CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Driving context classification using pattern recognition

Master's thesis in Computer Science – Computer Systems and Networks

Mattias Henriksson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Driving context classification using pattern recognition
MATTIAS HENRIKSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96
Sweden
Telephone + 46 (0)31-722 1000

Abstract

The performance of a vehicle system is to a large extent dependent on the driving context, such as the road infrastructure, in which the vehicle is operating. In order to achieve improved performance, different vehicle system applications may need to take driving context parameters into account. In this thesis, we develop a pattern recognition framework that classifies driving context based on data recorded by vehicles (speed, steering wheel angle, etc.) in a naturalistic setting. We train the framework on a large data set of vehicle data annotated with map attributes from a map database representing driving context. An inventory is made on the map attributes, finding two kinds of global-scale driving context classes to classify: (1) whether a vehicle is driving in a city or not, and (2) the functional class of the road the vehicle is driving on. We then review four pattern recognition models: Logistic Regression, SVM, Hidden Markov Model, and a simple Baseline model, comparing their ability to classify (1) and (2). We find that all models reach similar overall prediction accuracies, ranging between 76 % - 80 % for classification task (1) and 84 % - 86 % for task (2), but that the models differ slightly with respect to per-class prediction accuracy.

Acknowledgements

I would like to thank my supervisors Dag Wedelin at the Computer Science and Engineering department at Chalmers University of Technology, and Johan Engström at Volvo GTT, for their very valuable supervision and advice throughout this project. I also wish to thank my examiner Peter Damaschke for his support. Finally, a special thanks all the employees at Volvo TSS for all their help.

Contents

Chapter 1

Introduction

The main purpose of a vehicle system is to transport passengers or goods from point A to point B. In addition, there are subgoals such as fuel efficiency, passenger safety, and minimizing wear on the vehicle. These goals are to be achieved under some set of constraints that are dictated by the context in which the vehicle is operating, influencing the vehicle and its driver. Such context could for instance be the road infrastructure, traffic density, and weather conditions. In order to achieve improved performance, different vehicle system applications may need to take driving context parameters into account.

An example of a vehicle system application that needs to be driving context aware is *driving style evaluation*. The individual driving style (e.g. aggressive versus defensive driving) of a driver has a significant impact on a vehicle's performance, affecting parameters such as safety [23], fuel consumption [24], and wear on the vehicle. The ability to evaluate driving style has therefore become a crucial task in today's vehicle systems. Driving style evaluation is used in applications such as *Driver Coaching Systems* (see [11] for an example on such a system), which provide education and training programs intending to change the driver's driving style. In haulage firms with large vehicle fleets such training can significantly reduce costs related to for instance fuel consumption and vehicle crashes (vehicle repair, downtime, etc.). Furthermore, driving style evaluation is also utilized in so called *usage based insurance* (UBI), in which automotive insurance companies reward safe drivers by dynamically setting insurance costs based on the behavior of the driver (*Progressive Insurance* [13] and *MyDrive Solutions* [26] are examples of two companies that provide such insurance). Drivers with a risky driving behavior pays higher insurance premiums than drivers with a safe driving behavior.

Driving style evaluation can be performed based on vehicle data (speed, steering wheel angle, brake pedal usage, etc.). For instance, an aggressive driving style could be detected by measuring the brake pedal usage. However, the current driving context also affects vehicle data. For example, driving in a city center yields a higher brake pedal usage frequency than driving on the highway. Thus, in order to evaluate driving style properly, we must take the current driving context into account, filtering out the aspects of the vehicle data that is a result of the current driving context and not the driver's driving style. Figure 1.1 visualizes this idea.

Moreover, the purpose of driving style evaluation applications is to change a driver's driving style. Therefore, it is vital that the driver subject to the evaluation understands the evaluation calculations, i.e. why he/she received a certain evaluation score. This way he/she will know what behavior to change to improve

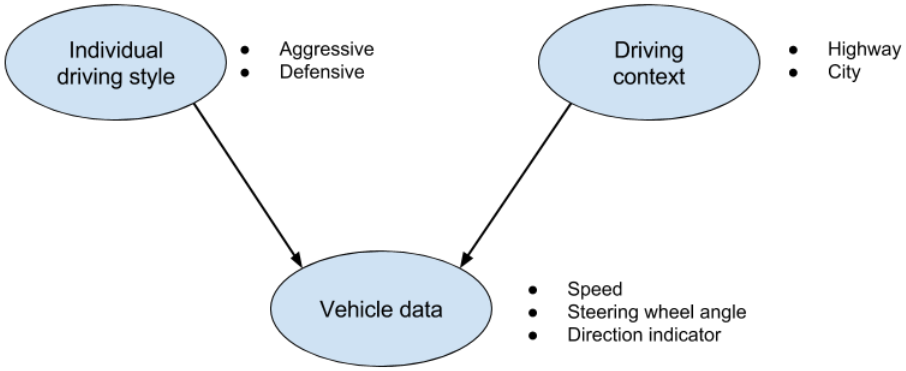


Figure 1.1: Vehicle data depends on two factors: the drivers individual driving style and the driving context in which the vehicle is operating. In order to evaluate driving style properly, we must take the current driving context into account, filtering out the aspects of the vehicle data that is a result of the current driving context and not the driver's driving style.

his/her evaluation score. In other words, the evaluation must be comprehensible and "actionable". This can be achieved by making all parameters (including driving context) in the driving style evaluation calculation explicit and interpretable to the driver. Such an approach requires that, when the driver interprets the evaluation score, driving context data used in the evaluation should be accessible to the him/her. For instance, a driver should be able see that at time t_0 when he/she was driving on the highway, a certain driving pattern resulted in driving evaluation score X, and at time t_1, when driving in the city, it resulted in score Y.

Driving context data can also be an important parameter in distraction-detection systems. A general method to detect driver distraction and risky driving situation is by looking at the driver's eye glancing patterns ([7]). Furthermore, studies ([27]) have shown that eye glancing measurements are also dependent on the current driving context. For instance, in complex driving context with high degree of curvature and traffic density, drivers spend more time with their eyes on the road. Accordingly, [27] concludes that functionalities based on glance metrics evaluation, such as distraction-detection systems, must take driving context parameters into account.

In this thesis, we develop a pattern recognition framework that classifies driving context based on an input of vehicle data (speed, steering wheel angle, etc.). The work was conducted at Volvo Group Trucks Technology (GTT) in Gothenburg, Sweden. Our study builds on a study that was performed at Volvo in 2001 ([4]) with the same objective as ours, which lead to the Volvo patent [6].

1.1 Identifying driving context

The objective of this thesis is to classify driving context. To succeed with this, we must first define what we mean by "driving context". As mentioned above, a vehicle system is always operating under a set of constraints that it must adapt to. These constraints arise from outer factors; both static factors, such as the road infrastructure, and dynamic factors, such as presence of other vehicles (traffic density), road construction work, and weather conditions.

In this thesis, however, we will only focus on the static constraints that arise from the road infrastructure. With road infrastructure we mean the layout of the road, i.e. curvature, number of lanes, etc. We also include road objects, such as traffic lights, stop signs, and speed restrictions in this concept. In other words, all static artifacts part of the road network that affects how a vehicle system can be maneuvered. Henceforth, it is this set of road infrastructure constraints we mean when referring to the *driving context*.

Driving context can be defined in fine-grained terms of road objects and obstacles such as red lights, crossings, and curvature degree. This, we refer to as *local* driving context. It can also be described in wider definitions, composed by a characteristic set of local driving context objects. This, we refer to as *global* driving context. For instance, the global driving context class *City* could be defined as an area where sharp turns and crossings are frequent, while the class *Highway* could be defined as an area where turns and crossings are infrequent.

1.2 Previous work

There are different solutions for inferring driving contexts. One solution is to attach a GPS to the vehicle system and fetch driving context information for the system's location from a driving context database. While this solution has some advantages, such as the possibility of predicting upcoming driving context, it also has some significant drawbacks, including the high cost of creating and maintaining the map database. Such a database can become outdated if it is not continuously updated.

An alternative approach, used for instance in [5] and [2], is to have the vehicle system collect vehicle data (speed, brake pedal usage, steering wheel angle, etc.), and infer the driving context from this data. For instance, low speed and frequent use of steering wheel could indicate driving in a city, while high speed and infrequent use of the steering wheel could indicate driving on the highway. This approach has the potential of being cheaper than the GPS approach, since there is no need to create and maintain a driving context database. Additionally, such a method would never produce outdated driving context, since evaluation is done on vehicle data parameters collected in real-time.

The most straightforward way to deduce driving context from vehicle data is probably to state a set of rules, e.g. "*if the speed is higher than x, the current driving context is highway*", etc. However, a drawback with this approach is that it is not scalable: the larger set of vehicle data parameters and driving context classes, the more complicated it becomes to formulate such rules. Another way to infer driving context from vehicle data is to use a fuzzy set approach, such as the one in [16].

Furthermore, in previous studies there have been attempts to infer driving context using statistical pattern recognition frameworks ([5, 2]). A statistical pattern recognition framework is created by training a pattern recognition model on a data set to find patterns that distinguishes classes from each other [1]. In our case, the data set consists of vehicle data and the classes are the different driving contexts. Thus, we would train our framework on a set of vehicle data (x)-driving context (y) records:

$$(x_i, y_i), i \in 1, ..., n \tag{1.1}$$

By doing this, we could find an approximation of the function f mapping vehicle data X to driving context Y:

$$f : X \rightarrow Y \tag{1.2}$$

In [5], such a framework is implemented by means of a feedforward *neural network* [1]. It has been shown that certain kinds of neural network models approximate Bayesian posterior probabilities [22]. Thus, such a framework could output that it is 100 percent certain that the class of a given input is A, or that it is 51 percent certain that the class is A and 49 percent certain the class is B. Such confidence information could be very useful when post-processing the output of the framework, for instance in applications performing critical tasks.

However, it can be argued that the neural network model is a rather difficult to interpret, "black-box" model (see discussion of this in Section 4.6). Other pattern recognition models, such as for instance the Logistic Regression model ([17]), are more interpretable, and therefore easier to work with.

1.3 This project

The goal of this thesis is to implement a pattern recognition framework that can classify driving context classes based on vehicle data. We will train the framework on a large data set of *naturalistic vehicle data* (ND). ND is data collected in a natural driving setting, i.e. the driver logging the data is not involved in a specific study, but is performing some everyday driving task ([28]). This differs from previous studies ([5, 2]), where vehicle data was collected with the specific studies in mind. The disadvantage with data which is not naturalistic is that the driver's awareness of participating in a study might affect his/her driving behavior, making the data unrealistic. ND does not suffer from this problem.

The vehicle data has been annotated with map attributes (representing driving contexts) from a map database. The annotation has been done automatically by matching the GPS coordinates of the records in the vehicle data set and the map database. This approach enables creating a large scale data set. In previous studies ([5, 2]) the data sets have been created by manually labeling vehicle data, which is a time consuming, work intensive task, limiting the size of the data sets. In order to see if driving context can be represented by map attributes from a map database, we do an inventory of the map database and evaluate the map attributes and their suitability in representing driving contexts.

The pattern recognition framework will be evaluated based on its prediction accuracy. Additionally, it should be possible to interpret the framework's output as probabilities representing how confident the framework is on its classification. For instance, the framework should be able to output that it is 100 percent certain that the driving context is highway, or 51 percent certain that the driving context is highway and 49 percent certain that the driving context is city. With these requirements in mind, we review four pattern recognition models of different complexity and compare them against each other. The models are (1) a simple Baseline model, (2) a Logistic Regression model, (3) an SVM model, and (4) a Hidden Markov Model.

1.3.1 Scope

When defining driving context (see Section 1.1), we mentioned a set of factors that affects measured vehicle data: road infrastructure, traffic density, weather condition, etc. These factors could be divided into static constraints (road infrastructure) and dynamic constraints (traffic density, weather conditions). In this thesis we focus solely on the static constraints. The main reason for this is that we have chosen a supervised learning approach, and our data set does not contain labels for dynamic constraints. See further discussion on this choice in the Section 7.3.

Furthermore, as mentioned above, driving contexts can be defined in fine-grained terms such as red-light, crossing, and curvature (local driving context), as well as in wider driving context classes such as *Highway* or *City* (global driving context). The framework that will be the product of this thesis, however, will only output global driving context. We will not attempt to classify fine-grained driving context entities. Specifically, we will focus on two kinds of global driving context classes: (1) whether a road segment is located inside the borders of a city, and (2) the functional class of a road segment. More info these classes is presented in Chapter 2.

We use a selected set of vehicle data signals to classify driving context (see Chapter 3). However, one signal which is generally logged by vehicles that we have not included in this set is the GPS signal, specifying the location of the vehicle. This signal could be used in addition to the vehicle data signals we have selected. For instance, since vehicles repeatedly drive on the same roads in the same locations, previous driving context classifications at a certain location could be used in the classification. Such solutions has been outside the scope of this thesis. However, we elaborate on the possibilities of such a solution further in Section 7.4, when discussing possible improvements on the framework produced in this thesis.

Moreover, the work in this thesis will focus on creating a classification framework with high accuracy. We will not consider other performance measurements, such as how long time the classification takes and what software and hardware resources are required to do the computations. While such factors are important to consider in some settings, they are not significant in the vehicle system context we are working in. Once a model is trained (which will be done offline, i.e. not on-board the vehicle system itself), the computational resources required to produce the classification output are very low. Thus, the resources supported by our vehicle systems are more than sufficient.

1.3.2 Report outline

This report is outlined as follows. In Chapter 2 we present the data set we are using as a base for training our classification models, and describe the different driving context classes which is the output of our framework. Chapter 3 then describe how we selected the models' input features, and Chapter 4 describes four different pattern recognition models, comparing their advantages and disadvantages with respect to the task of this thesis. After this, in Chapter 5 we describe the process of selecting the best hyper-parameter values for our models. Then, Chapter 6 presents our classification results, and finally, Chapter 7 discuss these results and concludes the study.

Chapter 2

Data

The data set plays key role in any pattern recognition problem. Size and quality of the data are significant factors in the performance of the classifier. Below, we describe the data used in this thesis: its content, origin, size, etc. We also describe how we selected a subset from an original, very large data set, on which we train our classifiers.

