and this solves external aliasing. For internal aliasing is necessary to select one of the itineraries of the line selected as segment classification. The itinerary selected in

this case will be the one with the highest number of stops. This choice is guided by the fact that a line has several itineraries, as explained before, but some of them include the others. So typically the itinerary having the highest number of stops is the one including all the others for the same direction.
The line and itinerary identifier together are the result of the classification which is called "SegmentClassification", and it will serve not only for the second phase of the itinerary matching but also to provide initial values of probability for the next segment to classify.

3.3.2.12 Stop sequence matching

Once all segments have been classified the second phase of the itinerary matching consist into building the sequence of bus stops that the bus followed during the day. The sequence can be built reiterating over the segments, but this time knowing the line and the itinerary the bus was following. Each position that the bus occupied in a segment is matched against the set of bus stops of the most probable itinerary of the segment and for each position near a bus stop an empty entry in the "StopsTable" is created. In case the entry in the "StopsTable" already exists it will not be created a duplicate.
After having reiterated over all segments and built the bus stops sequence using the knowledge of the segment classification the built "StopsTable" indicates only the time at which the bus arrived at each bus stop, but still misses the data about the devices detected on board which will be obtained through the process of joining the results of the two main algorithms.

3.3.2.13 Joining the data

The result of the ascent and descent analysis produces a list of time instants associated with a number which when positive indicates that in the particular moment there was an ascent on the bus otherwise if negative it indicates a descent. This list allows to complete the information that the "StopsTable" misses, that is the number of devices on board.
Every ascent (or descent) time is included between the time instants of two entries of the "StopsTable". Each ascent is assigned to the first entry which has as the time instant's value lower than the one of the ascent; similarly, each descent is associated to the first entry having the time instant's value greater than the one of the descent. To know the number of devices of a given entry of the table is sufficient to calculate the sum of all increments of the previous entries and add to the sum the increment of the entry of interest.

3.3.3 Functional component: Data visualizer

This software component is responsible for the graphic visualisation of the data processed by the Classifier allowing to see the utilisation of each line.
To fulfil its role the component builds a graph representing the metric for an itinerary of a bus line. Each of the graph's nodes represents a bus stop, and the weight of each edge represents the average number of devices for every pair of bus stops. In figure 3.17 there is a conceptual representation of the graph.

Figure 3.17: Conceptual representation of the metric's graph.

The component can build the graph in two ways; The first representation consists in a graph where each node represents a bus stop, and the width of the line joining each pair of adjacent nodes indicates the average number of devices present on the bus between the two stops.
The second representation is a projection of the first representation on a map where each node is positioned in the coordinates of the bus stop which is associated to it.

3.3.3.1 Data selection

To build this graph for a bus line is necessary to select and aggregate data coming from the stops table. The aggregation is necessary because the table's entries for the same line can come from different sensors. So the first step in the data selection is to choose a time interval of interest and a time of the day and filter out the entries not included in the time window.
The second step consists in grouping the remaining entries by itinerary and stop identifiers and calculate the average number of devices for each group. The following tables show an example of how the filtering and grouping work assuming the period of interest is the first week of September and the hour of the day between 8 and 10 am.

It is possible to observe that for the bus stop called "Desnosse" the average is computed not counting the first entry because belonging to a time period not of interest.
In figure 3.18 a class diagram shows the parameters on which the metric can be calculated and which information, regarding the bus stops, is associated to the parameters specified.

Itinerary ID	Stop ID	Stop name	Date	Hour	Sensor id	# devices on board
20	52	Desnosse	29/08/2018	9	4752	6
20	52	Desnosse	04/09/2018	8	4752	10
20	52	Desnosse	05/09/2018	8	4753	12
20	53	Jules	05/09/2018	8	4752	15

Table 3.1: Content of the stop table before grouping.

Itinerary ID	Stop ID	Stop name	Average devices on board
20	52	Desnosse	11
20	53	Jules	15

Table 3.2: Grouped content ready to be shown.

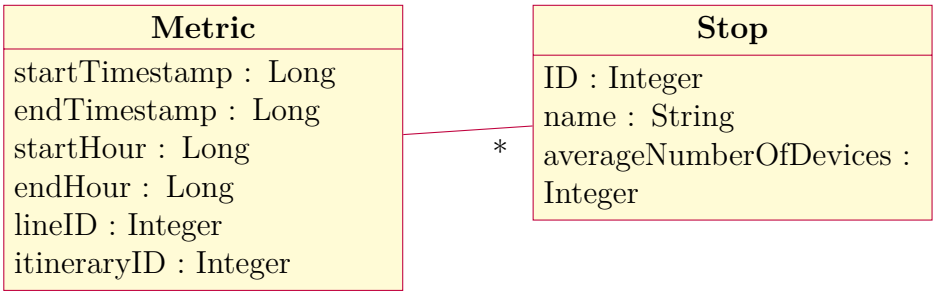


Figure 3.18: Class diagram of metric in a period of time for a specific itinerary of a bus line.

Chapter 4

Implementation

In this chapter the implementation details of the system covering technical details and all the choices made during the development process will be discussed.

4.1 Environment choice

This section will describe the choices made for the environment which are the choice of the language, the structure of components and the DB engine.

4.1.1 Existing system solution

The existing software architecture deployed on the remote server utilises the Spring Framework which is an open-source framework for developing secure Java EE applications. In particular, Spring is used to provide the APIs exposing the data of the devices and positions. These endpoints access directly the persistence layer of the server which is implemented using a NoSQL database which is MongoDB.

4.1.2 Extension solution

Since the developed software extension has to run on the same server in favour of future integration of the developed system in the old one and to avoid the need of implementing an API to access the services of the extension it has been chosen to develop the extension in Java programming language.
The software components described in chapter 3 have been structured in different ways.

4.1.2.1 Classifier structure

The Classifier component has been developed in the form of a Java standalone application that runs a TimerTask which executes every 24 hours. This choice has been made for several reasons which are:

- This component does not expose any UI.
- To apply the itinerary matching algorithm only a full day of data provides enough information to rebuild the sequence of bus lines and stops followed by the bus during the day.
- Not having access to the MQTT broker inhibits all real time processes.

4.1.2.2 Data visualizer structure

The data visualizer component has been developed in form of a Web Application. This application is made with two main layers which are the data access layer and the UI. The data access layer provides to the UI services to query the data in a parametrized way. Since it is not possible to access directly the database chosen for this software extension the data access layer provides classes that handle the following responsibilities:

- Receive requests from the UI extracting the parameters to query the db.
- Build queries.
- Forward query to the db engine and pack the result to the UI.

The UI is structured as a one-page application, and it relies on JavaScript with JQuery, Leaflet and D3 library to interact with the data access layer and build the graphic representations.

4.1.2.3 Database engine

The choice of the database engine was less constrained from the existing system and was driven mainly by two requirements.

- **Support for geographical queries** The itinerary matching algorithm requires this capability for computing the D function.
- **Performance** The existing system daily output is very high so the itinerary matching has to compute the results within 24 hours.

Among the possible choices there were two main solutions evaluated.
The two evaluated solutions were:

MongoDB NoSQL database engine, document oriented, which provides excellent performance for simple queries and it supports some functions in the scope of geographical queries.

PostgreSQL's PostGIS extention PostgreSQL is a relation database engine which has an extension called PostGIS that supports a great number of geographical functions.

The final choice went over MongoDB for two main reasons:

- The performance in spatial queries of MongoDB is superior of a factor of 10 than PostGIS.
- MongoDB is also the database engine of the existing system, so in case of future integration of the extension inside the existing system this choice is more flexible for future developments.

The database engine that has been chosen serves has a double purpose which is to store the results of the computation that is the stops table and be used as an engine of computation when geographical querying capability is needed.

4.2 Classifier implementation

In this section it will be described the Classifier component which among the two components has the role of analysing the data coming from existing system end-points and add new values to the "StopsTable" persisted on the db.

4.2.1 Main loop

The Classifier component is implemented in the form of Java Timer Task executed daily. The run method contains a for loop which performs a series of operations for each couple router and devices' sensor. As stated before, each bus is identified by a unique code[1] and in order to be analysed is necessary that this code is inserted in a specific collection inside the DB.
The tasks performed in the main loop are the following

- acquiring data.
- preparing it for processing (build bus path and divide it in segments).
- apply ascent and descent analysis.
- apply itinerary matching algorithm.

[1]Router's identifier of the router installed on the bus.

- join results.
- export data.

The following section will talk about the implementation of each operation.

4.2.2 Acquiring the data

The first process made from the Classifier component is the data acquisition.
The component has to acquire the data of all devices' sensors and the bus line's paths' data. This kind of data is available at two different data endpoints in JSON[2] format and it can be accessed through HTTP[3] GET requests.
Based on this, the component requires the capability of making HTTP requests.

4.2.2.1 Possible choices

In this section the possible solutions will be listed and analysed, some of them being integrated with the core packages of Java and other being libraries.

HttpUrlConnection The HttpUrlConnection class allows performing basic HTTP requests without the use of any additional libraries. All the classes that are needed are contained in the java.net package. The disadvantages of using this method are that the code can be more cumbersome than other HTTP libraries, and it does not provide more advanced functionalities such as dedicated methods for adding headers, authentication, or serialisation mechanisms.

Retrofit Retrofit is a type-safe REST client for Android, Java and Kotlin developed by Square. The library provides a robust framework for authenticating and interacting with APIs.
To use this framework is necessary first to define a Java Interface which resembles the structure of the API. For every resource's path it must be specified the following information:

- resource path.
- return type.
- parameters, that can be passed through the url or as query parameters.

In figure 4.2 there is a simple example of how the mapping is performed.
Once the annotated Java Interface is defined it is possible to obtain a client where each of its methods when called trigger an HTTP request.

[2]JavaScript Object Notation

[3]HyperText Transfer Protocol

```
URL url = new URL("http://example.com");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("GET");
BufferedReader in = new BufferedReader(new
    InputStreamReader(con.getInputStream()));
String inputLine;
StringBuffer content = new StringBuffer();
while ((inputLine = in.readLine()) != null) {
   content.append(inputLine);
}
in.close();
con.disconnect();
```

Figure 4.1: Http request made using HttpUrlConnection.

```
public interface GitHubService {
   @GET("users/{user}/repos")
   Call<List<Repo>> listRepos(@Path("user") String user);
}
...
Retrofit retrofit = new Retrofit.Builder()
   .baseUrl("https://api.github.com/")
   .build();

GitHubService service = retrofit.create(GitHubService.class);
service.listRepos();
```

Figure 4.2: Retrofit library utilisation example.

4.2.2.2 Final choice

Among the possible choices for the HTTP client, it has been chosen to use Retrofit library because of its flexibility. The context of the thesis is one of an evolving system and Retrofit with its mapping between HTTP resources and Java Interfaces allow easily to redefine the mapping thank to straightforward Java Annotations. For querying the existing system Retrofit alone with interface mapping, it is sufficient, for the bus line data API it has been implemented a class which performs several requests and aggregates them in a single response. The need of aggregating several requests together derives from the fact that the bus line data API can only return information about a single line for each call.

4.2.3 Preprocessing input data

Once the data has been acquired, there is a preprocessing phase which builds the data structure containing the path that a bus followed during the day in exam and then subdivide it into meaningful segments.

4.2.3.1 Building the path

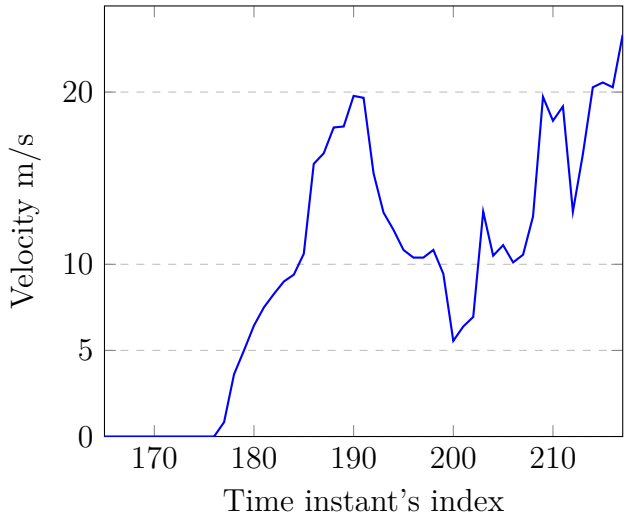
From the positions' endpoint, it is possible to obtain the sequence of coordinates of all the buses registered in the system in a given period. Since the analysis is done considering one devices' sensor at the time. Using the MyMoby website, it is possible to observe that the bus starts the streaming of coordinates around 5'o clock and sends a series of coordinates coming from the same point for a period that can go up to one hour. Using the map on MyMoby the point were the coordinates come from is the same point where the bus goes in its last moments of the day, so logically it has to be the bus deposit. So to build an exact path this coordinates must be excluded from the path. The solution to this problem comes from the observation of the bus's speed over time reported in the plots in figure 4.3. The commonly observed behaviour is that the bus stays at 0 m/s for at least an hour and then after the first acceleration it starts its path. The time's instant when this happens is meaningful because it defines the lower bound of the interval of validity for both itinerary matching and ascent and decent analysis.

4.2.3.2 Bus path segmentation

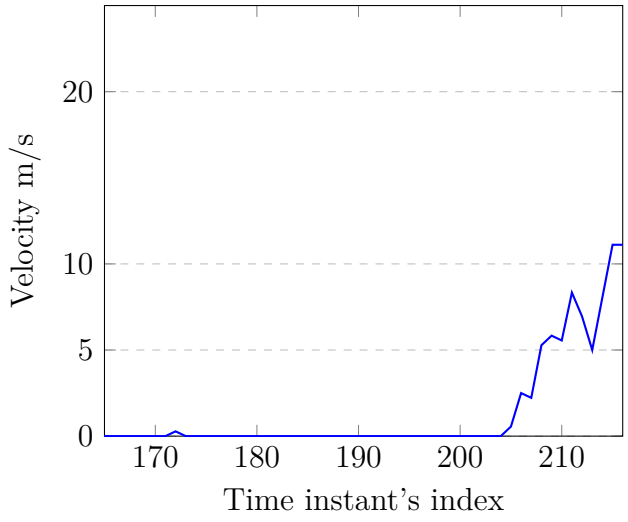
After the bus path is built then is necessary to divide it in segments where probably the bus is following only one itinerary of a bus line. This operation is fundamental to reduce the effects of internal and external aliasing, so multiple solutions where considered:

Divide the path in timeslots This approach is based on estimating the average trip time the bus takes to complete an itinerary and divide the sequence of coordinates into adjacent intervals of values having temporal width equal to the average trip time. The affability of this method depends mostly on making a good estimation of the average trip time.

Consider the variation of velocity overtime The velocity is a good indicator of the behaviour of the bus, so it can also be used to produce segments. The velocity graph considered in an interval where the bus is following an itinerary presents small periods where the bus velocity is equal to 0, it is reasonable to think that these intervals are breaks that the bus takes when it reaches the last stop of an itinerary.
The two main limitations to this method are that the bus can have this behaviour also in situations of high traffic queue or sometimes it may happen



(a) 10/07/2018 from 5:00 to 6:00



(b) 11/07/2018 from 5:00 to 6:00

Figure 4.3: Plots of the speed of the bus in two different days.

that a bus does not do a break between two itineraries. The events described cause in the first case to create too many segments with low width and the second case generates few segments with huge width.
In figure 4.4 an example that has been verified using the MyMoby portal.

The final choice went in favour of the first approach because even the speed's analysis yields better result only when the limitations stated above are not happening. Instead, the first approach gives good results more consistently.

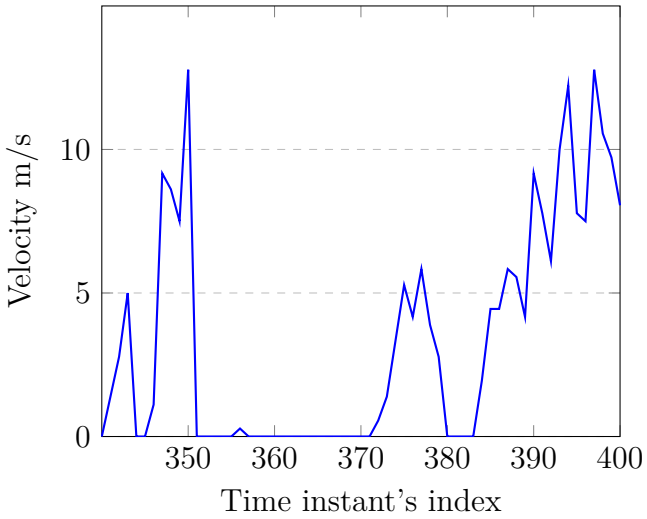


Figure 4.4: Example of a case where the bus is waiting at the last stop before starting a new itinerary.

4.2.4 Device analysis implementation

The device analysis, recalling 3.3.2.1, is divided in two main operations:

- filtering.
- determining ascent and descent time for each valid device.

Those two operations are implemented in the FilterJob class which implements the Callable interface. The Callable interface allows to define objects of which code can be run on a separate thread, but contrary to the Runnable interface[4], the result of the computation can be accessed once the execution of the Callable object is terminated. The execution of any thread trying to access the result of the Callable before the result is available will be blocked until the result is available. The choice of using the callable interfaces comes from the fact that the devices' analysis can be executed in parallel of the itinerary matching. Although the system is predisposed for multi-threading in the current implementation it is not used.

4.2.4.1 Device filtering

The device filtering consists in first classifying the device under one of the categories introduced in 3.3.2.2 and then removing from the device list, obtained through the device endpoint, all the devices not falling under the classification of "User device

[4]Typical method to define a code which can be executed on a separate thread.

on board".

4.2.4.2 Common patterns

To implement the solution the MyMoby portal has been used to observe the devices and highlight the common patterns.
Figures 4.5, 4.6, and 4.7 show the most common patterns for the devices. Observing figure 4.6, it is safe to say that this device belongs to a person due to the variable positions overtime and the fact that is possible, zooming on the map, to pinpoint the ascent stop and the descent stop.
The device shown in 4.7 has the exact opposite behaviour, all the coordinates where it has been detected concentrate in a point and observing the time bar, on MyMoby UI, the detection happens periodically implying that is probably a device not belonging to a human, probably an IP camera. In figure 4.5 it is reported the temporal behaviour of a non human belonging device.

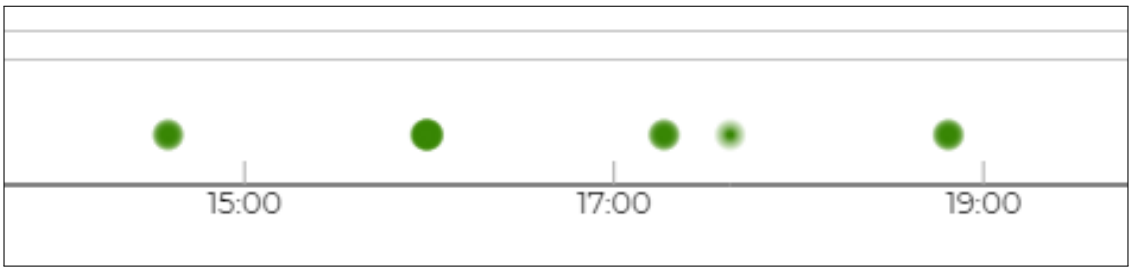


Figure 4.5: Behaviour of a device not belonging to a human.

Another typical pattern is the detection of devices in the first hours of the day while the bus is standing still at the deposit, probably these detections come from people working at the deposit passing by the bus.
The device list comes from the endpoint ordered by ascending time instants, but to filter, it must be reordered to group detections coming from the same device.