Moreover, this thesis investigates the possibility of representing driving context with attributes from a map database. Making an inventory of the map database and evaluating whether the different attributes are suitable to represent driving context concepts constitutes a significant part of the work in this thesis. In this chapter, we describe this task.

2.1 Data set

The objective of this thesis is to map vectors of vehicle data, such as speed, steering wheel angle, and brake pedal usage, to a driving context. We use a supervised learning approach to build a pattern recognition model that realizes this goal. Supervised learning requires labeled data, and hence, to achieve our goal, we need a data set containing vehicle data-to-driving context mappings.

The vehicle data that we use come from the *Eurofot* (European Field Operational Test) data set [8]. We use a subset of Eurofot that contains data collected by 30 vehicles operating in real traffic in the UK and the Netherlands with surrounding countries. Eurofot contains so called *Naturalistic data* (ND), i.e. data collected in a natural driving setting, where the driver logging the data is not involved in a specific study, but is performing some everyday driving task ([28]). Previous studies ([5, 2]) classifying driving context have used data which was logged with the specific studies in mind, which has the disadvantage that the behavior of the driver logging the data might be affected by the fact that he/she is aware that he/she is participating in a study, i.e. the data might exhibit unrealistic patterns. Eurofot, containing ND, does not suffer from this problem.

The data in Eurofot is organized in trips; each trip contains data collected by one vehicle while traveling between two locations, sampled at a rate of $10Hz$ (see example of data from a trip in Figure 2.1). Our Eurofot subset contains 80768 trips, which sums up to 59250 hours of vehicle data. The size of the data is larger data sets used in previous studies. For instance, in [5] a data set consisting of 11.5 hours of driving data was used.

Vehicle data in the Eurofot data set is recorded from the truck's Control Area

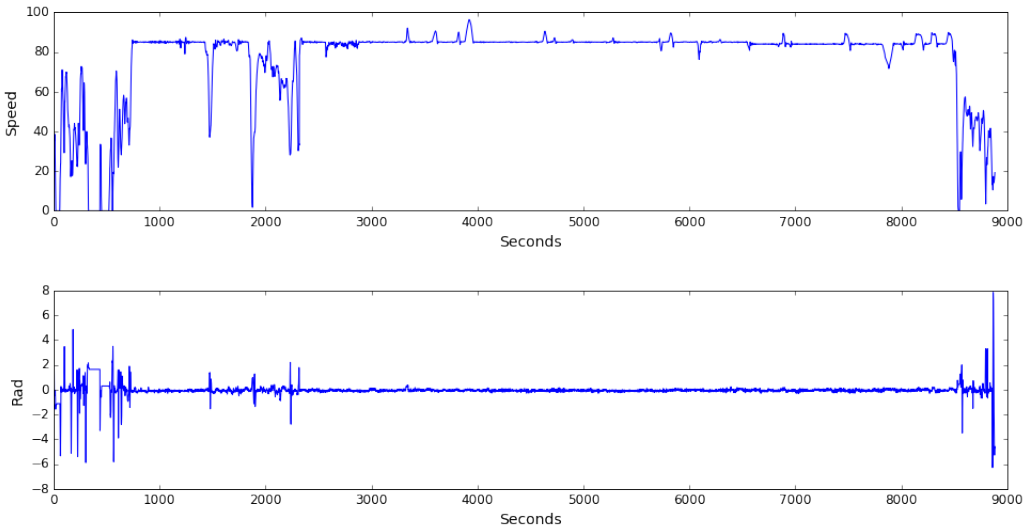


Figure 2.1: Vehicle data from the Eurofot data set. The Speed (top) and Steering wheel angle (bottom) measurements collected by a truck traveling between two European cities

Network (CAN) bus [14]. The CAN bus facilitates communication between micro controllers and other devices in the vehicle. Consequently, many different kinds of data can be collected by tapping the bus. Eurofot contains a plethora of vehicle data parameters, including vehicle speed, steering wheel angle, director indicator activation, and more.

Each record of vehicle data in the data set is annotated with a set of *map attributes*. We let the map attributes represent driving contexts, and use them as labels when training our classifier. The attributes come from a map database provided by the company HERE ([10]). The annotation has been done automatically by matching records from Eurofot with records from the map database using the records' GPS coordinates. This approach has enabled the use of a large data set. Previous studies ([5, 2]) have annotated their data sets manually, which is a time consuming task, limiting the size of the data set.

To investigate what attributes the map database included and evaluate their suitability with regards to representing driving context concepts, an inventory was made. The map attributes that was found were divided into two categories: (1) local driving context attributes, and (2) global driving context attributes. Local driving context were the attributes defined in fine-grained terms of road objects and obstacles such as red lights, crossings, and curvature degree. Global driving context were attributes with wider definitions composed by a characteristic set of local driving context objects. These attributes are presented in Table 2.1 and Table 2.2 respectively.

2.2 Target classes

In this thesis we are focusing on classifying global driving context. Local context recognition has been attempted in previous studies ([2]), but is outside of the scope of this project. From the inventory of the map database we found two categories of global driving context classes: InCity and Functional Class (Table 2.2). These are

Table 2.1: Local driving context attributes in the data set.

Context	**Comment**
Slope	
Curve Radius	
Number of lanes	
One-way restriction	
Toll booth	
Divided	Specifies if the road's lanes are divided.
Ramp	
Roundabout	
STF	Special traffic figure (such as roundabouts with crossings inside)
Tunnel	
Speed restriction	
Crossing	
Stop sign	
Traffic light	
Yield sign	
Parking	

Table 2.2: Global driving context attributes in the data set.

Context	**Comment**
Functional Class	More info in section 2.2
InCity	More info in section 2.2

described in detail below.

Table 2.3: Some functional classes and the driving context constraints they are characterized by ([19]).

Functional Classification	**Access Points**	**Speed Limit**	**Number of travel lanes**
Highway	Few	Highest	More
Collector	Medium	Medium	Medium
Local	Many	Lowest	Fewer

Functional classification is a scheme that divides road segments into classes based on an access-versus-mobility scale. Houses, workplaces, stores, schools, recreational areas and other destinations attract trips which involve movement through the road network. To facilitate such trips in a seamless and safe way, the road network is built such that some roads provide a high level of access (e.g. the road that leads up to the driveway of a house), while other roads provide high level of mobility (e.g. a controlled access highway with multiple, divided lanes). Clearly, roads with high level of access has lower level of mobility, and vice versa. A schema of the functional class hierarchy is presented in Figure 2.2. It is clear that different Functional Classes

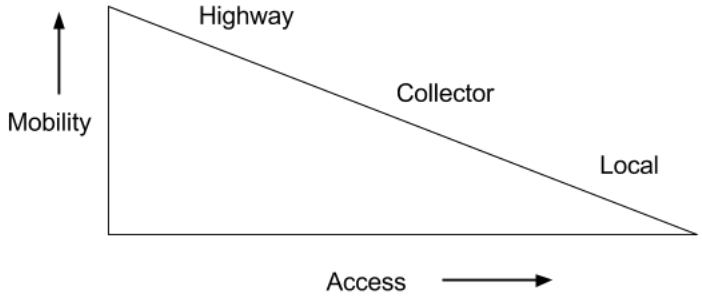


Figure 2.2: Functional classification organizes road segments on a access-versus-mobility scale.

describes different driving contexts. A road segment with low level of mobility will have frequent occurrences of speed limits, crossings, sharp turns, and other driving context constraints, while a road with high level of mobility will have few occurrences of such artifacts. Table 2.3 lists some functional classes and the driving context constraints the characterize them ([19]).

The exact definition and nomenclature of the Functional Classification scheme differ slightly between countries (see for instance [19] for the U.S. Department of Transportation's definitions). However, the main mobility-versus-access concepts described above are universal. In our data set, roads are organized on a scale of five different functional classes. In this thesis, however, we have chosen to merge the first two of these classes, and the third and fourth of the classes, resulting in a set of three different functional classes. The names we have chosen for these classes are inspired by the naming scheme used in the US Department of Transportation's Functional Class definitions. The classes are enumerated below. Moreover, Figure 2.8 on page 14 provides some visual examples of roads with different functional classes.

- **Highway** – High level of mobility, low level of access.
- **Collector** – Medium level of mobility, medium level of access.
- **Local** – Low level of mobility, high level of access.

Furthermore, the map attribute *InCity* specifies if a vehicle is located inside the borders of a city. Clearly, driving in a city introduces constraints that are not present when driving outside a city. For instance, red lights, roundabouts, crossings, and other obstacles will be much more frequent inside a city's borders than outside. Figure 2.3 presents an example of roads that are part of the InCity class.

2.3 Reducing the data set

Our data set is huge. Not only does it contain a large number of records; it also contains a large number of trips from various traffic contexts and countries. The large amount of records enables us to utilize complex pattern recognition models that can describe elaborate classification hypotheses. Also, the fact that the data is collected from many different locations and countries means that it captures many different

Figure 2.3: An example of road segments labeled with the driving context class InCity. Picture taken from Google Earth.

traffic situations. This enables us to build a very general framework. Compare for instance with the framework in [5], where all data is collected in the area around the town of Gothenburg. Such a framework would in theory only work in areas that are similar to Gothenburg. A framework built on our data set, on the other hand, has the possibility to be used in a more general context.

Clearly, all data in our data set could not be used when building a model. This would introduce practical difficulties regarding CPU and memory consumption that is outside the scope of this thesis. Furthermore, in practice we do not need all the data in the original data set to build a high performing classification framework. What is required from the data is that it represents the underlying patterns of all driving context classes in the framework's output. Much of the data in the original data set is, however, simply repetition of the same patterns. Consequently, to decrease the size of our data set, we used the methods described below.

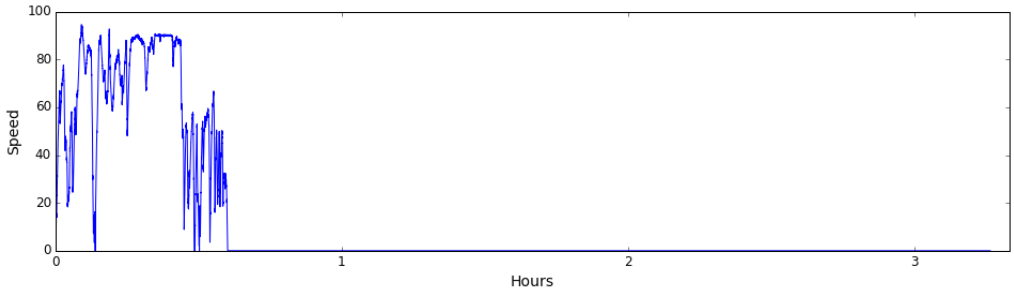


Figure 2.4: The vehicle speed value during a trip. As we can see, the speed recordings becomes "inactive" after a while, and stays that way for hours. Trips exhibiting these patterns were filtered out.

To start with, some of the data from the original data set contained obviously faulty information and could therefore be filtered out. Specifically, it contained trips were the vehicle parameters were "inactive" over long periods of time. For example, the vehicle speed value was 0, steering wheel angle 0, etc., for periods ranging in

the order of hours (see Figure 2.4 for an example of such a trip). We assume such data could be the result of a truck stopping for a break, not turning of the ignition, which would leave the vehicle's data logger on. Additionally, some trips were very short. For programmatic/practical reasons we wanted to avoid such data. Hence, all trips from the original data set shorter than an hour were filtered out.

Next, after applying the filtering steps described above, we wanted to create two data sets: one for training a model predicting the InCity-attribute, and one for training a model predicting the Functional Class. Thus, from the original data set, we selected two subsets of 120 trips each. The trips were selected such that all classes were well represented with a large number of records. Much information from each class enables the model to learn the patterns of the different classes well.

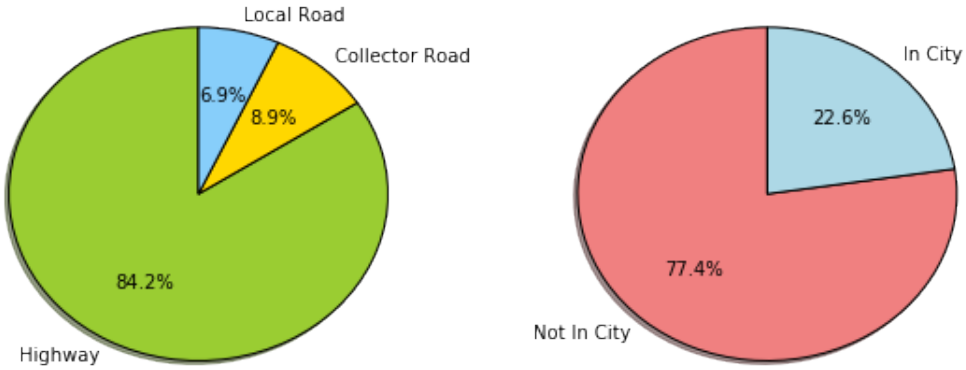


Figure 2.5: The class membership distribution in the Eurofot data set.

Further, we wanted the distributions of classes in the data sets to mirror the "real" distributions. For instance, trucks operate a much larger portion of time on highways than inside a city, and accordingly, our data set should contain more Highway records than city records. This is important since some pattern recognition models take such information into account (e.g. when calculating the transition probabilities in the Hidden Markov Model, see Section 4.4). This requirement - that the class membership distribution in the data sets should mirror the real distributions - was also one of the main reasons why we needed to create one data set for each classification problem (the fact that the class distribution of functional classes in a data set is correct does not mean that the class distribution of InCity classes is correct). We estimated the real portion spent in each driving context class by calculating these portions in the full Eurofot data set. These numbers are presented in Figure 2.5.