4.2.4.3 Grouped devices

Contrary to the abstract representation of the devices shown in 3.3, the devices' endpoint return groups of devices' detections which have been detected in very short periods (< 20s). Among the fields which have not been introduced yet, there is the "counter" field which indicates how many detections the group contains. This field is quite useful because it can be used to determine the total amount of detections for each device, allowing to filter the ones having not enough detections. A threshold value, that can be customized in the configuration, indicates the minimum number

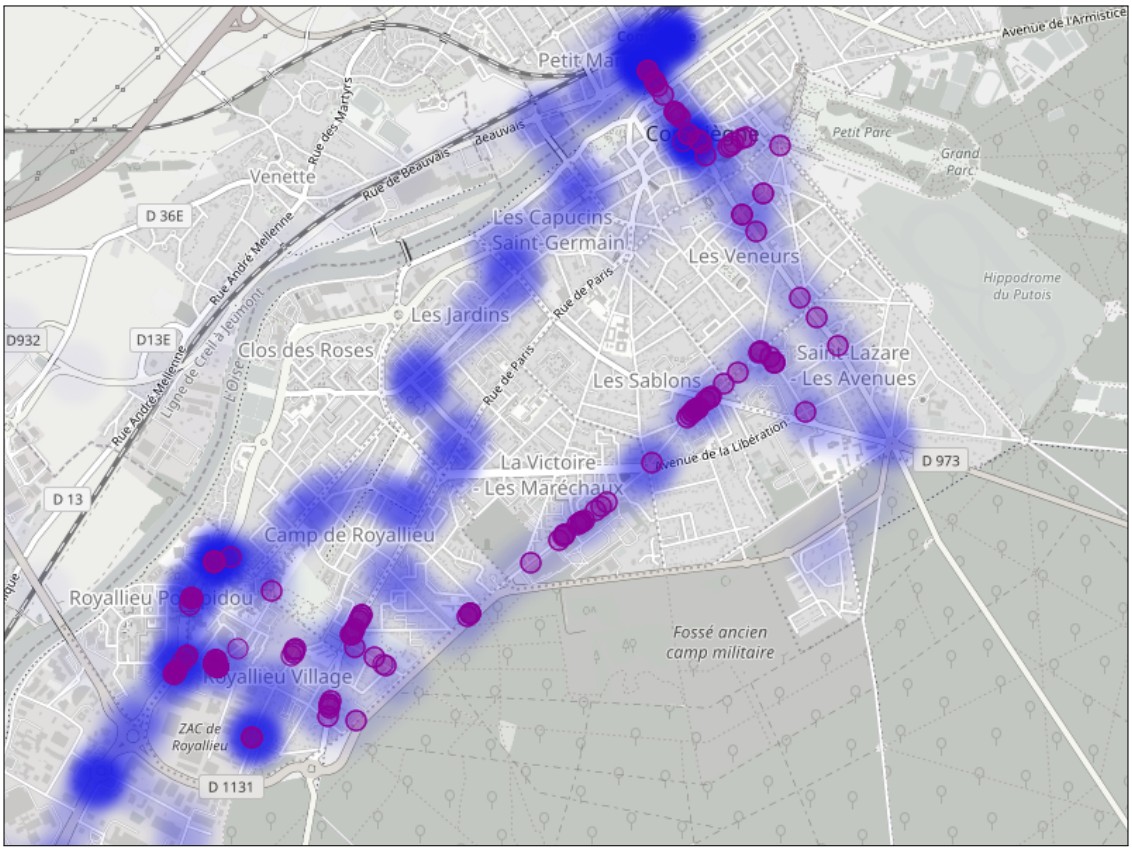


Figure 4.6: Device detection probably coming from an user device.

Figure 4.7: Device detection probably coming from a fixed IP device.

of detection required for each device. In figure 4.8 is reported the structure of a group of device detections.


```
{
   "_id": "5b7aaa459daec200018f68b0",
   "router": "861107035440552",
   "globalMacAddress": null,
   "localMacAddresses": [...],
   "footprints": [...],
   "ssids": [...],
   "rssis": [...],
   "timestamps": [...],
   "points": [...],
   "counter": 46,
   "expired": true,
   "startTime": 1528224555000,
   "endTime": 1528225984000,
   "duration": 1429000
}
```


Figure 4.8: Structure of a list element returned by the devices endpoint.

Based on what explained so far the implemented filtering steps are:

- Filter from the list all grouped devices having as first time instant an instant lesser than the first valid time instant of the built bus path.
- Sort the device list by their MAC[5] address.
- Group together the devices.
- Classify each group, applying the following rules.
  - Get the total number of device's detections, removing groups having a total lesser than the threshold value.
  - Calculate the central point among all devices belonging to a group, removing groups having their points all concentrated within a radius from the centre point. The radius can be customised in the configuration.
  - Calculate the period of stay of the device on the bus, removing the group if it has been on the bus for a time higher than a threshold value defined in the configuration. This value should be set to have a value greater or

[5]Medium Access Control

equal to the average bus driver's shift time, which can be deducted from the MyMoby portal.

After this steps, the device list will be made only out of devices with a high probability of belonging to an individual and so the ascent and descent analysis can begin.

4.2.4.4 Ascent and descent analysis

To affirm that a device entered the bus at a given time instant and left it in another is necessary to identify the period where the devices' sensor detected the device with continuity. The temporal distribution of a device is not continuous because composed by a set of isolated points, so an approximation has been taken and it has been defined when two detections' group are contiguous. What has been decided is that if two points have a temporal distance lower than a given threshold, then they are contiguous. The threshold can be defined in the configuration. All the time instants belonging to a group of device's detections[6] are considered already contiguous, so the analysis focuses on checking continuity among a series of groups of detections of the same device.
Having defined the temporal continuity is possible to explain the implementation of the algorithm.
In figure 4.9 the code is reported. The main idea behind this algorithm is that two time instants have to be selected to form an interval representing the period where the device was probably on the bus.
So at the first iteration left and right extreme are coincident then in the next iterations the right extreme is confronted with the first time instant of another group of detections to see if they are contiguous. In case two groups are contiguous, the time window is enlarged otherwise the two extremes of the time window are added to a list which collects all intervals of continuous detection. This list will contain an even number of elements, where the even indexed elements represent ascent times and odd indexed elements represent the descent times. Each pair of extremes will be enclosed in the "Variation" class, introduced in 3.3.2.3, and a list of them will be built, allowing to calculate the number of devices present on the bus at any instant of time by summing all variations up to the instant of time of interest.

4.2.5 Itinerary Matching

This section will talk about the itinerary matching algorithm which required the most effort to implement, tune and improve. First, the general implementation will be discussed, and then the choices that improved the algorithm will be highlighted.

[6]Device list's element returned by the device endpoint.

```
List<Long> extremes = new ArrayList<Long>();
for (int i = 0; i < currentGroup.size(); i++) {
   // time window begin
   Long newTimeInstant = currentGroup.get(i).getStartTime();
   if (rightExtreme == null) {
      leftExtreme = currentGroup.get(i).getStartTime();
      rightExtreme = currentGroup.get(i).getEndTime();
      continue;
   }
   // enlarging process
   if (Math.abs(rightExtreme - newTimeInstant) <
       CONTINUITY_THRESHOLD) {
      rightExtreme = newTimeInstant;
   } else {
      // end of continuity add points to the set of extremes
      extremes.add(leftExtreme);
      extremes.add(rightExtreme);
      leftExtreme = null;
      rightExtreme = null;
      i--;
   }
}
```

Figure 4.9: Code for the enlarging time window.

4.2.5.1 Algorithm overview

The itinerary matching algorithm is applied to each segment, and it consists of an initialisation part where the 3.15 and an object able to interact with MongoDB to perform geographic queries are instantiated. To initialise the line probability object is necessary to have the data of all bus lines which the bus currently in the exam can follow. Since there is no way of automatically infer which lines a bus can follow a manual mapping has been performed on the database associating to each router's ID[7] a set of line identifiers. In figure 4.10 it is shown an example of a MongoDB document containing this mapping.

The line probability object is initialised by associating to each couple line-itinerary an initial probability and what is defined as a sequence number.

The initial probability is either 0 or a high probability (> 0.7) if the itinerary was the result of the previous segment classification.

After the initialisation part is over, the operation that has to be performed is to

[7]The router is installed in the same bus as the devices' sensor.

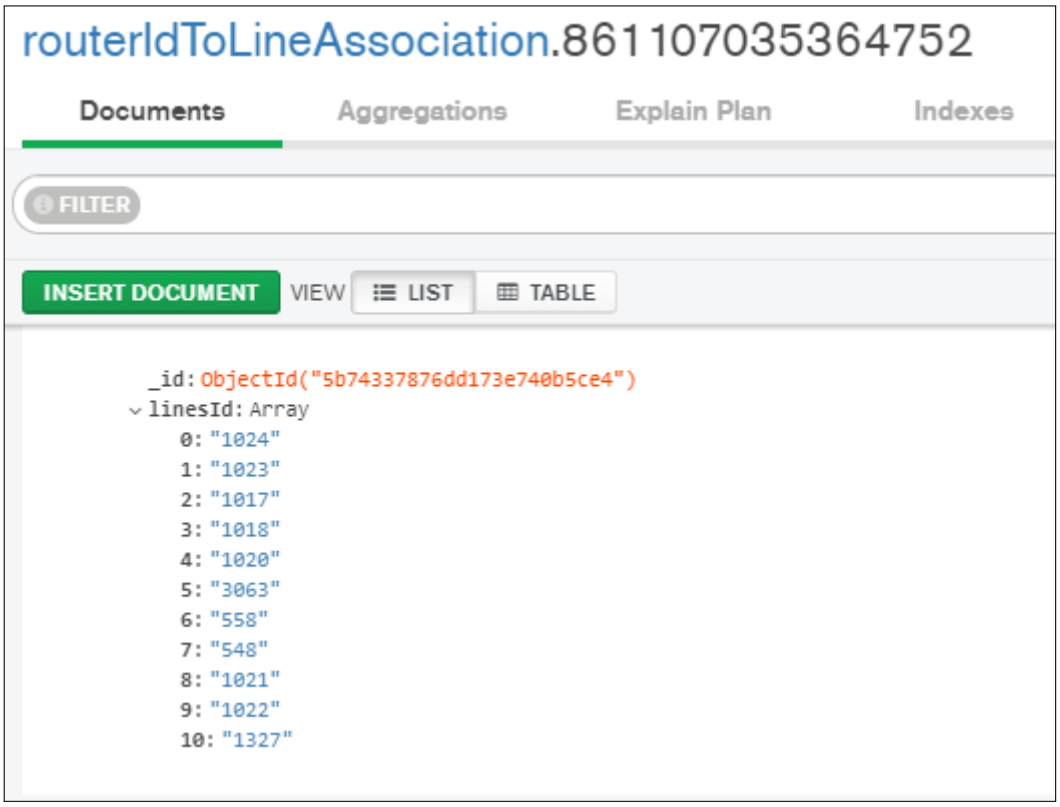


Figure 4.10: Mapping between a router ID and a sequence of bus line identifiers. On top is specified the router ID while on the bottom there is the sequence of line ids.

confront the path the bus has followed with all the possible itineraries using the mathematical model introduced in 3.3.2.7. In practice for every coordinate, that will be referred as time point in this section, of the bus path three operations are made:

1. copy the time instant of the point inside the line probability object, this has the utility of keeping a global reference to which time instant is being analysed.
2. using the current time point update the probability of being on each possible itinerary, applying the D function and checking that the bus is following the bus stops sequence for the itineraries having high probability.
3. select the itinerary having the highest priority as the best itinerary for the time instant. This operation will come into action later when the classification of the segment has to be calculated.