Finally, one reason that we have so much data is that it is sampled at a fairly high rate ($10Hz$). For our features (see Chapter 3), we don't need that level of data granularity to represent the patterns that we are interested in. For instance, in the case of the vehicle speed feature, we are not interested in the speed changes over the last tenth of a second, but rather speed changes over some longer period of time. In other words, many records are redundant and could be filtered out. Therefore, we decided to downsample the data to a rate of $1Hz$. This was done by aggregating every group of 10 records to a single record having the mean value of the aggregated 10 records (see source code in Section A.4 in Appendix A). Figure 2.6 shows that

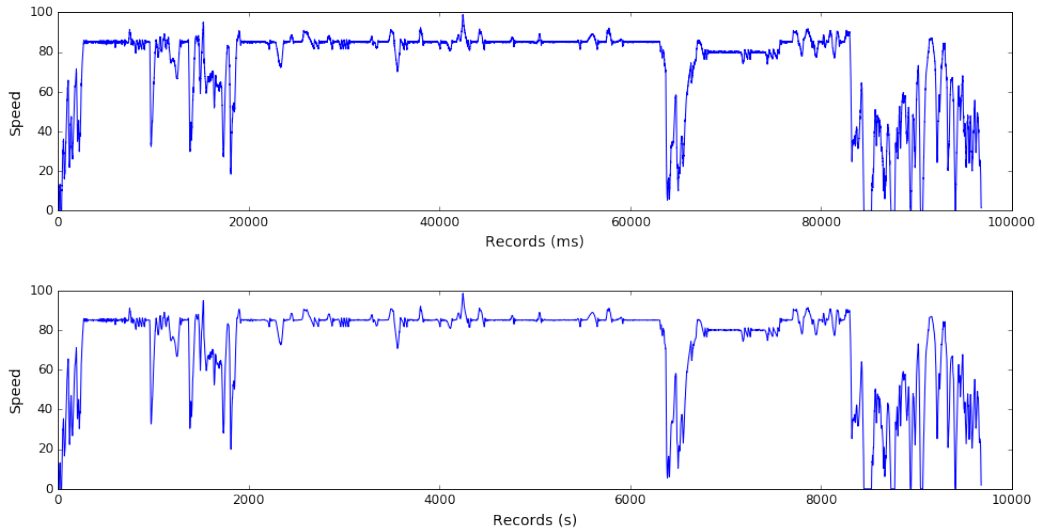


Figure 2.6: No significant piece of information is lost when reducing sample rate from $10Hz$ (top) to $1Hz$ (bottom).

reducing the sample rate to $1Hz$ does not remove any significant information from the speed feature.

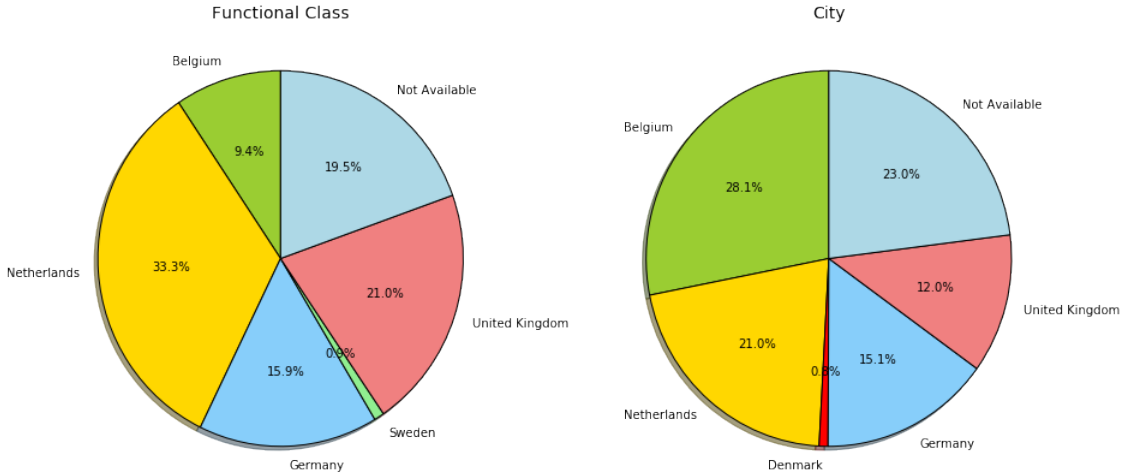


Figure 2.7: The distribution of country membership in the two data sets. Some records where not labeled with a country. These records are included in the "Not Available" category.

The resulting data sets contained 2565 hours (InCity) and 3250 hours (Functional Class) of data respectively. The data sets included records recorded in Belgium, the Netherlands, Germany, Denmark, Sweden, and the UK. The distribution between these countries is presented in Figure 2.7). The variety of countries in our data set ensures our framework will be very general.

2.4 Training and test set

To avoid the problem of overfitting, where the pattern recognition model performs well predicting the data set it was trained on, but performs poorly predicting new,

previously unseen situations, we created a training set and a test set. We train our model on the training set, and then test how well it generalizes on the test set. It is vital that the data in the training and test set are independent of each other. Simply moving a random set of records from an original set to a test set would not be sufficient. This, since our data set is sequential and consecutive records are highly dependent on each other, and thus such a divide would create a training and a test set that would be very similar. Consequently, such a training/test set divide would not help against overfitting. Instead, we divided test and training set by trips: one third of the trips (randomly chosen) was included in the test set and the remaining two thirds in the training set. This approach ensures independence between the data in the two sets.

(a) Highway

(b) Collector Road

(c) Local road

Figure 2.8: Some examples of road segments of different functional classes. Pictures taken from Google Earth.

Chapter 3

Selecting the features

In the previous chapter, we described the data set used in this thesis, and the vehicle data signals it contains. From all these signals, we selected a subset of signals to use as input to our pattern recognition framework. We call these selected signals our *features*. With the exception of one of the models (the HMM, see Section 4.4 and 4.5), we did not, however, use the raw vehicle data signals as features. Instead, the features were created by applying functions to the raw vehicle data signals. In this chapter, we describe the feature creation and selection process conducted in this thesis. We describe how we chose the initial set of vehicle data signals, and how we applied functions to these signals to create the final features.

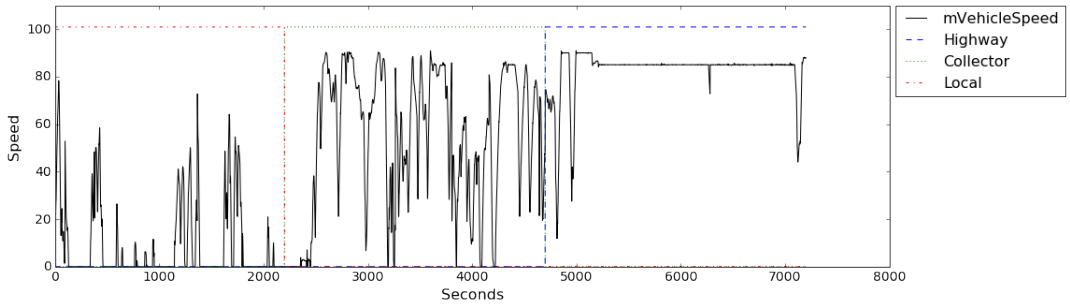


Figure 3.1: Speed of a vehicle during a trip between two European cities. The records are grouped by functional class, and then sorted chronologically in each group. We can see that, in general, the speed is lower and the variation of the speed is higher for the Local functional class, the other way around for the Highway functional class, and somewhere in between for the Collector class.

For a pattern recognition model to perform well, good features are vital. Good features distinguish driving context classes from each other, i.e. there is a strong dependency between the feature and the driving context class. To select a good set of features for our patter recognition framework, as a first step, we selected an initial set of seven vehicle data parameters in the data set from which we would create our features. The set was the same as the one that was used in a previous study classifying driving context ([5]), and included the vehicle data parameters *vehicle speed*, *steering wheel angle*, *acceleration pedal position*, *gear*, *direction indicator*, *engine speed* and *brake pedal position*. All these parameters are intuitively dependent on the current driving context. For instance, frequency of crossings, stop lights, etc., affects the vehicle speed, the brake pedal position, and frequency and degree of curvature affects the steering wheel angle. Figure 3.1 visualizes the dependency

between one of the features, the vehicle speed, and the driving context classes from the functional class problem.

3.1 Features for sequential data

Driving context data is sequential, that is, values of consecutive records are highly dependent on each other. For instance, since (global) driving context classes changes in the order of minutes or hours, if our data is sampled at $1Hz$, the probability that two consecutive records belong to the same class is very high. Thus, when classifying a record x_0 recorded at time t_0, it is good if we use features with info on vehicle data values from a period surrounding t_0. One way to achieve this is to define a time window w including the records surrounding x_0, and extract some aspect of these records. In this thesis we are using three such aspects: the *mean* (μ), the *variance* (σ^2) and *maximum* (max).

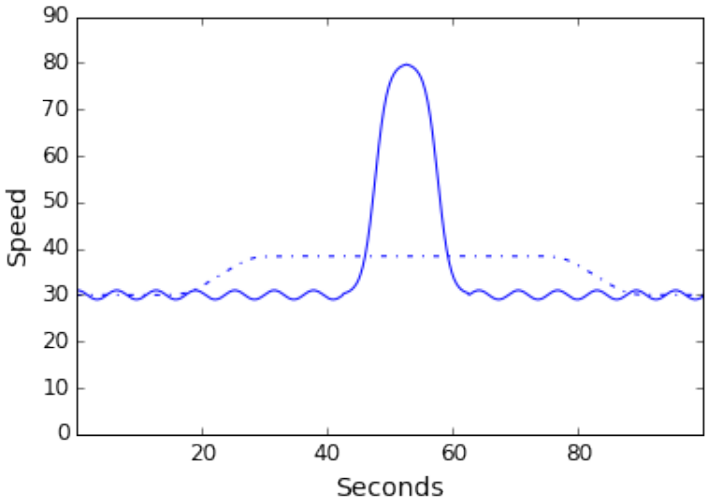


Figure 3.2: The mean function (μ) can be used to remove short term feature variations. Here the full blue line is the vehicle speed and the dotted line is the rolling mean of the speed.

The *mean* (μ) is defined as follows (N denotes the number of records we apply the function on):

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{3.1}$$

The rationale behind using the mean as a feature aspect can be described with the following example. Consider two driving context classes: *Highway*, which is characterized by high speeds, and *City*, which is characterized by low speeds. Changes between these contexts occurs rarely; at the least in the order of minutes. Now, a vehicle is driving at a low speed in class City. During a short period of time, in the order of seconds, the vehicle accelerates to a high speed, and then returns to a low speed. Since we are dealing with contexts that do not change frequently, we do not want the acceleration maneuver of the vehicle to be classified as Highway. We can filter such short time period changes out by classifying context based on the mean of

the speed during some time window, rather than the momentous speed (see Figure 3.2).

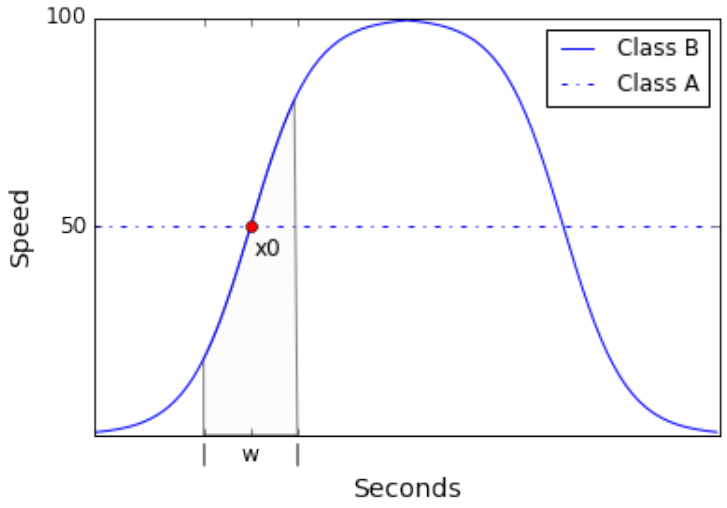


Figure 3.3: In this case presented in this figure, it will not be sufficient to measure the mean speed over the time window w to decide whether x_0 should belong to driving context class A or B. However, the variance in speed over time window w will be different for class A and B.

The *variance* (σ^2) is defined as follows:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{3.2}$$

The variance is a measurement on how much a parameter varies over some time period. The following example describes why such a measurement could be useful in classifying driving context. Consider again two driving context classes: Class A and Class B. *Class A* has a high frequency of road infrastructure obstacles such as red lights, crossings, etc., but no speed limits. *Class B*, on the other hand, contains few road infrastructure obstacles, but a speed limit of 50 km/h. Class A generates a speed pattern with high variance, while Class B generates a speed patterns with a constant value of 50 km/h. To distinguish if a record x_0 should belong to Class A or Class B, the mean speed feature is not sufficient. Consider for instance the case presented in Figure 3.3. For record x_0 in Figure 3.3, the mean speed feature would not be able to place x_0 in Class A or B. However, the speed variance feature would be able to do so.

The *maximum* aspect was designed exclusively for the steering wheel angle feature, and is defined as the maximum value the feature reached during some time period. The following example explains the patterns this feature is meant to capture. Say that we have a class *City*, where segments of straight road are mixed with sharp, 90 degree turns. Then say that we have a class *Rural road*, which contains segments of road with frequent turns of low curvature degree. It is not certain that neither the mean nor the variance aspects would be able to separate vehicle data between these two classes. However, the maximum aspect would succeed in doing this.

The mean and variance aspects were applied to all the vehicle data signals in the initial subset, with one exception: the variance aspect was not applied to the direction indicator signal since this did not make sense. Furthermore, the maximum aspect was only applied to the steering wheel angle feature. Additionally, before this, some processing was done to the steering wheel angle feature and the direction indicator feature. Firstly, the steering wheel angle feature was originally encoded by positive values for left turning and negative values for right turning. We were not interested in the direction of the turning, and for that reason, the absolute value was applied on the steering wheel angle feature. Secondly, the direction indicator feature was encoded with the value 1 for left indicator and 2 for right indicator. Again, we were not interested in the direction, and therefore all active occurrences of direction indicator were normalized to the value 1.

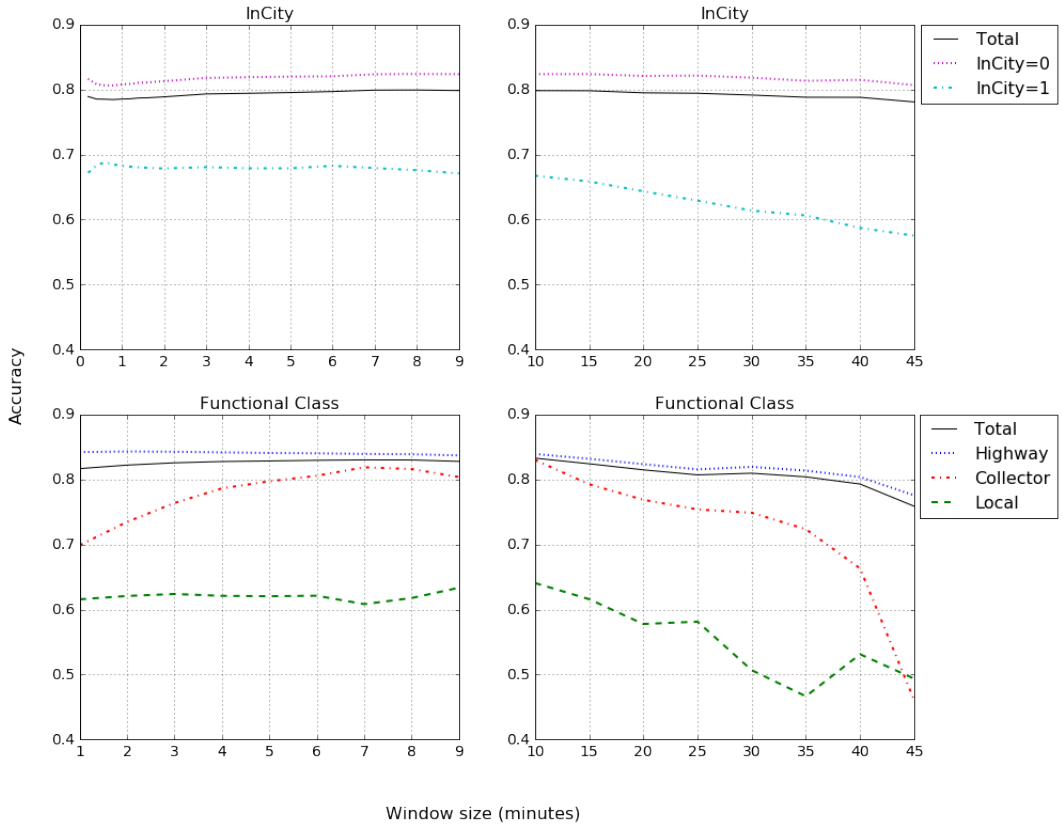


Figure 3.4: A Logistic Regression model was trained to classify the Functional Class and the InCity problems using features with window sizes ranging from 1 to 45 minutes. A window size of 6 minutes yielded a high prediction accuracy for all the classes in both the classification tasks.

To find the optimal value on the size of the time window w used when applying the aspects, the following experiment was conducted. A set of possibly optimal window size values was selected as the range of integer values between 1 and 9 minutes, and every fifth minute between 10 and 45 minutes (i.e. 10, 15, ..., 45 minutes). For all of these window sizes, a model was trained to predict the Functional Class and the InCity problems, and the per-class prediction accuracy was computed using four-fold cross-validation in the training set. The experiment was done both using a Logistic Regression model (see Section 4.2) to find the optimal window size for such a model, and using an SVM (see Section 4.3) to find the optimal window size

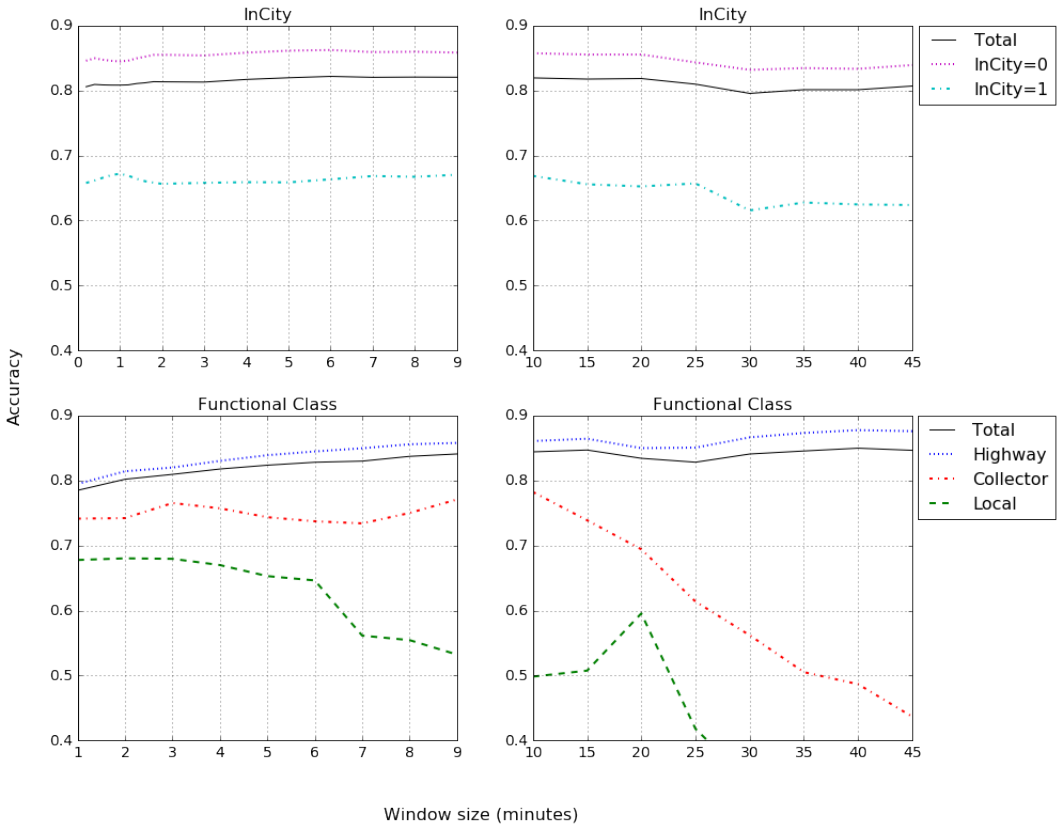


Figure 3.5: An SVM model was trained to classify the Functional Class and the InCity tasks using features with window sizes ranging from 1 to 45 minutes. The window sizes 3 minutes (Functional class task) and 7 minutes (City task) yielded high prediction accuracies for all the classes in the respective problem.

for such a model. The result, presented in Figures 3.4 (Logistic Regression model) and 3.5 (SVM), shows that for the Logistic Regression model, a window size of 6 minutes yields a high prediction accuracy for all the classes in both the problems. Therefore, 6 minutes was the window size used when creating the final features for this model. Furthermore, for the SVM model, a window size of 3 minutes yielded a relatively high prediction accuracy in all the classes for the Functional Class task, and a window size of 7 minutes yielded high accuracies in the City task. Therefore, these were the window sizes chosen for the features for this model.

3.2 Removing redundant features

Clearly, many of the features in our initial feature set exhibits the same behavior, i.e. they display similar patterns given the different classes. For instance, when driving on the highway, the speed and gear vehicle data signals will generally have high values, and the steering wheel angle and direction indicator signals have low values. When driving in the city, it will be the other way around. This indicates there is a possibility that some features do not provide any extra information, and therefore can be removed. To test if this was the case, the following experiment was conducted. A model was trained to predict the InCity and the Functional Class tasks, starting with a feature set containing only the mean vehicle speed. Per class

accuracy was measured using four-fold cross-validation on the training set. Then, one by one, all of the remaining features were added to the feature set, a new model was trained, and accuracy was measured. The feature that yielded the highest *Local Road/InCity=True* class prediction accuracy together with the mean speed feature was kept in the set of included features, and the experiment was run again, now starting with the two selected features. This was repeated until all features had been added. The experiment was done both using a Logistic Regression model to find the redundant features for that model, and an SVM to find that model's redundant features.

Table 3.1: Final feature set for the InCity problem using the Logistic Regression model

Feature	**Aspects**	**Window size (minutes)**
Vehicle speed	μ	6
Acceleration pedal position	μ	6
Gear	μ, σ^2	6
Engine speed	μ	6

Table 3.2: Final feature set for Functional Class problem using the Logistic Regression model

Feature	**Aspects**	**Window size (minutes)**
Vehicle speed	μ	6
Acceleration pedal position	μ, σ^2	6
Gear	μ	6
Engine speed	μ	6
Direction indicator	μ	6
Steering wheel angle	μ	6
Brake pedal position	μ	6

Table 3.3: Final feature set for City problem using the SVM model

Feature	**Aspects**	**Window size (minutes)**
Vehicle speed	μ, σ^2	7
Acceleration pedal position	μ, σ^2	7
Direction indicator	μ	7
Engine speed	μ	7
Brake pedal position	σ^2	7
Steering wheel angle	σ^2	7

The results from this experiment are presented in Figure 3.6 on page 23. For the Functional Class problem using the Logistic Regression model, after adding the steering wheel angle mean feature, Local Road accuracy as well Highway accuracy falls. Therefore, for this problem, we used the set of features that was included

Table 3.4: Final feature set for Functional Class problem using the SVM model

Feature	**Aspects**	**Window size (minutes)**
Vehicle speed	μ, σ^2	3
Acceleration pedal position	μ, σ^2	3
Gear	μ	3
Direction indicator	μ	3
Steering wheel angle	σ^2, max	3

up until then. For the InCity problem using the Logistic Regression model, after adding the acceleration pedal position mean feature, the *InCity=True* accuracy stops increasing. Thus, for this problem, we used the set of features that was included up until then.

Moreover, for the Functional Class problem using the SVM, after adding the acceleration pedal variance feature, the Collector and Local accuracies goes down, and the Highway accuracy stops increasing. Consequently, we used the features added up until then for this task. For the InCity probelm, after adding the steering wheel angle variance feature, the *InCity=True* accuracy starts decreasing. Therefore, for this task, we used the features added up until then.

This left us with the feature sets presented in Tables 3.1 (InCity, Logistic Regression), 3.2 (Functional Class, Logistic Regression), 3.3 (InCity, SVM), and 3.4 (Functional Class, SVM) respectively. Note that for the Baseline model (see Section 4.1), we used that same feature set as for the Logistic Regression model. Furthermore, for the HMM model (see Section 4.4), we did not apply sliding window functions to the vehicle signals to create features, but used the raw vehicle signals. This choice is described more thoroughly in Section 4.5.

3.3 Feature scaling

The values of the features we use ranges over different orders of magnitude. For instance, the mean speed feature ranges over about 0 km/h to 100 km/h, while the steering wheel angle feature ranges between 0 rad and 10 rad. Some of the pattern recognition models we use, that is Logistic Regression and SVM (see Chapter 4), utilize a method called gradient descent to learn the vehicle data patterns of the different driving context classes. However, having features on different scales has some negative consequences for the gradient descent method. First, it makes the learning process of the models very slow, and second, features with large values might dominate the learning process, ignoring the potentially important information in the features on smaller scales.

One popular method for overcoming this problem is *feature scaling*. Feature scaling is a procedure which ensure that all features are on the same scale. The feature scaling method we used in this thesis was so called *standard scaling*, that is, we transform all the values of a feature such that the mean of the values of all the samples in the data set is 0, and the variance is 1. This is done by subtracting the feature mean and dividing by the feature standard deviation (formulated mathematically in Equation 3.3 below).

$$x' = \frac{x - \mu}{\sigma} \tag{3.3}$$

Here, x is the feature value before applying the standard scaling formula and x' is the feature value after applying the formula. Furthermore, μ denotes the mean value (see Equation 3.1) of the feature and σ the feature's standard deviation. The definition of the standard deviation is presented below.

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{3.4}$$

Here, N is the number of records we calculate the standard deviation over, and x_i is one such record.

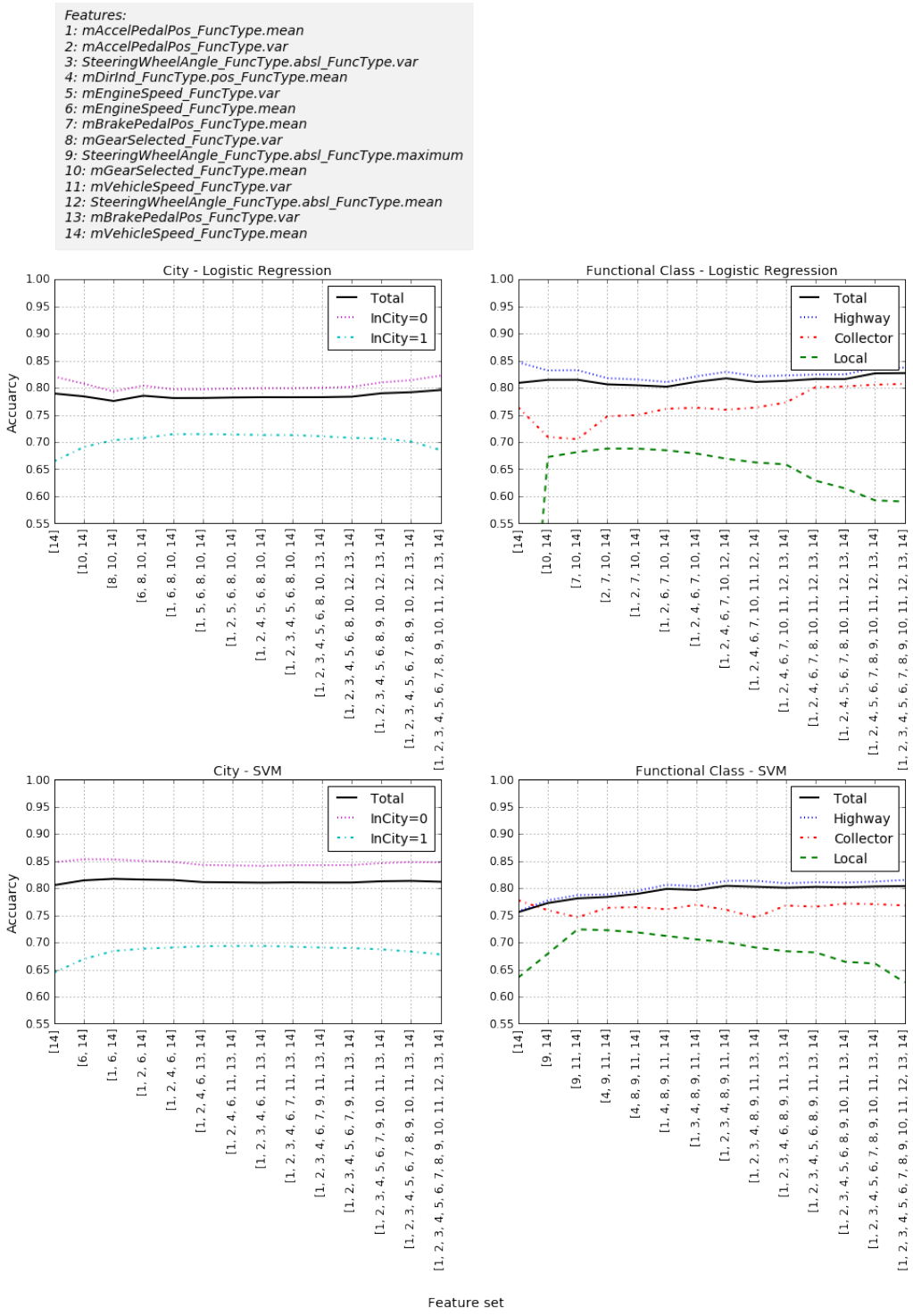


Figure 3.6: Features were added to a feature set one by one, and per class prediction accuracy of a model trained on this feature set was measured. This was done using both a Logistic Regression model and an SVM, on both the classification tasks.