There is a particular case that allows determining right away which is the classification of the segment and that is when, after having calculated the probability

associated to each itinerary, only one itinerary has a high probability associated. This case happens when the bus follows a road which is unique to a single itinerary.

4.2.5.2 P function implementation

The calculation of the probability associated to each itinerary requires the computation of the P function which is dependent on the D function which requires locating the closest point for each itinerary path to the current time point.
Since searching the minimum distance between the current time point and each point of each itinerary is computationally expensive if naively done, this functionality has been implemented using MongoDB geographical query capability.
The approach consists in loading all points of the itinerary in a MongoDB's collection having as an identifier the itinerary identifier and as structure of each document equal to the one shown in figure 4.11. Also each collection needs to be geographical indexed, meaning that it must be built a "2dsphere" geographical index which if defined allows sending geographical queries to a collection. The geographical index can be created programmatically.
Once an itinerary is loaded inside the database, it will be reused for all future queries. At the moment there is no functionality checking the consistency of the itinerary loaded on the database with the ones returned by the transport society API.
Every time the D function matches a point causing the probability associated to the itinerary to go above a small constant ϵ then the next queries will start their matching from a subset of the points belonging to an itinerary. The first point of the will have as lower bound the index of the previously matched point and as upper bound the last point of the itinerary's path. If the probability for a given itinerary becomes high (>0.7) then the probability will be altered as defined in 3.3.2.12.
After the P function has been calculated for all itineraries, the line and itinerary identifiers are saved to a list indicating for each time instant which is the itinerary having the highest probability. This list will be used to classify the segment by calculating the occurrence of each couple line-itinerary and the couple having the highest frequency will be the result of the classification.
The result of the classification is enclosed in the segment classification class introduced in 3.3.2.6.

4.2.5.3 Loading an itinerary on MongoDB

The data describing the path of an itinerary is a list of coordinates indicating the vertices of straight lines. If these vertices are joined in order the resulting polyline would coincide to the first and last bus stop of the itinerary if the polyline is positioned with the first and last vertex coincident to the first and last bus stop. The vertex information is insufficient to perform the itinerary matching since two

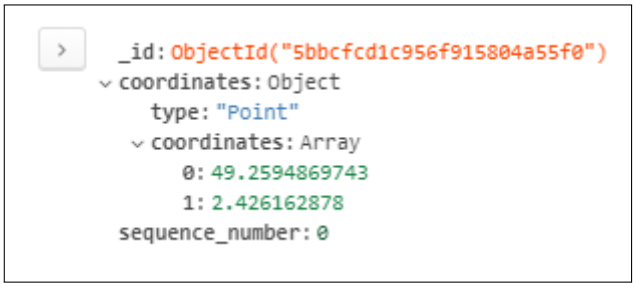


Figure 4.11: Representation of a itinerary's point inside MongoDB.

vertices can be put at an high distance in case the bus path includes a long straight road.
To overcome this problem when the itinerary is loaded inside MongoDB not only the vertex are loaded, but also all the points belonging to the geographical line interpolation between each pair of adjacent vertices.
To avoid approximating this problem to a plane problem it has been used a library called "GeographicLib" which is capable of performing coordinate interpolation between two geographical coordinates. The interpolation logic has been wrapped inside a class called "GeoInterpolator" that given in input two geographical points can return all points belonging to the line that connects them using the capabilities of the library mentioned before. The "GeoInterpolator" class is structured as a Iterator, so it implements the .hasNext() and .next() methods.
In figure 4.12 the principle on how to use the library to get the point belonging to the line interpolation of the two geographical vertexes using a parameter "t". This

```
Double t = 0.5;
Geodesic geod = Geodesic.WGS84;
GeodesicLine line = geod.InverseLine(lat1, lng1, lat2, lng2,
    GeodesicMask.DISTANCE_IN |GeodesicMask.LATITUDE |
    GeodesicMask.LONGITUDE);
GeodesicData g = line.Position(distance * t,
    GeodesicMask.LATITUDE | GeodesicMask.LONGITUDE);
return new GeoPoint(g.lat1, g.lon2);
```

Figure 4.12: Example of use of the "GeographicLib" to select the middle point of the line connecting the first point (lat1, lng1) and the second point(lat2, lng2).

parameter can have values in the interval [0, 1] and allows to select any unique point of the line. In order to obtain a meaningful sequence of points representing the line connecting the two vertices, the parameter will be initialised at 0 and each .next() call, in the interpolator, it will be incremented by a specific value called step. The

step is defined as $\frac{1}{N}$ where N is the number of points the interpolator can return between the two vertices. The parameter N can be defined through configuration. The procedure of loading the itinerary inside a collection inside MongoDB is done uniquely when in the DB there is no collection having as an identifier the itinerary identifier.

4.2.5.4 Use classification to determine stop sequence

Once all segments are classified the stops table for the sensor and date in exam can be finally generated. The information about the classification about each segment is used to produce the sequence of bus stops the bus passed by during the day as explained in 3.3.2.12.

4.2.5.5 Optimization

The use of MongoDB for the calculation of the D function speeds up the process of calculating the probability associated to the itinerary for each time instant, but since the high volume of time instants, other improvements have been made to the algorithm.

1. Access time to line probability data structure.
2. Calculation of best couple line-itinerary of the time instant.

The "LineProbability" class is composed of five arrays where elements having the same index are correlated between them. In figure 4.13 the actual declaration.
The arrays are filled in the same order on which the iteration of all possible bus

```
private int[] lineIds;
private int[] itineraryIds;
private double[] probabilities;
private int[] coordinateSequenceNumber;
private Stop[] nextStops;
```

Figure 4.13: Data structure used in the "LineProbability" class.

line data's itineraries would be done. Doing this allows to the method calculating the probability for each time point of the segment to access this structure using a manually incremented index allowing for direct access to the structure.
The second optimization consist in calculating the maximum value of probability every time the "LineProbability" object is updated. The optimization consist in keeping a reference to the maximum value of probability in all moments so that when a new value of probability is sent to the object then it can be immediately

confronted with the maximum value, updating it in case the new value is greater than the max.

4.2.6 Metric computation and storage

The previous sections introduced the device analysis and the itinerary matching implementation. This section will explain how their results are joined together to obtain the line's utilization.

4.2.6.1 Interval of validity

After having executed the algorithms of itinerary matching and ascent and descent analysis, the "StopsTable" contains the sequence of all bus stop the bus passed by during the day, but not all of them are valid.
Observing the MyMoby portal a pattern came out where the bus was seen in motion for the whole day, but no devices were detected in that period. It is not clear whether it is due to a system fault or the bus is empty. In any case this event influences negatively the average number of devices on board, so it has been decided to build an interval of validity defined from the time instant composed by the time instant of the first detection[8] and the last time instant. Every entry in the "StopsTable" being temporally out of this interval will be removed from the "StopsTable". The metric will be computed only on the interval of validity. In figure 4.14 an example shows, that the first devices heard during the day are not accounted.

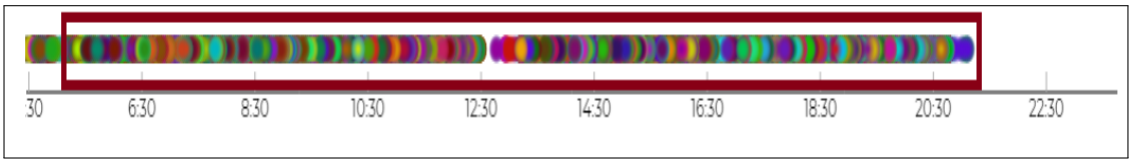


Figure 4.14: Interval of validity where the metric will be calculated.

4.2.6.2 Storing the result

After that, the metric is computed on every valid entry of the "StopsTable" the result has to be stored on the DB. It has been decided to create a collection on MongoDB for each line and put inside the entries of the stops table referring to itineraries belonging to the line. This choice has been made, so that is possible to compute two information:

[8]In the set of devices belonging to users.

- overall usage of the line.
- usage of an itinerary, considering only the stops table entries coming from that itinerary.

In figure 4.15 the structure of the document of a StopsTable's entry.