Chapter 4

Models

There are a variety of pattern recognition models with different characteristics. Models differ in complexity (i.e. how elaborately they can describe a classification hypothesis), interpretability (i.e. how straightforward it is to understand the theory behind the model and what the effect will be when tweaking different model parameters), scalability (i.e. how well the model performs as we increase the number of records in the data set), etc. In this chapter, we evaluate a set of different models with respect to the goals of this thesis.

4.1 Baseline model

The first model we considered in this thesis was a very simple baseline model. The purpose of this model was to produce a benchmark measurement that the more complex models could be compared to. Such a comparison yields an understanding of how much value the additional complexity in the other models adds.

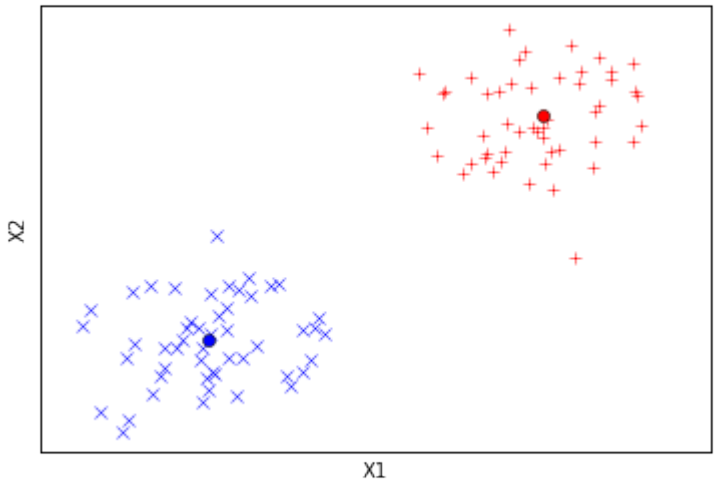


Figure 4.1: The baseline model defines a set of "beacon"-points (circles in the figure), one for each class, and classifies records based on what beacon point they are closest to. In the figure, crosses are classified as one class and pluses as another class. X1 and X2 denotes two different features.

The idea with the baseline model, visualized in Figure 4.1, was to define a set of "beacon"-data points $\mathbf{b_k}$, one for each driving context class k. A record $\mathbf{s}$ would then

be classified according to which of the beacon-data points it was closest to, using the Euclidian distance $d(\mathbf{b_k}, \mathbf{s})$ (see Equation 4.2) as the distance measurement. Accordingly, the mathematical formulation for the model was:

$$k = argmin_k \; d(\mathbf{b_k}, \mathbf{s}) \tag{4.1}$$

The Euclidian distance d is defined as:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{4.2}$$

Here, n is the dimension on the $\mathbf{p}$ and $\mathbf{q}$ vectors, which in our case would be the number of vehicle data features.

The coordinates of the beacon data point for class k were found by calculating the mean over all records belonging to k for each feature. Furthermore, the method for calculating confidence probabilities was done as follows. When the beacon data points were found, we predicted all the records on the training data set. We then calculated how high ratio of the data points in each class was correctly classified. For instance, if 70 % of the points belonging to class Highway was actually classified as Highway, we said the model had a 70 % confidence probability for all future Highway predictions.

The Baseline model was implemented using the Python programming language. The source code can be found in Section A.1 of Appendix A

4.2 Logistic Regression

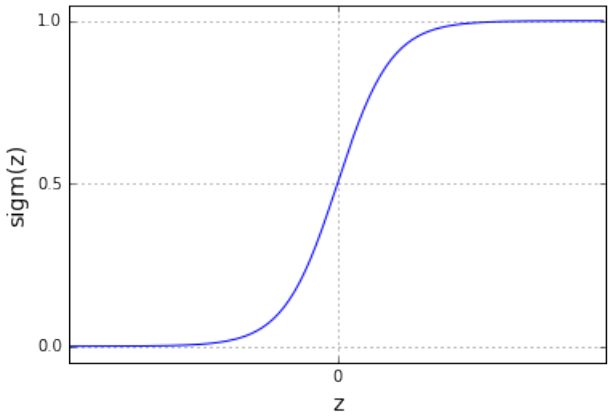


Figure 4.2: The Logistic Regression hypothesis representation $P(Y = 1|\mathbf{X}, \boldsymbol{\theta}) = 1/1 + \exp(-(\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}))$. In the figure, $sigm(z)$ denotes the sigmoid function $sigm(z) = 1/1 + \exp(-z)$, and z denotes the dot product of the record $\mathbf{X}$ and the weights $\boldsymbol{\theta}$ plus the bias term θ_0, $z = \theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}$. We can see that $sigm(z)$ is larger than 0.5 (resulting in classification of class 1) when z is larger than zero.

One of the most widely used pattern recognition models in both industry and academia is the *Logistic Regression* model. The classification hypothesis $h_\theta(\mathbf{X})$ of the Logistic Regression model is defined as follows:

$$h_\theta(\mathbf{X}) = \frac{1}{1 + \exp(-(\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X})} \tag{4.3}$$

The Logistic Regression model uses the *sigmoid* function $sigm(x) = 1/1 + e^{-x}$ to ensure that the output of the model will be a number between 0 and 1. This enables us to interpret the output of the model as a probability. We say that:

$$\begin{aligned} P(Y = 1|\mathbf{X}, \boldsymbol{\theta}) &= \frac{1}{1 + \exp(-(\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X})} \\ P(Y = 0|\mathbf{X}, \boldsymbol{\theta}) &= 1 - P(Y = 1|\mathbf{X}, \boldsymbol{\theta}) \end{aligned} \tag{4.4}$$

In Equation 4.4, Y is the class and $\mathbf{X}$ is a record that we want to classify. Furthermore, $\boldsymbol{\theta}$ is a set of weights that describes the influence of each attribute in $\mathbf{X}$. For instance, if the attribute vehicle speed has a large influence on the probability that a record belongs to class 1, the weight corresponding to vehicle speed should have a large (absolute) value. Furthermore, θ_0 is the so called *intercept* or *bias term*. If the model $P(Y = 1|\mathbf{X}, \boldsymbol{\theta})$ outputs a probability larger than 0.5, the record will be classified as class 1, and if it outputs a probability less than (or equal to) 0.5 it will be classified as class 0. The Logistic Regression hypothesis representation is visualized in Figure 4.2.

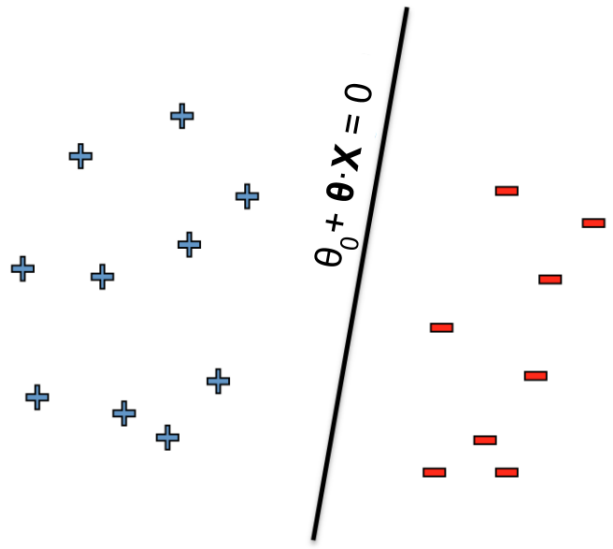


Figure 4.3: Logistic regression can be interpreted as defining a hyperplane $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X} = 0$ separating records into two classes. Here, records classified as Y=1 are marked by plus-signs, while records classified as Y=0 are marked by minus-signs. The figure is originally from [30].

Looking at the visualization in Figure 4.2, it becomes clear that $P(X = 1|\mathbf{X}, \boldsymbol{\theta})$ is greater than 0.5 when $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}$ is greater than zero. In other words, the classifier will predict Y=1 whenever $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}$ is greater than zero, and Y=0 if $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}$ is less than (or equal to) zero. From this follows that $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X} = 0$ can be interpreted as a hyperplane in some n-dimensional space (where n is the number of features), separating records into classes (visualized in Figure 4.3). Records $\mathbf{X}$ yielding a value $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X} > 0$ will be on one side of this hyperplane and records $\mathbf{X}$' yielding a value $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}' \leq 0$ will be on the other side of that hyperplane.

To find the parameters $\boldsymbol{\theta}$ describing the hyperplane, we define the *conditional log likelihood function*, $J(\boldsymbol{\theta})$:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{2}\, pen(\boldsymbol{\theta}) \tag{4.5}$$

Here, m is the number of records in our data set. Furthermore, $cost(h_\theta(x^{(i)}), y^{(i)})$ is a function that defines a penalty for the output our model makes for record $x^{(i)}$, given the true class of the record $y^{(i)}$. We define $cost(h_\theta(x^{(i)}), y^{(i)})$ as follows:

$$cost(h_\theta(x), y) = -[\, y \;\log h_\theta(x) + (1-y)\;\log(1-h_\theta(x))] \tag{4.6}$$

We can see that if the true class of a record is $y = 1$, the closer the value $h_\theta(x)$ produces is to zero, the larger the penalty will be. Also, if the true class of the record is $y = 0$ there will be a high penalty if $h_\theta(x)$ produces a value close to 1. Accordingly, we want the cost to be as small as possible, and find the optimal values of $\boldsymbol{\theta}$ by minimizing the conditional log likelihood function:

$$\boldsymbol{\theta'} = argmin_{\boldsymbol{\theta}}\; J(\boldsymbol{\theta}) \tag{4.7}$$

Furthermore, the $pen(\boldsymbol{\theta})$-term in the conditional likelihood function (Equation 4.5) is a function that puts a penalty on having too high values on the coefficients in $\boldsymbol{\theta}$. In this thesis we use the following penalty function (L2-regularization):

$$pen(\boldsymbol{\theta}) = \sum_{j=1}^{n} \theta_j^2 \tag{4.8}$$

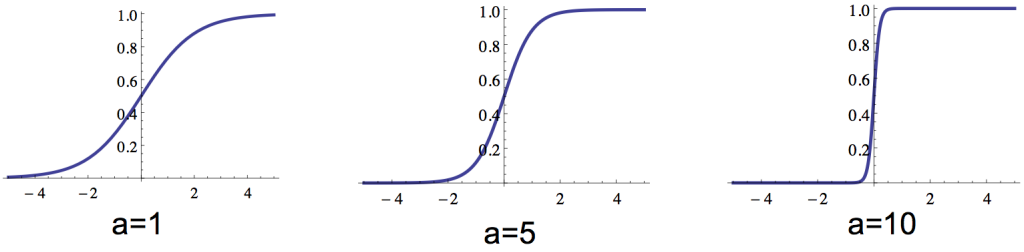


Figure 4.4: Larger (absolute) values on coefficients causes $h_\theta(x) \approx 1$ for more of the records belonging to class 1, and $h_\theta(x) \approx 0$ for more of the records beloning to class 0, resulting in a lower cost of the cost function. In this figure, a symbolizes the sum of the coefficients. The figure is originally from [30].

In Equation 4.8, n is the dimension on the $\boldsymbol{\theta}$ vector. The reason we have the *pen*-term is that the *cost* function favors large (absolute) values on the coefficients in the $\boldsymbol{\theta}$ vector. For instance, according to the cost function, for all the records belonging to class 1, we want $h_\theta(x) \approx 1$. The higher coefficient values we use, the closer will $h_\theta(x)$ be to one for records close to the decision boundary (Figure 4.4 visualizes this). Minimizing the *cost* function thus leads to large values in $\boldsymbol{\theta}$ vector. However, when using to large values on coefficients in $\boldsymbol{\theta}$, there is a risk of focusing too much on the training data, leading to overfitting on the training data set and higher generalization error. The *pen* term prevents this from happening, putting a penalty on high $\boldsymbol{\theta}$ coefficient values. This method, penalizing to much focus on the

training set, is called *regularization*. How much we want the penalty to influence the likelihood function is controlled by the λ parameter. A high value on λ results in lower values on the weights, and a model which is less specialized on predicting the records of the training set.

The conditional log likelihood function is a convex function, which is a very nice property since we do not risk getting stuck in local minima when optimizing it. However, it has no closed form solution, so we find the minima using gradient descent. Specifically, in this thesis we use the *Stochastic Averaged Gradient* (SAG) descent approach [25]. This since we train our model on very large data sets, and the SAG method is fast on large data sets.