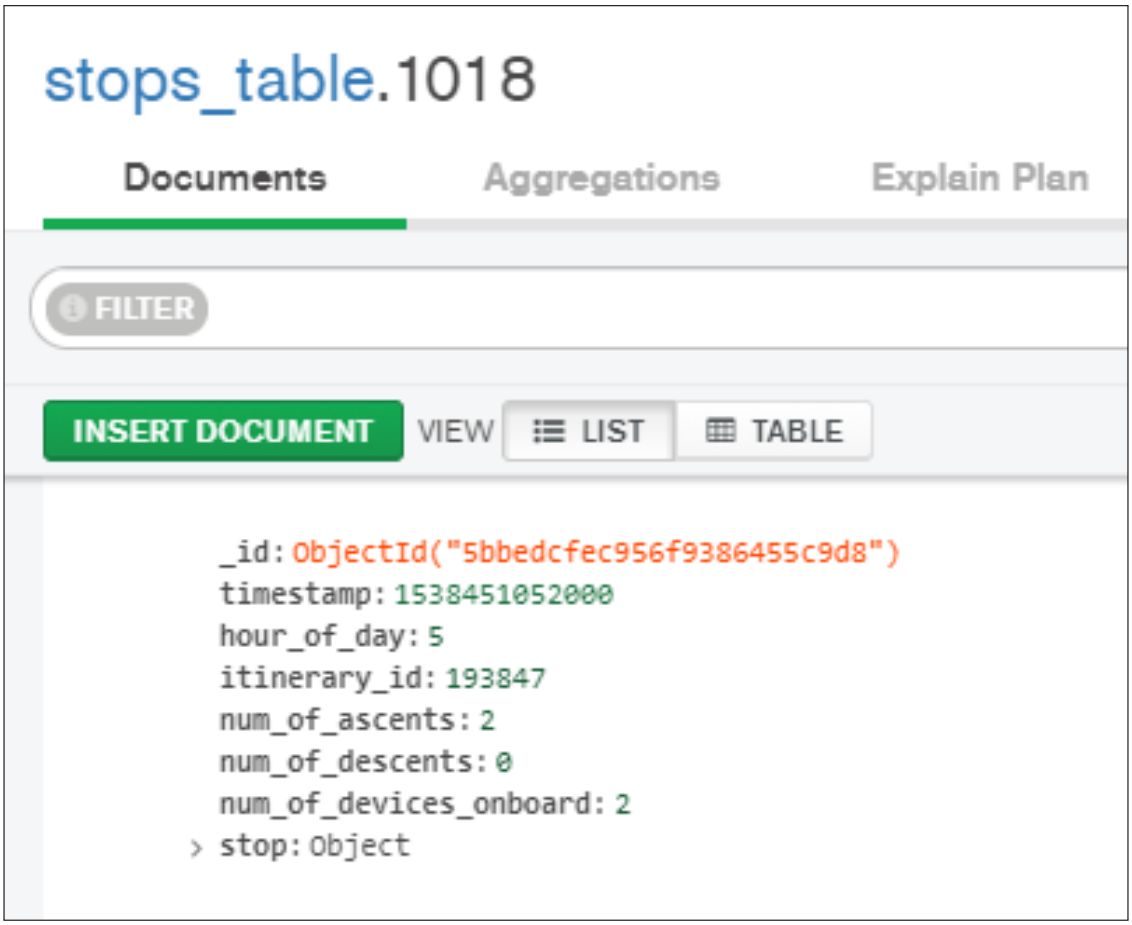


Figure 4.15: StopsTable entry in the db.

4.2.7 Configuration

In the previous sections, the implementation of the main algorithm has been discussed without going into the details of the configuration of them that instead will be discussed in this section.
Since the Classifier relies for most of its need on having an instance of MongoDB running, it has been decided to deploy the configuration in a collection on the DB. From the application perspective, the access is wrapped by a singleton class called "Config". This class when invoked can have two possible behaviours:

1. if the collection containing the parameters is not found on the MongoDB node at which the application is connected, then it will create a collection deploying a set of key-values containing the default values of the parameters, and then return the default value for the key that has been asked.

2. otherwise it will load the parameter from the database.

After that the default values are loaded in the DB, is possible to modify them accessing and updating the table with any MongoDB client.
In the table 4.1 are reported all the parameters name along with a description and default value.

4.3 Data visualizer

In this section, it will be presented the implementation of the data visualizer component. Recalling 3.3.3 the primary operations performed by this component are:

- Query the data with some parameters.
- Aggregate the retrieved data.
- Build the graphic representation.

This component, as stated before, has a graphical interface and is implemented as a one-page application and it is divided logically in UI and data access layer.

4.3.1 UI

The UI consist of an HTML page that is dynamically updated using JavaScript and its capability of making asynchronous HTTP requests which are forwarded to the data access layer of this component.
The UI can be divided into 3 logical sections:

- **Parameters selection**: includes a series of inputs to be provided as query parameters for the data access layer. The possible parameters are:
  - Line identifier.
  - Itinerary identifier.
  - Time of the day.
  - Time period in which calculate the metric.

  The period is composed by a starting and ending date in which calculate the metric. Tuning the values of start and end date is possible to obtain three particular views on the metric.

1. Start date = end date -> daily average, useful mostly for system components validation.
2. Start date < end date -> metric in a certain period, useful to focus on specific events like school beginning or specific seasons.
3. Start date and end date not specified -> all time average utilization of the itinerary.

- **Stops' graph**: section of the UI where is displayed the first form of graph in which the metric can be visualized.
  Figure 4.16.

- **Map**: a map where the projection of the graph of the bus stops is displayed. In the projection the graph nodes are positioned on the bus stops' coordinates of the bus stops they represent.
  Figure 4.17.

In order to update the map and graph section, forward requests to the data access layer several scripts written with JavaScript have been implemented and added to the page. Following it is a list of all the scripts implemented:

- **init.js**: In this file are defined the implementation of each event handler to the interactive components on the UI, and at page startup it is called the data access layer to get the meta-data of the system which consists in the list of all line identifiers known to the system.

- **data.js**: This script handles the validation of the input parameters and it contains the logic to make requests to the data access layer and format the response to be processed by the others scripts that builds the graph. The requests to the access layer are made using JQuery's ajax method.

- **mapScript.js**: In this script is defined the logic to handle the map section, calling the methods of the Leaflet library for web map handling. This library will be described in later sections.

- **nodeLinker.js**: This script handles the creation of the itinerary utilisation graph, in order to create a graphic representation the script relies on the methods of the D3 library.

Two of the listed scripts rely on JavaScript libraries so in the following section the libraries will be introduced and it will be presented how their functionality has been used to implement the required features.

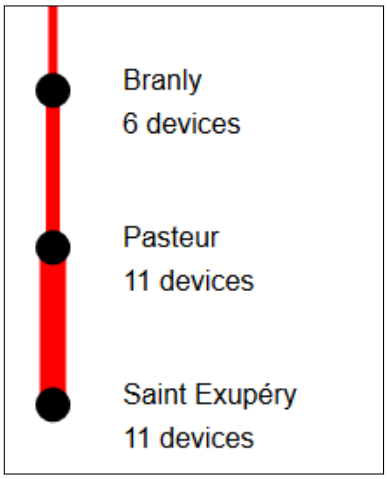


Figure 4.16: Graph showing the metric between 3 bus stops.

.

4.3.1.1 D3 Library

D3 (Data-Driven Documents or D3.js) is a JavaScript library for visualising data using web standards. D3 helps bring data to life using SVG, Canvas and HTML. D3 combines powerful visualisation and interaction techniques with a data-driven approach to DOM manipulation.
D3 employs a declarative approach, operating on arbitrary sets of nodes called selections; numerous methods for mutating nodes: setting attributes or styles; registering event listeners; adding, removing or sorting nodes; and changing HTML or text content are provided by the library. Styles and attributes can also be modified dynamically depending on the value of the data that can be associated with an element. This operation is called data binding, and it consists of associating to each select an object of data to use to compute the properties and the attributes of each node selected. In the next figure is shown an example of how is possible to modify the property "font-size" depending on the data associated to the nodes "p".
This library is often used for the generation and update of SVG[9] elements. SVG is a way to render graphical elements and images in the DOM. In the following example it is shown how to add an rectangle element to an SVG container.

Used functionalities Using the dynamic properties to bound data of D3 library is possible to draw the first typology of the introduced graphs. First, it has been developed a data structure that holds the information needed for the graphic representation called Node (fig 4.20). Each Node represents a bus stop belonging to

[9]Scalable Vector Graphics

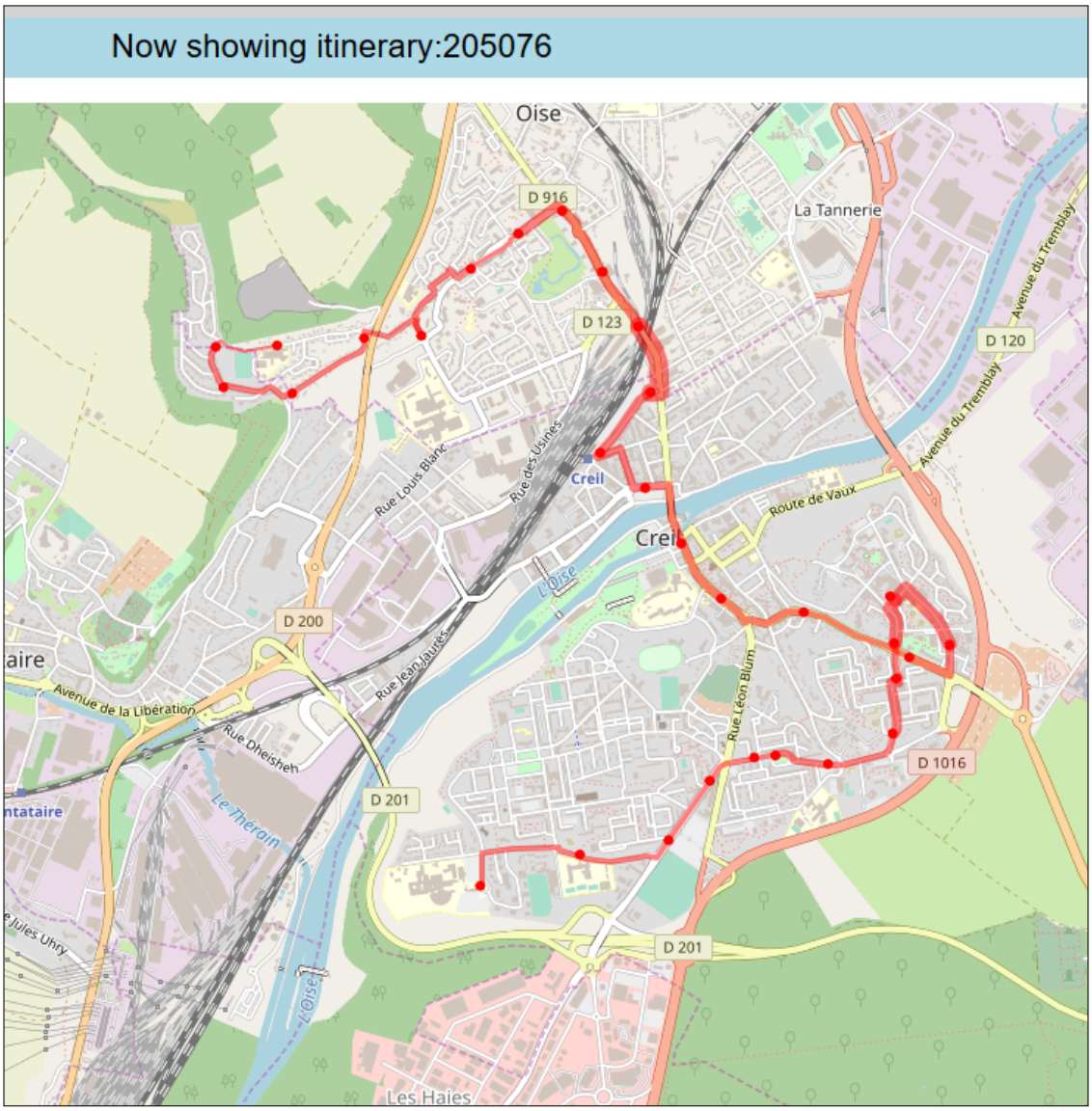


Figure 4.17: Map section of the UI.

```
d3.selectAll("p")
.data([4, 8, 15, 16, 23, 42])
.style("font-size", function(d) { return d + "px"; });
```