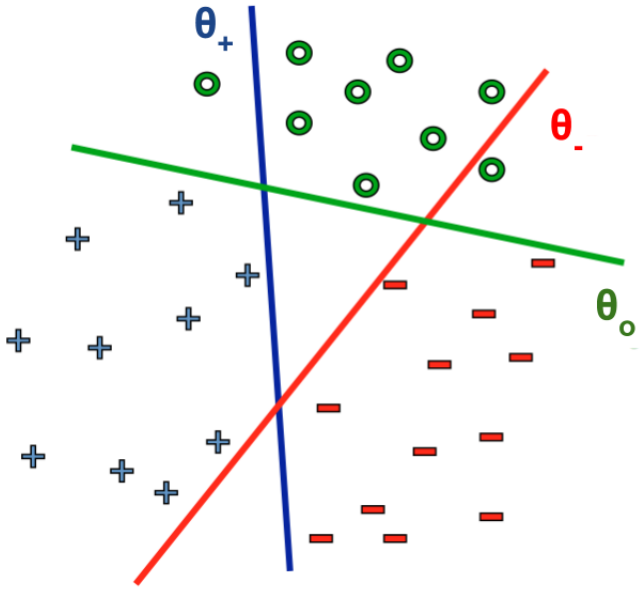


Figure 4.5: In the one-against-all method, we train one classifier ($\boldsymbol{\theta}$) per class. When predicting a record, we calculate the probability for each class using each classification boundary, and chose the class that had the highest probability. The figure is originally from [30].

Logistic Regression can be extended to predicting a set of multiple classes using the one-against-all method. In this method, we train one classification boundary per class, and then combine all of the boundaries to produce our final classifier (see Figure 4.5). When predicting a record, we calculate the probability for the record for each classification boundary, and chose the class that yielded the highest probability:

$$y^* = argmax_y \; P(Y = y|X) \tag{4.9}$$

In both our data sets, there is one class which is more common than the others. In the Functional Class problem, there is a ratio of approximately 11:1:1, with the Highway class occurring about elven times as often as the other classes. Furthermore, in the InCity problem, the *InCity=False* class is the most common with a 7:2 ratio (see Section 2.3). Such class imbalance can lead to a model that produces high classification accuracy for the common class, but low accuracy for the less common

classes. Basically, when training the model, since there are many more samples in the common class, the gain of predicting these samples correctly is favored over the penalty of misclassifying the uncommon classes.

In this thesis, we wanted a model that could predict all classes fairly well. To achieve this, we weighted training records with a weight equal to the inverse of the training set frequency of the class. This results in a higher penalty in misclassifying the uncommon classes, and thus, leads to a model that will be better at predicting these classes. However, prediction accuracy for the common class and overall prediction accuracy might suffer. This is a tradeoff we are willing to make. Furthermore, a consequence of class weighing is that the output of the model $P(Y|\mathbf{X})$ represents the a-posteriori probability assuming that classes are equally common. Prior knowledge of class distributions can be included by correcting the intercept θ_0, for instance using the method suggested by King and Zeng ([15]). However, such measures have been outside the scope of this thesis.

The Logistic Regression model was implemented using the Python programming language and the Scikit-learn library [20] (see full source code in Section A.2 of Appendix A)

4.3 SVM

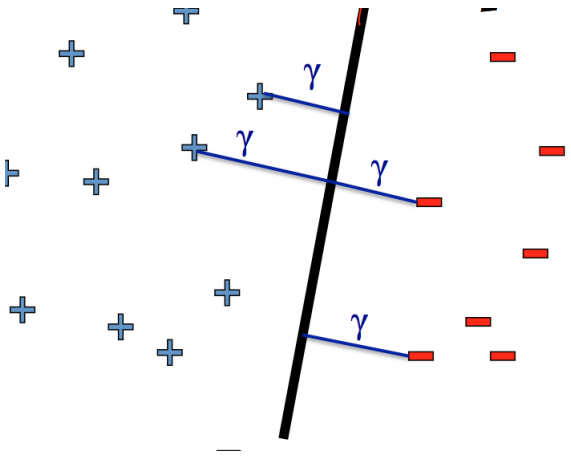


Figure 4.6: An SVM defines a hyperplane which maximizes the (minimum) margin γ to records in the two classes in the data set. The figure is originally from [30].

Another widely used pattern recognition model is the *Support Vector Machine* (SVM). The SVM has some similarities to the Logistic Regression model: it is a binary classifier which defines a hyperplane separating data points into two classes. The hypothesis representation of the SVM $h_\theta(\mathbf{X})$ is defined as:

$$h_\theta(\mathbf{X}) = \theta_0 + \boldsymbol{\theta} \cdot \mathbf{X} \tag{4.10}$$

We classify the record $\mathbf{X}$ as class 1 if $h_\theta(\mathbf{X})$ is greater than zero and as class 0 if $h_\theta(\mathbf{X})$ is less than (or equal to) zero.

The SVM and the Logistic Regression model differ in the way that they use different cost functions to find the optimal hyperplane. The idea with the SVM is to find the hyperplane that maximizes the minimum distance (*margin*, γ) to records of each class (see Figure 4.6). This intuitively minimizes generalization error. It can

be shown that maximizing the margin is done by minimizing the squared norm of the $\boldsymbol{\theta}$ vector ([17]). This is used in the cost function $J(\boldsymbol{\theta})$ for the SVM, called *Hinge loss*, which is defined as follows:

$$J(\boldsymbol{\theta}) = C \sum_{i=1}^{m} [\ y^{(i)} cost_1(h_\theta(\mathbf{X})) + (1 - y^{(i)})\ cost_0(h_\theta(\mathbf{X}))] + \frac{1}{2}\|\boldsymbol{\theta}\|^2 \tag{4.11}$$

Here, m is the number of records in the data set. Furthermore, $cost_0$ and $cost_1$ are two functions that specify the cost of misclassifying a record belonging to class 1 and class 0 respectively. Specifically, they are defined as follows:

$$cost_1(z) = [-z + 1]_+ \tag{4.12}$$

$$cost_0(z) = [z + 1]_+ \tag{4.13}$$

Here, $[z]_+$ is defined as:

$$[z]_+ = max(z, 0) \tag{4.14}$$

As we can see, if a record belongs to class 1, the smaller value we produce for $\theta_0 + \boldsymbol{\theta} \cdot \mathbf{X}$, the larger will the cost be, and the other way around if the record belongs to class 0. The C parameter in the hinge loss function specifies at what level we should take this cost into account. Hence, a high value on C forces the SVM to fit the training set very well, while a low value on C implies higher regularization (the inverse to the λ-parameter in the Logistic Regression model).

Finding the optimal hyperplane is done by minimizing the Hinge loss function:

$$\boldsymbol{\theta'} = argmin_\theta\ J(\boldsymbol{\theta}) \tag{4.15}$$

Just as the conditional log likelihood function of the Logistic Regression model, the Hinge loss function is convex so it has no local minima. It has no closed form solution, and is thus solved using gradient descent.

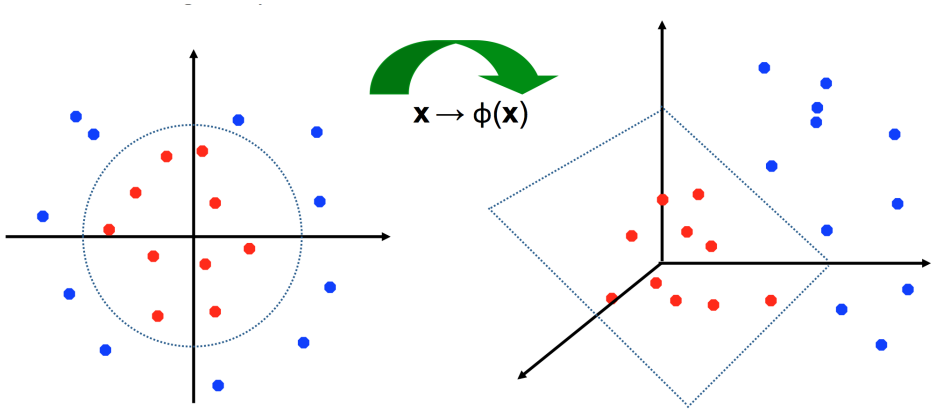


Figure 4.7: Using a kernel function $\Phi(x)$ we can map a feature vector to a higher dimensional feature space. This allows a linear hyperplane to perfectly separate data that was not linearly separable in the original feature space. The figure is originally from [30].

One important characteristic of the SVM is that it can be used to create non-linear hypotheses. That is, we can separate data points with non-linear boundaries. We accomplish this by using so called kernel functions to map features into a higher dimensional space. This lets us separate points with our hyperplane that was not linearly separable in the original feature space (see figure 4.7).

There are a set of different kernel functions that are commonly used. For instance, there is the *Radial basis function* (RBF) kernel (also known as Gaussian kernel), the polynomial function kernel, etc. Which kernel that is best suited for a particular pattern recognition problem depends on the nature of the data, and is usually found by comparing the prediction accuracy for different kernels using cross validation.

SVM is a binary classifier, but can be extended to the multi-class case by using the one-against-all method (same as for Logistic Regression, see Section 4.2).

Given a record, the SVM model outputs the dot-product between that record and the vector orthogonal to the decision boundary hyperplane. This means that points far from the decision boundary will generate a high-valued output, while records close to the hyperplane will generate a low-valued output. Hence, the output can be interpreted as a score on how confident the model is that the record belongs to that class. However, this output score is not a probability. To transform the output to a probability, we can use a method called *Platt scaling* (binary case [21], multi-class case [29]). The idea with Platt scaling is that we fit a Logistic Regression model to the output of the SVM:

$$P(Y = 1|\mathbf{X}) = \frac{1}{1 + \exp(A + B * h_\theta(\mathbf{X}))} \tag{4.16}$$

Here, $h_\theta(\mathbf{X})$ is the output of the SVM. The Logistic Regression model is fit using class labels from the training set the SVM model was trained on. Platt scaling can be used for many pattern recognition models, but as it turns out, it is particularly well suited for SVMs [18].

Just as in the case of the Logistic Regression model, the SVM model is affected by that we have an unbalanced data set. We solved this using the same record weight method that was used in the Logistic Regression model.

The SVM model was implemented using the Python programming language and the Scikit-learn library [20]. Source code can be found in Section A.2 of Appendix A.

4.4 Hidden Markov Model

When dealing with sequential data, one of the most popular models is the *Hidden Markov Model* (HMM). The HMM is simple enough that it can estimate parameters and classes efficiently in a reasonable time frame, but rich enough that it can actually be used for many real world applications. The HMM is defined by the Markov graph presented in Figure 4.8, with the form of what is called a *Trellis* diagram. Here, $z_1, ..., z_n$ are a set of random variables called *Hidden Variables*, and $x_1, ..., x_n$ are a set of random variable called *Observed variables*. The hidden variables represent a state, or in a classification problem, a class. The output of the model is the probabilities for the different classes for each z_t. The model is often interpreted such that z_1 denotes the class at time 1, z_2 the class at time 2, etc. The observed variables

represent some value dependent on the hidden variables. By assigning values to the observed variables we can calculate the probabilities of the hidden variables. When applying the HMM to the task of this thesis, x_t represents the feature vector for record t that we want to classify, and z_t represents the class the model outputs for record t.

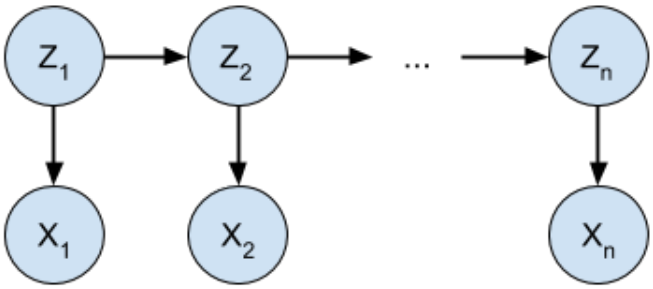


Figure 4.8: An HMM is defined by the Markov graph with the form of the graph in this figure, also known as a Trellis diagram. Here, $z_1, ..., z_n$ represents hidden variables and $x_1, ..., x_n$ observed variables. In the context of this thesis, x_t represents the feature vector for record t that we want to classify, and z_t represents the class the model outputs for record t.

From the graph in Figure 4.8 follows that the joint probability distribution of all the random variables, $p(z_1, ..., z_n, x_1, ..., x_n)$ can be factorized as follows:

$$p(z_1, ..., z_n, x_1, ..., x_n) = p(z_1)p(x_1|z_1)\prod_{k=2}^{n} p(z_k|z_{k-1})p(x_k|z_k) \tag{4.17}$$

Here, n denotes the number of states. Furthermore, Equation 4.17 is often divided in to three terms that parametrizes the HMM:

- The *Transition matrix* $T(i,j) = P(z_{k+1} = j|z_k = i), i, j \in 1, ..., m$. This describes the probability of changing from class j to class i.
- The *Emission probabilities* $\epsilon_i(x) = p(x|z_k = i), i \in 1, ..., m$.This describes the probability of a certain feature vector x given class i.
- The *Initial distribution* $\pi(i) = P(z_1 = i), i \in 1, ..., m$. This describes the probability of the first class in the Markov chain being class i.

Here, m denotes the number of classes. Writing Equation 4.17 in terms of these parameters we get:

$$p(z_1, ..., z_n, x_1, ..., x_n) = \pi(z_1)\epsilon_{z_1}(x_1)\prod_{k=2}^{n} T(k-1, k)\epsilon_{z_k}(x_k) \tag{4.18}$$

To classify a set of consecutive records (observed variables) x, we want to find the most probable sequence of hidden variables z for that sequence of records:

$$z^* = argmax_z\ p(z|x) \tag{4.19}$$

Here, z denotes all Zs $z_{1:n}$ and x all Xs $x_{1:n}$. We solve this using the *Viterbi algorithm* ([9]).

The HMM-model was implemented using the *hmmlearn* library [12]. The source code can be found in Section A.3 of Appendix A.

4.5 Comparing models

As mentioned earlier in this report (see Section 1.3), the criteria we have set for our models are:

1. It should be possible to interpret the model's output in terms of probabilities describing how confident the model is that the predicted class is correct.

2. The classification accuracy, i.e. the number of correctly predicted records on the test set, should be as high as possible.

3. The model should be as interpretable as possible.

In this chapter we have described four models who all fulfil the first requirement of producing output probabilities. However, the probability interpretation of the Baseline model is a bit more primitive than that of the other models. Specifically, it produces the same probability for all records with the same class membership. For example, intuitively, if a record has the speed 150 km/h we can be more certain that it belongs to the highway driving context class than if the record has a speed of 80 km/h. This idea is reflected in the probability output of all the models except from the Baseline model.