Figure 4.18: Dinamic transformation of the font-size of several paragraph depending on the data bound to the paragraphs.

```
//Select SVG element
var svg = d3.select('svg');
//Create rectangle element inside SVG
svg.append('rect')
.attr('x', 50)
.attr('y', 50)
.attr('width', 200)
.attr('height', 100)
.attr('fill', 'green');
```

Figure 4.19: Creating a rectangle SVG element using D3 library.

an itinerary, and a set of them is built for every stop belonging to an itinerary that has to be drawn. The set of Node objects is bound to a sequence of SVG elements

```
Node = function (id, nodeName, numOfDevices, coordinates) {
   this.stopId = id;
   this.name = nodeName; // bus stop Name
   this.numOfDevices = numOfDevices; // average number of device at
      the stop
   this.coordinates = coordinates; // coordinates of the bus stop
};
```

Figure 4.20: Node data structure.

of type 'circle' and 'line'. Assuming the set of Node has cardinality N then the code implemented will create N circle elements and N-1 lines. Once the elements have been created, then their attributes are modified to create the graphical representation. Following there is the list of attributes set for each element and how they contribute to the graphical representation.

circle The circle element have to be disposed in an increasing distance from the origin of the representation basing on their position in the sequence. Following there is the list of attributes to obtain this effect assuming vertical representation, "cy_0" vertical offset to the origin, "i" index of the node in node sequence, and "l" length of the line connecting two circles:

- cy (center's y coordinate): $cy_0 + i * (l + 2 * r)$
- cx (center's x coordinate): Constant value A
- r (radius): Any value

line The line elements must connect each pair of graph's node with a variable line stroke's width. Assuming "r" as radius of a circle element.

- x1 (first vertex's x's coordinate): Constant value A
- x2 (second vertex's x's coordinate): Constant value A
- y1 (first vertex's y's coordinate): $(2 * r * l) * i + cy_0 + r$
- y2 (second vertex's y's coordinate): $(2 * r * l) * i + cy_0 + r + l$
- stroke-width: mapNumberOfDeviceThicknessFunction()

The stroke width attribute is dependent from a function which can not be expressed as a mathematical formula but requires some explanation. D3 library provides a set of helpers function to work with sets of data. One of them is the scale function which can map values in an interval to values inside another interval. In this case, it is used to map the interval composed by the $[numOfDevices_{min}, numOfDevices_{max}]$ to the one [1, 15] representing the stroke width.

4.3.1.2 Leaflet library

Leaflet is a widely used open source JavaScript library used to build web mapping applications. First released in 2011, it supports most mobile and desktop platforms, supporting HTML5 and CSS3. Along with OpenLayers, and the Google Maps API, it is one of the most popular JavaScript mapping libraries and is used by major websites such as FourSquare, Pinterest and Flickr. An example of basic usage of Leaflet is shown in figure 4.21.

```
var map = L.map('mapdiv', {
   center: [51.505, -0.09],
   zoom: 13
});
L.tileLayer(
   'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
      attribution: 'Map data &copy; <a
          href="http://openstreetmap.org">OpenStreetMap</a>
          contributors, ',
      maxZoom: 20
   }
).addTo(map);
```

Figure 4.21: Creation of a Leaflet's map object and introduction of a tile layer coming from OpenStreetMap. This operation will display a map in a div having as id 'mapdiv'.

Leaflet exposes a series of primitives that can be used to draw vectors graphics layers on top of the map. Following there is the description (taken from Leaflet documentation) of some functions used in this thesis work to draw this kind of layer:

- L.circle([lat, lng], options): Instantiates a circle object given a geographical point, and an options object which contains the circle radius.
- L.polyline([latLng1, ..., latLngN], options): Instantiates a polyline object given an array of geographical points and optionally an options object. In the options is possible to specify also the width of the line.

Those primitives allow the drawing of circles and lines over imposed on a map thus enabling the projection of the stops' graph on the map. The data structure used it is still the set of Node objects introduced before. To draw the stops is sufficient to access the coordinates' field of a Node and pass it as the first argument of the L.circle primitive. Following is reported the function that draws a circle on the map based on the stop position.
To draw the polyline connecting two stops is necessary to provide also the vertex

```
var createStopCircle = function (node) {
   var coordinates = node.getCoordinates();
   var circle = L.circle([coordinates.lat, coordinates.lon], {
      color: config.color,
      fillColor: config.fillColor,
      fillOpacity: config.fillOpacity,
      radius: config.radius
   }).addTo(map);
   self.stops.push(circle);
};
```

Figure 4.22: Function to represent a bus stop on the graph.

list of each straight line composing the path. This information is contained in the Geometries field of an itinerary.
The full path connecting the first stop the last one can be drawn following this steps:

- iterate on all Node objects
  - for each Node recover the associated Geometry object, which is the series of coordinates that lead to the next stop, from the bus line data of the itinerary.

– use D3 Scale function to get the width associated to the Node's number of devices.

– call function reported in figure 4.23.

```
var drawPolyLine = function (geometry, lineWidth) {
   if (geometry) {
      var coordinatesList = geometry.Geometry[0].coordinates;
      var latLngs = [];
      for (var i = 0; i < coordinatesList.length; i++) {
         var coord = coordinatesList[i];
         latLngs.push(new L.LatLng(coord[1], coord[0]));
      }
      var polyline = L.polyline(latLngs, {
         color: "red",
         weight: lineWidth
      }).addTo(map);
   }
};
```

Figure 4.23: Function to draw the path connecting two stops on the map.

4.3.2 Data access layer

Since the UI cannot access directly the data stored on MongoDB, it has been decided to implement a layer accessible through AJAX requests capable of querying the database. Data's endpoints have been implemented and the UI can send HTTP requests to get the needed data. The endpoints implemented are two, which alone are capable of satisfying the requirements needed for the UI. They are explained in the following list:

- **/stopTable**: This data endpoint returns to the UI an aggregation of stops table's entries belonging to a specific itinerary of a bus line and the transport society data about the line in case it is needed for additional processing. The request's parameters restrict the aggregation to specific periods of time.

- **/lineIds**: To access the stops table is mandatory to send in the body of the request a bus line identifier, so this endpoint returns all possible identifiers that can be queried.

These endpoints have been developed using Java Servlets which are Java Objects which are associated with a web resource, and their code is executed to process all

the HTTP requests sent to them.
The implementation of the endpoint returning the identifiers of the bus lines is elementary, and it consists in connecting to the database and reading the name of the collections where stops table entries are stored and then packing this list of strings in JSON Array.
The implementation of the other endpoint requires more explanation as more operations are done.
The operations performed by the second endpoint are:

- validate request parameters.
- submit an aggregation query to MongoDB.
- perform a HTTP get request to the transport society endpoint to get the data of the request line.
- pack the result of the two previous operations in one JSON response.

Aggregation The operation of aggregation is done using MongoDB's Java Driver aggregation functions which need the programmer to define two phases: the match and group phases. The matching phase consists into specify a filter to select a set of documents on which the aggregation should be computed. In this phase the parameters about the period and the hour of the day are used to match a specific set of table's entries. The matched entries are then sent to the group phase where they are groups are formed identified by the pair itinerary and stop identifiers, and for each group, the average of the number of devices is then calculated. In figure 4.24 it is reported an example of MongoDB's aggregation function.

```
aggregationQuery = Arrays.asList(
   match( // matching phase select the set of documents to aggregate
      and(
         gte(HOUR_OF_DAY_MONGODB_FIELD_NAME, startHour),
         lte(HOUR_OF_DAY_MONGODB_FIELD_NAME, endHour))),
   group( // group and aggregate
      and(
         eq("itinerary_id",
         "$itinerary_id"),
         eq("stop_id",
         "$stop.id")),
         avg("num_of_devices_onboard",
         "$num_of_devices_onboard")));
```

Figure 4.24: MongoDB aggregation syntax example.

The results of the aggregation and of the HTTP request to the bus line data's endpoint are joined in a JSON containing two fields: lineData and stopStatistics. This object is sent as a response to any valid HTTP request. The UI will use it to build its graphic representations.

Parameter Name	Description	Default Value
DECREMENT	Indicates the number of days before the current one where to select data to analyse.	-1
LINE_PROBABILITY_ THRESHOLD	Threshold value above which a probability is considered to have an high value	0.7
ITINERARY_MATCHER_ RESET_TIME	Time difference between two positions above which the LineProbability object is reset.	1h
BUS_PATH_SEGMENT_ START_HOUR	Indicates the first time instant where both positions and device are considered valid.	5h
P_FUNCTION_A_ PARAMETER	Weight parameter 'a' used in the P function.	0.9
MONGODB_ITINERARY_ MATCHER_ INTERPOLATION_POINTS	Number of points generated by the GeoInterpolator when loading an itinerary the db.	50
MONGODB_ITINERARY_ MATCHER_MATCH_ DISTANCE	Maximum distance of matching when confronting a point to all points constituting an itinerary in the D function calculation	50.0m
MONGODB_ITINERARY_ MATCHER_STOP_MATCH _DISTANCE	Maximum distance of matching when finding the nearest stop to a point in the stop sequence matching algorithm.	20
MAX_DISTANCE_ BETWEEN_TIME_ INTERVALS	Maximum allowed period of stay device on board, any device staying longer is filtered away in device filtering algorithm.	3h
MIN_RELEVATIONS	Minimum number of device detections needed to consider a group of device detections valid.	2
NEARBY_RADIUS	Radius of a circle where all points included in the circle are considered concentrated.	20.0m
CONTINUITY _THRESHOLD	Time duration indicating under what time difference two device detections are considered continuous in the ascent descent analysis.	20min

Table 4.1: Configuration parameters for the Classifier component.

Chapter 5

Results

This chapter will talk about the validation of the work which has been done describing the details of the methods followed and the error rate on the classifiers implemented.

5.1 Requirement validation

In this section all the requirements will be validated one by one, enunciating the validation method followed by the error rate obtained following the validation method.

5.1.1 R1 - Device Filtering

This requirement refers to what is explained in 3.3.2.2, that is the fact that not all the device's detection belongs to devices owned by people.
The validation of this requirement consist in making the system output for each device, detected on a given date, its identifier coupled with its classification. All the results of classification are then confronted with the behaviour of the devices on the MyMoby portal (time distribution, positions occupied). The use of the portal to validate the classification consist in identifying the pattern, for each device, looking at the patters presented in 4.2.4.2.
The console of the browser is exploited to run a simple JQuery instruction which immediately displays the positions of a device if this is present in the device list on the UI. This instruction speeds up very much the process of validation.
In figure 5.1 it is shown the command.

```
$("#deviceList").val("<idenfitier-to-search>").change()
```

Figure 5.1: Quick search on MyMoby devices' list.

Following an example of the validation it is reported.
In figure 5.2 it is shown the classification result of the filter. Using the JQuery

```
850e3f0426e0c9be2243d2ce95c332b6: User device on board
02:f9:84:b2:08:f5: Not a user device
8509984453b90bd914a8d292b20d747f: Not a user device
1b1ab184d8d3055f1a9467e272982a37: Not a user device
```

Figure 5.2: Console's output of the filter.

comand on the identifiers in the list produced it is possible to see very quickly their behaviour on map and determine if the classification was correct or wrong. In this example the classification was correct for all devices.
In figure 5.3 and 5.4 it is shown the behaviour of the first two devices in the list shown in 5.2.

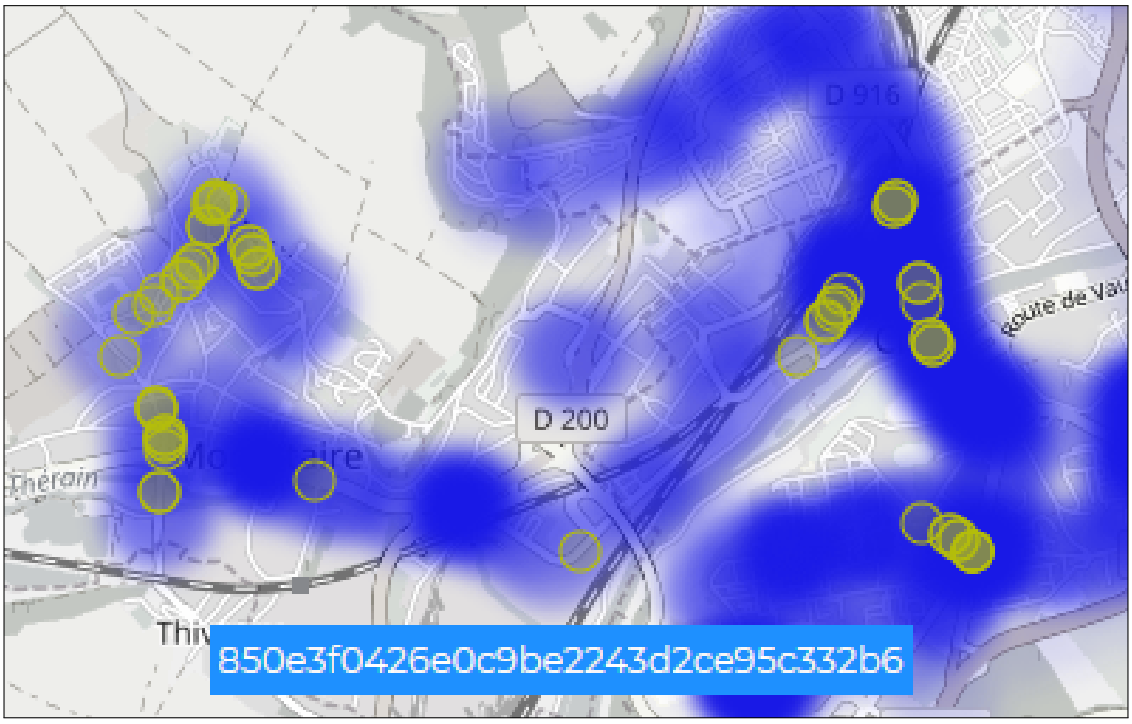


Figure 5.3: Device belonging to an human.

5.1.1.1 Error rate

The application of the validation method described before allows calculating the number of devices classified. In figure 5.5 it is shown the error on classification

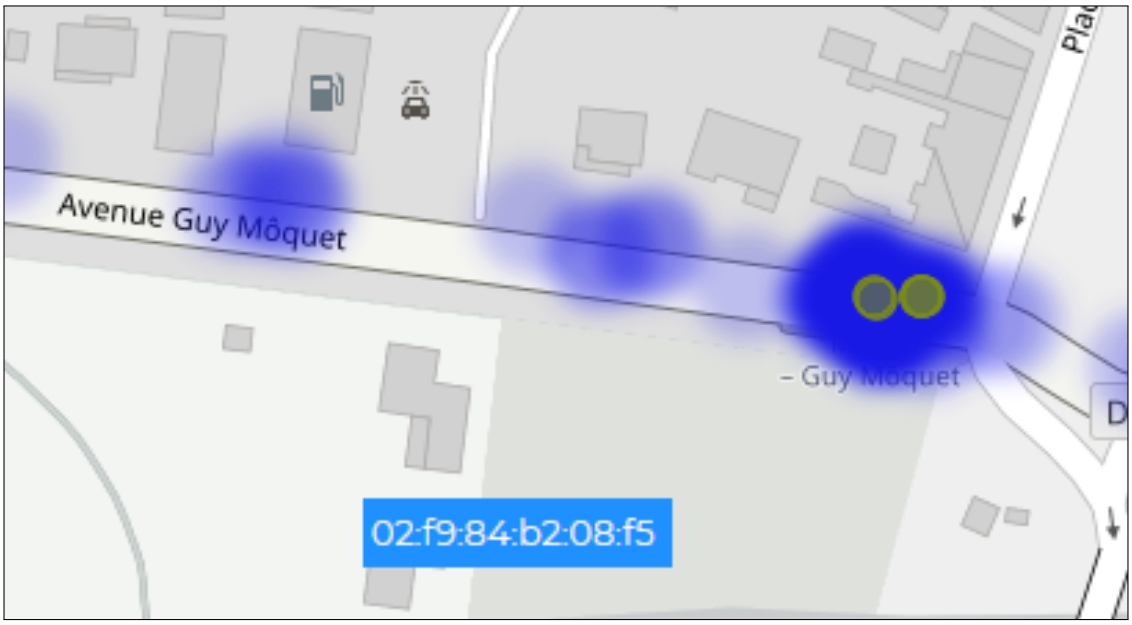


Figure 5.4: Device not belonging to an human, probably filtered because all its detections are concentrated at the end of a road.

calculated on each date of the month having the highest number of device detections since the system is running showing an average error rate equal to 9.13% with a variance equal to 0.11%. Since the validation method is time-consuming, because it relies on manual verification, it has not been possible to calculate the average error rate on all days available since the system is running which are more than 150.

5.1.2 R2 - Path classification

This requirement to be validated requires much effort due to the complexity of the problem, so the validation method used is manual and relies on sight. As done before, the system has been modified to output a string representation of the "StopsTable" composed by a set of lines each one indicating entry, and for each line, the fields showed is:

- Line identifier.
- Itinerary identifier.
- Bus stop name.
- Date time of arrival.

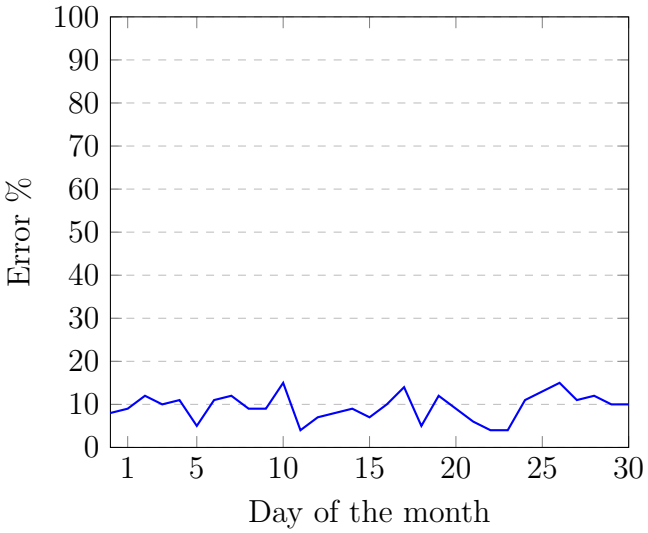


Figure 5.5: Chart showing the error rate on the device classification in the time period corresponding to a month of detections.

The output of the system allows to identify the periods of time where the bus was serving each line. The series of pairs of time instants defining an interval of time where the bus was serving a line is the starting point for the validation. In figure 5.6 the structure of the validation list (JSON representation is used for convenience):

```
[
  {
    interval: [t0, t1],
    lineIdentifier: 1024
  },
  {
    interval: [t1, t2],
    lineIdentifier: 1023
  },...
]
```

Figure 5.6: Validation data structure for itinerary matching.