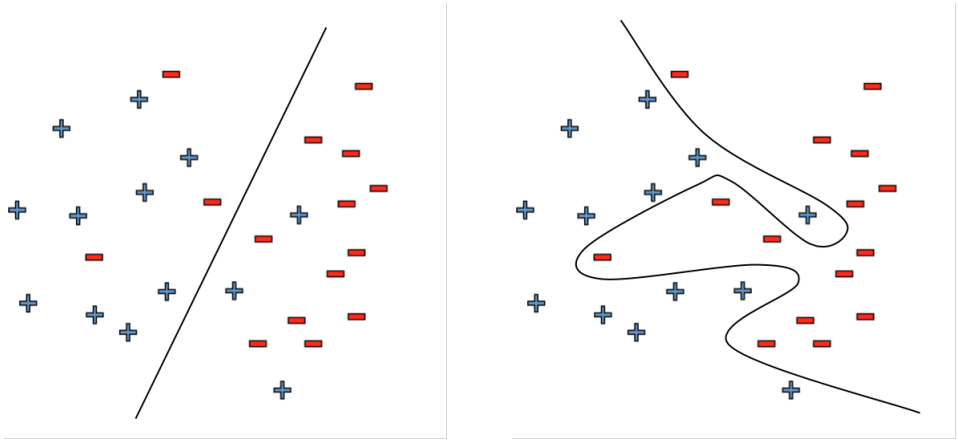
Furthermore, we want to create a model that can achieve a high prediction accuracy. Which model will produce the highest accuracy depends on the nature of our data, and we will see the answer to this in Chapter 6 where results from applying the classification tasks of this thesis to the models is presented. However, what can be concluded from the model review in this chapter is that the models differ in complexity, and therefore have different possibilities of reaching high accuracy predictions.

The Baseline model and the Logistic Regression model, for instance, can only predict linear decision boundaries[1], and thus, if the data is not linearly separable, does not have the ability to reach as high accuracy as the SVM and HMM models, which can describe non-linear decision boundaries. Figure 4.9 visualizes the idea of linear and non-linear decision boundaries.

Moreover, comparing the Logistic Regression model and the Baseline model, we can see that the Logistic Regression model, using the conditional log likelihood function to find a decision boundary, has the ability to find a better decision boundary than the Baseline model. The following example explains why.

Consider a data set A with records from two different classes, and where all records belonging to a class are "equally" distributed around some mean for that class. Such a data set is presented in Figure 4.10a. The Baseline model will generate

[1]For the record, it is possible for a Logistic Regression model to define a non-linear decision boundary by mapping features to a higher dimensional space, for instance using a kernel function. However, this cannot be done in a computationally efficient way as it can in the case of the SVM, and therefore, it is ruled out as an option in this thesis.



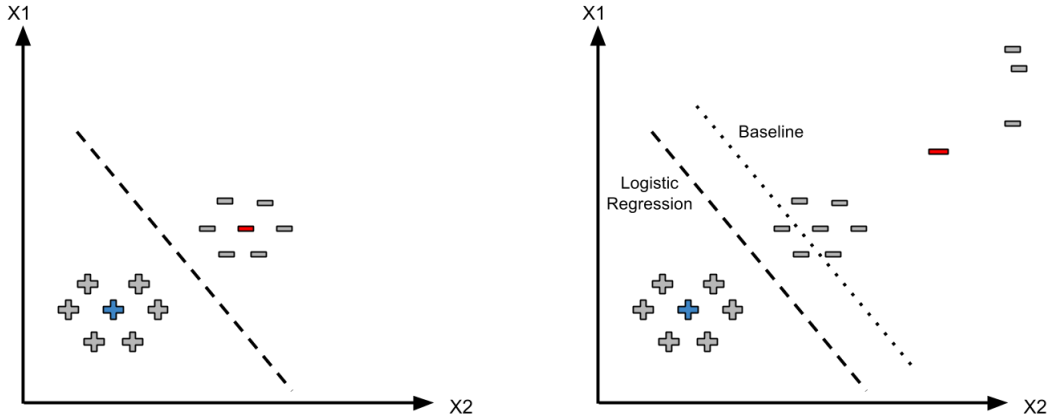
(a) Linear classifier (b) Non-linear classifier

Figure 4.9: A linear classifier (a) such as Logistic Regression cannot predict a data set which is not linearly separable perfectly[1]. A non-linear classifier (b) such as kernelized SVM, however, can.

beacons located in the middle of the data for both classes, and the decision boundary separating records will be placed just between the beacons. In the case of this data set, the Logistic Regression model finds the same decision boundary as the Baseline model, producing the same prediction. Then, consider a data set B, where data is not equally distributed around some mean per class. Instead, the data contain some outlier records. Such a data set is presented in Figure 4.10b. In this scenario, the outliers affect the positioning of the Baseline model's beacons, resulting in a suboptimal placement of the decision boundary. The Logistic Regression however, which calculates the decision boundary by maximizing the conditional log likelihood, finds the optimal decision boundary, achieving a better prediction result than the Baseline model.

When it comes to interpretability, all the models we have presented are rather straight forward. It is easy to interpret the vector of weights $\boldsymbol{\theta}$ specifying the importance of each feature, and to understand what it means to modify the λ and C-parameters and how it affects the model and its prediction. Furthermore, the HMM has a very clean probabilistic definition, and we can see how emission and transition probabilities affect the model. One thing that might stand out slightly is the SVM's kernel function. The SVM's results can differ depending on what kernel function we use, and it is not always clear why one kernel function outperforms another.

One thing that makes the HMM stand out is its ability to directly handle sequential data. The data we use is highly sequential and to create a good classifier this fact has to be taken into account. For the Baseline, Logistic Regression and SVM models, this is handle by creating features that include information from some time window surrounding a record x_0 that we want to predict (see Section 3.1). For instance, we apply a rolling mean on raw vehicle data to stabilize the predictions of the model, filtering out short term changes in vehicle signal values. However, the drawback with this approach is that to achieve an optimal prediction, we want to be able to detect short term variations. Consider for instance the situations presented in Figure 4.11 and 4.12. In Figure 4.11, a vehicle is driving at a slow speed in Class



(a) Data set A. Since records for a class are "equally" distributed around some mean, the Baseline model and the Logistic Regression model produces the same decision boundary.

(b) Data set B. Data is not equally distributed around some mean per class. The Logistic Regression model can find the optimal decision boundary but the baseline model cannot.

Figure 4.10: Two data sets resulting in different predictions for the Baseline model and the Logistic Regression model. Records from one class is denoted with pluses, and records from the other class is denoted with minuses. The blue and red records are the Baseline model beacons for each class. The decision boundary is denoted with a dashed/dotted line.

A, and then rapidly increases its speed changing to Class B. With the mean applied, the rapid change in speed will not be visible to the model, but the speed change will be "spread out", making it harder for the model to correctly predict when the class change takes place. Furthermore, in Figure 4.12, a vehicle is driving at a high speed in Class B, slowing down for a short while changing class to Class A, and then going back up to a high speed and Class B again. When applying the rolling mean on this, the speed change almost disappears, making it difficult for the model to predict the situation correctly.

The HMM, on the other hand, does not use features with information from some time window. It achieves stability in its predictions by the use of the transition probabilities. Since it makes predictions on raw vehicle data values, it has the potential to succeed better than the other models, for instance when predicting the scenarios depicted in Figure 4.11 and 4.12.

4.6 Additional models

In addition to the four aforementioned models, two models were considered but ruled out. Firstly, the *Naive Bayes* model is a popular pattern recognition model which builds on Bayes Theorem. An assumption of conditional independence between features makes computing a probability distribution over a set of classes conditional to a vector of features very fast and scalable. Additionally, the direct connection to Bayes Theorem makes the Naive Bayes model very interpretable. However, an implication of the conditional independence assumption, which in the problem of this thesis is unrealistic (for example, given driving context class City, the vehicle

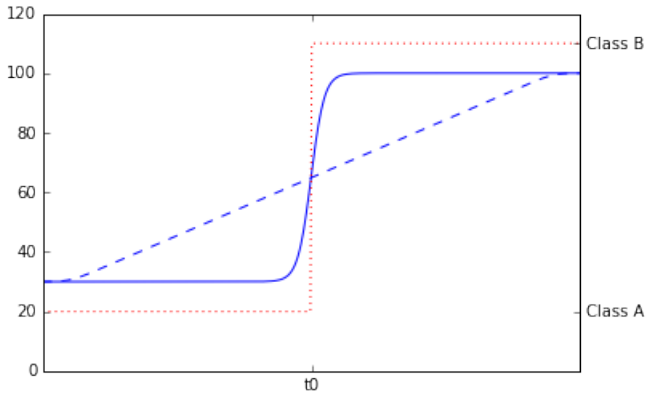


Figure 4.11: A vehicle is driving at a slow speed (full blue line) in Class A (dotted red line), and then rapidly increases its speed changing to Class B. With the mean applied (dashed blue line), the rapid change in speed will not be visible to the model, but the speed change will be "spread out", making it harder for the model to correctly predict when the class change takes place.

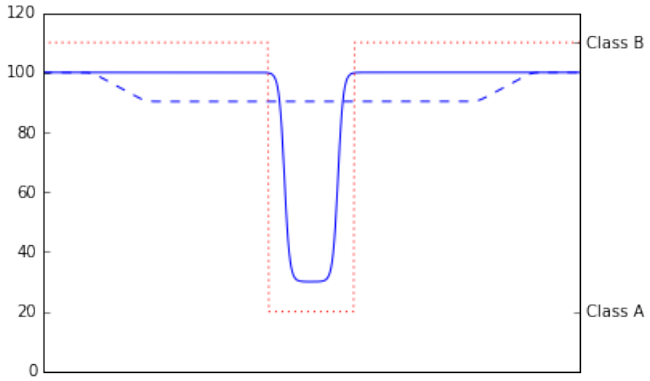


Figure 4.12: A vehicle is driving at a high speed (full blue line) in Class B (dotted red line), slowing down for a short while changing class to Class A, and then going back up to a high speed and Class B again. When applying the rolling mean (dashed blue line), the speed change almost disappears, making it difficult for the model to predict the situation correctly.

speed is *not* independent of the steering wheel angle), is that the Naive Bayes models is not a good estimator, i.e. it is not good at producing prediction probabilities ([18] describes this in more detail). According to the requirement of this thesis saying that the model should be able to produce output probabilities, the Naive Bayes model was ruled out. Furthermore, one of the models we do use, the HMM model, is a generalization of the Naive Bayes model. Since a HMM model can describe any Naive Bayes model, the Naive Bayes model would have been redundant.

Secondly, a *Neural Network* is a model which is widely used for different pattern recognition problems. Specifically, it has been used in previous studies ([5]) to classify driving context. However, it can be argued that neural networks are difficult to interpret. For instance, in contrary to the Logistic Regression and the SVM models, which is defined by a one-dimensional vector of weights ($\boldsymbol{\theta}$), a Neural Network is defined by a network of nodes, each with its associated weights. The interpretation of the weight vector in the Logistic Regression and SVM cases is straightforward;

there is one weight per feature specifying the importance of that feature. However, in the Neural Network case, weights describe the combinations between the nodes in the network and cannot be associated to a specific feature. It is also not straightforward how the design of the structure of the network (number of hidden layers and hidden nodes) should be interpreted and how a specific structure affects the result. Furthermore, the method used to calculate the weights in the Logistic Regression and the SVM models, that is gradient descent on a convex function, is simpler to conceptually grasp than the back propagation algorithm used in the neural network model.

Because of these considerations, we chose to not include a neural network in this thesis. Furthermore, the SVM model we use has the ability to define non-linear classification hypotheses. Therefore, in terms of ability to represent complex classification hypotheses the gains of including a Neural Network as well would not be significant.

Chapter 5

Hyper-parameter selection

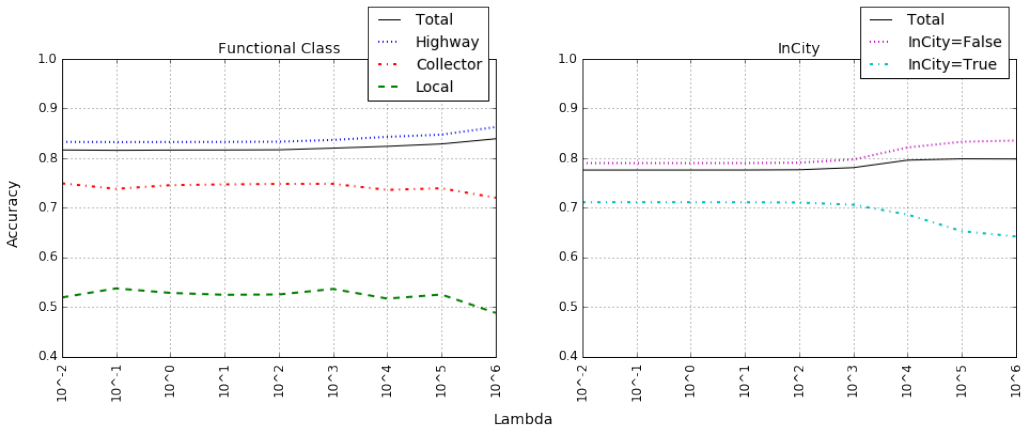


Figure 5.1: λ parameter selection for the Logistic Regression model. For both the classification problems, λ-values of 1000 and below yielded a high prediction accuracy for all the classes.

Two of the models, the Logistic Regression model and the SVM model, are defined by hyper-parameters which affects the results the model produces. For instance, the Logistic Regression model has the λ parameter defining the degree of regularization in the model (see Section 4.2). Optimal hyper-parameters values for each model were found by doing four-fold cross-validation on the training set, and selecting the hyper-parameter value that yielded the highest prediction accuracy. However, selecting hyper-parameter value based on *overall* prediction accuracy was not sufficient. This, since in both the Functional class and the InCity classification problems there is one class which is more common than the others (the Highway class and the *InCity=False* class respectively). In such cases, a model can reach a high overall accuracy by focusing on predicting the common class very well, neglecting the less common classes. For instance, if a model predicted all records as Highway, it would reach an overall accuracy of about 85 % in the Functional class problem, but 0 % accuracy for the Collector Road and Local Road classes. Clearly, such a classifier is not very practical. To avoid training such a classifier, and instead create a classifier with the ability to predict all classes relatively well, we found the optimal hyper-parameter values by evaluating the model's *class specific* accuracies.