After having build the intervals showing at which time the bus was serving a line, the MyMoby portal is used to see the path the bus did starting from the first time instant of the interval to the last time instant. At this point the path saw on the MyMoby portal has to be confronted with the actual path of the bus. Fortunately the transport society has a web site where is possible to see a map showing

the geometry of the itineraries the bus follows for each line. The map is shown in figure 5.7. Confronting the actual bus path with the one observed on the MyMoby

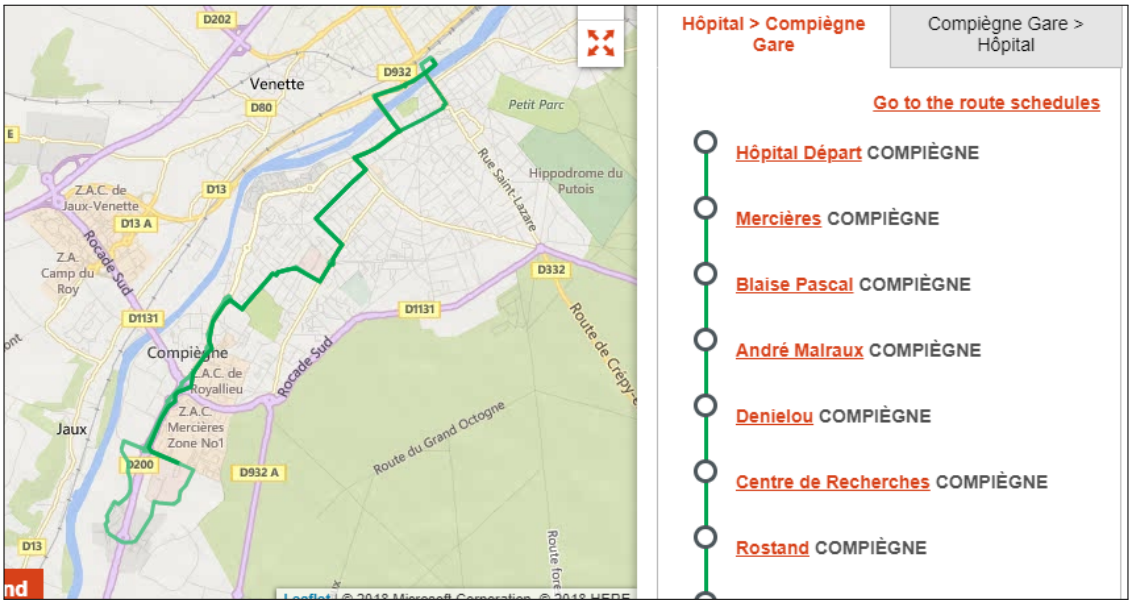


Figure 5.7: Graphical user interface of the website of the transport society used to validate the itinerary matching algorithm.

portal allows to say if an interval is correctly classified.

5.1.2.1 Error rate

For itinerary matching the evaluation of the Classifier is even harder because confronting two paths by sight is a time consuming processes. So the error rate has been calculated over a small period of two weeks from Monday to Monday.

The chart in figure 5.8 shows interesting things:

- The weekends present a lower error rate, this is probably due the fact that during the weekends the bus rides are limited and so the bus with the sensors on board was not used very much.

- There is a slight difference between the first week and the second week. Observing the path followed by the bus the problems of aliasing described in 3.3.2.10 where higher in respect to the second one. From all of this it is possible to deduce that aliasing has an high impact on itinerary matching correctness.

On the validation set chosen the average error rate is of the 15% with a variance of 0.59%.

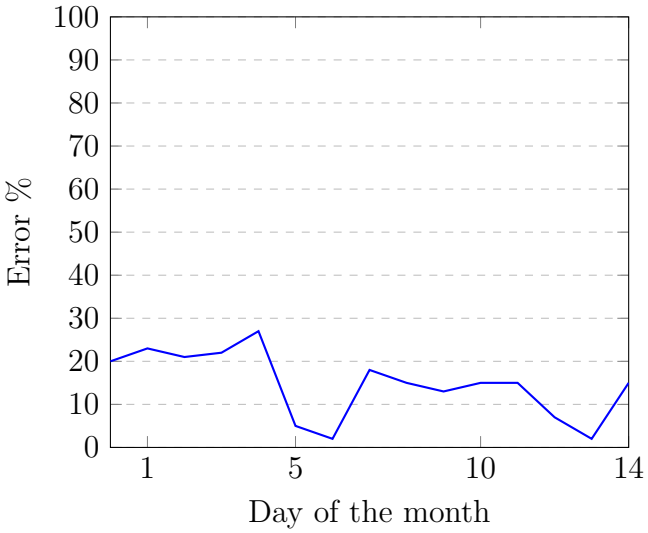


Figure 5.8: Chart showing the error rate on itinerary matching in the time period corresponding to two weeks.

5.1.3 R3 - Data aggregation

This requirement can be validated in a similar way as the ones discussed before using MyMoby portal and modifying the system to make a specific output, called validation list, just for the validation. The number of devices on the bus at each time depends mostly from the output of the ascent and descent analysis, so to validate and calculate the error that algorithm has to be validated.
The validation consist in modifying system output to print a list of lines telling the time of ascent and the time of descent. This list containing identifier, ascent and descent time will be compared with the behaviour of the device on the MyMoby portal. The JQuery instruction used in 5.1.1 can be reused also in this case to quickly search device in the device list. The validation method consist in checking that the initial and ending start point of the path that a device has made are nearby the stops which are associated to the ascent and descent time. This procedure has to be repeated for each device in the validation list.

5.1.3.1 Error rate

For this requirement it hasn't been done any statistical evaluation since the performance of the data aggregation are strictly related to the output of the device filtering that in most majority of cases produces devices' detection groups having time instants close together producing always a couple of "Variation". Furthermore, the validation of this requirement does not provide certainties about the fact that

the device actually left the bus at the time stated by the system for two main reasons which are that the device could have turned off or the person moved away in a zone of the bus where the sensors can barely catch the signals coming from the Wi-Fi.

5.1.4 R4 - Readability

This requirement does not require to be validated because it is satisfied already with its implementation. Since the requirement states only that the representation should be human readable. In the context of the thesis which is a proof of concept the requirement does not contain any additional constraints, so it can be considered satisfied.

Chapter 6

Conclusion

This thesis work was aimed at extending a software system of passive device detection of devices on board of the bus which had capabilities of just collecting and show the data detected from the bus, but not to extrapolate any further statistic out of it.
The extension consisted in extrapolating the statistic regarding the utilization of each bus line served by the buses where the sensors are installed, this required to calculate the number of devices on board of the bus for all bus stop. This problem was divided in two sub problems, which are matching the bus path to one of the possible itineraries and determining correctly the ascent and descent time. The resolution of this problem brought to the development of two classifications algorithms which reached high levels of accuracy.
The metric calculated from the classifications algorithms needed a way to be human readable so an UI having the role of building two graphical representation of the metric has been built. The representations built are a graph where the nodes represent the bus stops and the width of the edge connecting each node is proportional at the number of devices between that pair of stops. The two representations differ only from the fact that the first one is a linear representation whereas the second one is a projection of the first on a map where the node are positioned on the same coordinates of the bus stop they are associated to. The development of this software provides several advantages for the analysis of the bus line utilization and thus its optimization:

- **Quick statistics**: the algorithm developed are run every night on the data of the day before with a performance able to give access to the statistics already in the morning.

- **Straightforward graphs**: the graphs developed for this thesis work are easy to read and immediately highlight the presence of crowding for a specific bus line.

- **Effortless analysis** the system requires no user interactions neither in the collecting phase of the data nor the analysis.

The developed extension reached well its objective, as discussed in chapter 5 and leaves room for future developments.

Future development This paragraph will talk about what directions can be taken provided that the open issues have been solved.

- **New classification rules** The set of classification rules, both for the itinerary matcher and the device filter, can be extended by creating rules acting on other aspects of the data coming from the existing system thus improving even more the accuracy of this algorithms and reduce the effects of aliasing described in 3.3.2.10.
- **Tools for classify quickly the data** One of the main problems of this thesis was the methods of validations which relied entirely on observing an user interface showing a raw representation of the collected data. A future development could the one of developing tools to allow an operator to save the result of the classification resulting from his observation, allowing to create validation data that can be used to perform automatic testing.
- **Better classification algorithms** This development is related to the previous one. The creation of validation data that can be used in automated way opens the possibility of implementing more efficient classifications algorithms that rely on training data.
- **Portal for deploying and studying mobility plan** If the algorithms of classification is improved at point to easily represent reality it can be used to study performance in a deployment of a public transport plan, for example to demonstrate, with real-time dashboards, the effect of adding o removing a new bus line.

The one listed here are just few of the potential developments that this technology of passive detection provides. Any development will increase the quality of public transportation that more and more relies on IT solutions to improve the quality of service and provide new solutions.
To conclude considering what has been done in this work, it is fare to say that the knowledge of the utilization of each bus line obtained through low cost and efficient systems will allow transport societies to build and deploy better transportation plans that will match the mobility's needs of the community.

Bibliography

[1] Vladimir Agafonkin. Leaflet library, 2017. https://leafletjs.com/.

[2] Mike Bostock. D3 library homepage. https://d3js.org/.

[3] M. De Luca. *Manuale di pianificazione dei trasporti*. CNR.: Progetto finalizzato trasporti 2. Franco Angeli, 2000.

[4] Paolo Fadda. *Concenzione dei progetti di trasporto in ambiente sistemico*. Rubbettino, 2002.

[5] Francesco Lombardo. Progetto implementazione di un'architettura pervasiva per la raccolta di dati analitici relativi all'utilizzo dei mezzi pubblici. Master's thesis, Politecnico di Torino, 2018.

[6] E J Meyer, M D; Miller. *Urban transportation planning: A decision-oriented approach, second edition*. McGraw-Hill Higher Education, 2001.

[7] India Sarthak Agarwal International Institute of Information Technology Hyderabad Gachibowli, Hyderabad and India KS Rajan International Institute of Information Technology Hyderabad Gachibowli, Hyderabad. Analyzing the performance of nosql vs. sql databases for spatial and aggregate queries, 2017. https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1028&context=foss4g.

[8] Nathan To. How iot improves public transport for passengers, 4 2018. https://davra.com/2018/04/25/how-the-internet-of-things-improves-public-transportation-for-passengers/.