5.1 Logistic Regression

The Logistic Regression model has one hyper-parameter, λ, specifying the degree of regularization in the model (see Section 4.2). The optimal value for the λ parameter was found by running cross-validation with λ-values on a logarithmic scale between 0.01 and 10^6 (i.e. 0.01, 0.1, ..., 10^6). This test is presented in Figure 5.1. For both the classification problems, λ-values of 1000 and below yielded a high prediction accuracy for all the classes. Thus, $\lambda = 1000$ was chosen for the final models.

5.2 SVM

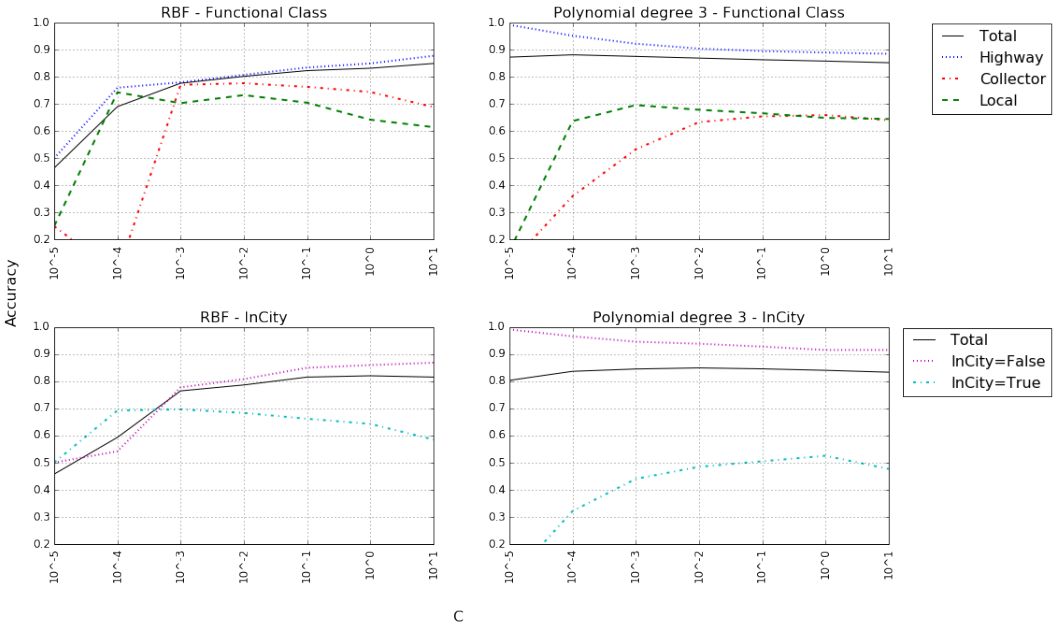


Figure 5.2: Kernel and C-value selection for the SVM model. For both the classification tasks, the RBF-kernel and a C-value of 0.1 yielded a high prediction accuracy for all the classes.

The SVM model has two hyper-parameters: (1) the C parameter specifying the penalty for misclassifying a record, and (2) what kernel function to use (see section 4.3). The optimal values for these parameters was found by testing C values on a logarithmic scale between 10^{-5} and 10, and two different kernel functions: the RBF-function and the polynomial function with degree three. This test is presented in Figure 5.2. For both the classification tasks, the RBF-kernel and a C-value of 0.1 yielded a high prediction accuracy for all the classes. This parameter combination was hence chosen for the final models.

Chapter 6

Results

In this Chapter, we present the results from applying two classification problems on four different models: a simple *Baseline* model, a *Logistic Regression* model, an *SVM*, and a *Hidden Markov Model*. The two classification problems were (1) classifying the InCity-attribute, and (2) classifying the functional class of the road (see Chapter 2 for more info on these driving context classes). The measurement used was the prediction accuracy, i.e. the number of correctly predicted records on the test set. We look at both the overall prediction accuracy, i.e. the ratio of correctly classified records in total, and the per-class prediction accuracy, i.e. the ratio of correctly predicted records belonging to a given class. Furthermore, for each prediction we produced a confusion matrix presenting, for all records belonging to a given class (rows), what class they were predicted as (column).

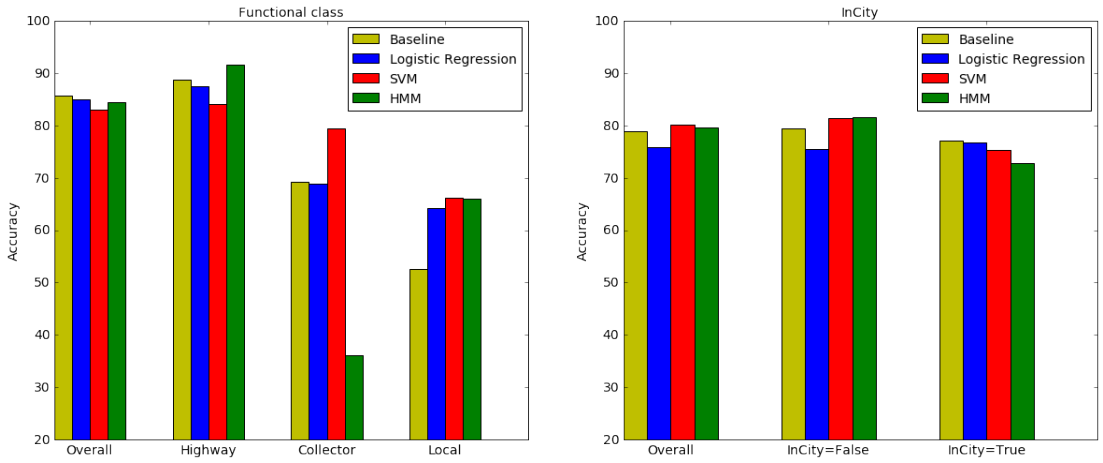


Figure 6.1: Overall and per-class prediction accuracy for all the different models.

The results from evaluating prediction accuracy of each model for both the Functional class and InCity classification problems is presented in Figure 6.1. Confusion matrices can be found in Figure 6.9 on page 46 (Functional Class problem) and Figure 6.10 on page 47 (InCity problem). Below, we comment on these results.

6.1 Functional class

In the results from the Functional class problem, we can see that there are no big differences between the models in overall accuracy. All models predict about 85 %

of the records correctly (Baseline - 85.7 %, Logistic Regression - 85.0 %, SVM - 84.4 %, HMM - 84.5 %). However, looking at the class specific accuracies, we see that the models differ somewhat.

In the case of the Baseline model, looking at the accuracies for the less common classes, that is the Collector and Local classes, the model performs relatively bad (69.2% and 52.5% respectively). The Baseline model places a large portion of the records in the Highway class. Since a very large portion of the records in the data set (about 85 %, see Section 2.3) belong to the Highway class, such a scheme yields a high overall prediction accuracy, while the Collector and Local class accuracies suffer.

The Logistic Regression model achieved a higher accuracy than the Baseline model for the Local class (64.3% vs 52.5% respectively). The Logistic Regression model probably performs slightly better since it has the ability to find a better decision boundary than Baseline model (see Section 4.5).

Furthermore, the SVM model performs slightly better than the Logistic Regression model when it comes to classifying the Local and Collector classes (80.8% vs 68.8% and 68.5% vs 64.3% respectively). This is probably since there is some non-linear relationship in the data which the SVM has the ability to learn, but which the Logistic Regression model cannot (Figure 4.9 on page 34 describes the difference between linear and non-linear classifiers).

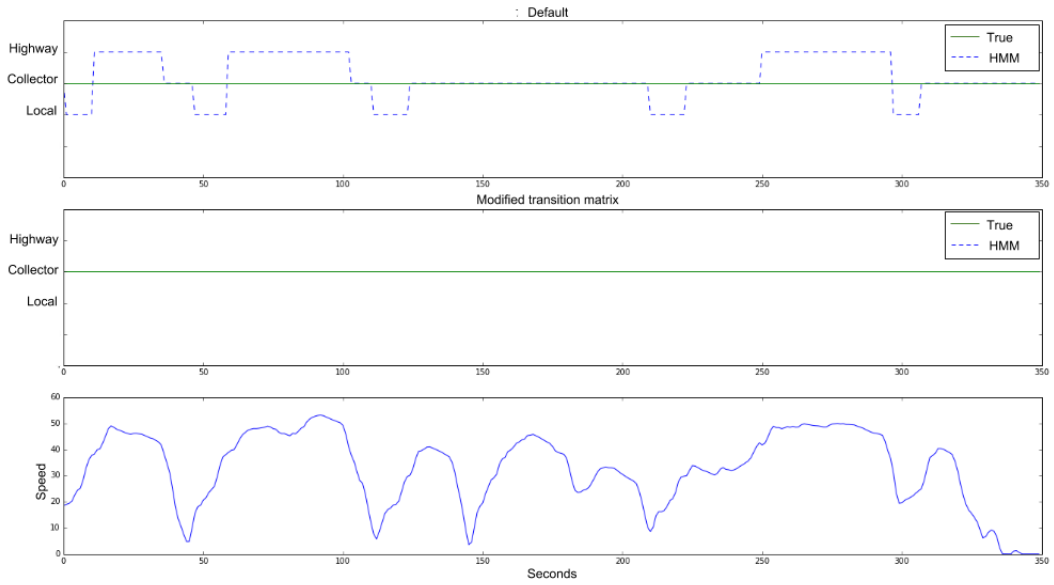


Figure 6.2: The HMM model fails in predicting Collector class, mistaking it for the Highway class when speed is high, and for the Local class when speed is low. This can be seen in the top diagram, which shows the actual driving context class (green line), and the class predicted by the HMM model (dashed blue line). The diagram furthest down shows the vehicle speed. This problem can be mitigated by modifying the model's transition probabilities. In the middle diagram probabilities for staying in a state has been increased, making the HMM model less sensitive to feature variations, predicting the Collector class correctly.

Finally, the HMM model exhibits significant difficulties classifying the Collector class. The Collector class is characterized by high variation of feature values, interfering with the other two classes' values. For example, the Collector class exhibits

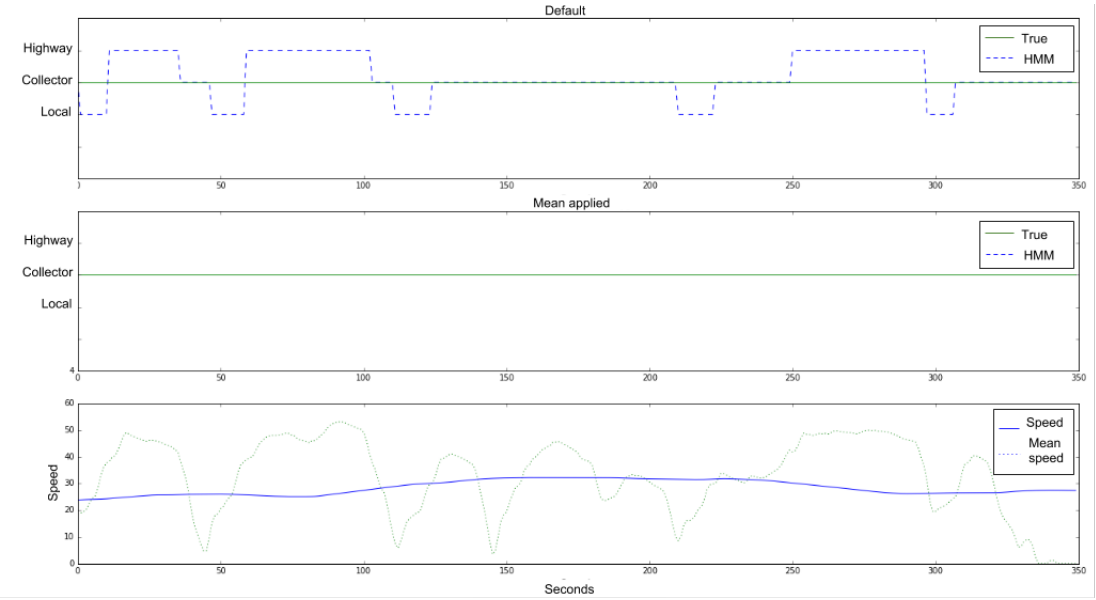


Figure 6.3: Just as Figure 6.2, this figure shows how the prediction by the HMM is to sensitive to changes in speed (top diagram). This figure displays how the problem can be mitigated by applying a rolling mean on the vehicle speed feature. The bottom diagram shows the original speed value (dotted green), and the speed values after applying a rolling mean. The middle diagram shows how the HMM model predicted the Collector class correctly based on the features where the rolling mean was applied.

speed values that varies between low speeds (such as in the Local class) and high speeds (such as in the Highway class). The HMM model should ideally be able to distinguish the Collector class from the other classes using the fact that the emission probability distribution for the Collector class has a high variance, while the other two classes has a lower variance.

As can be seen in Figure 6.2, though, the model misclassifies the Collector class as the Highway class when speed is high, and as the Local class when speed is low. To make the model less sensitive to feature value variations, the transition matrix probabilities were changed. The probabilities for staying in a state were increased, and the probabilities for changing state decreased. This improved the model some, raising its prediction accuracy for the Collector class from 35.1 % to 48.6 %. The result of this modification of the transition matrix probabilities is presented in Figure 6.2. Moreover, one way to deal with difficulties in predicting a class that varies much in its features is to remove the variation. This can be accomplished by applying a rolling mean on the data. Figure 6.3 shows an example of how this helped the HMM model to predict the Collector class better. Applying a mean of six minutes (as is done when creating features for other models) brings Collector class accuracy up to 66.0 % for the HMM, but on the flip side, decreases Local class accuracy to 45.6 %.

Furthermore, one hypothesis for why the HMM performed badly was that feature values (observed variables) are highly dependent on each other. We use records sampled at a rate of $1Hz$, so the speed, for instance, for record at time t seconds is highly dependent on the speed at time $t-1$ seconds. The HMM model, however, does not describe this relationship, but assumes that features are only dependent on the current state. An attempt to alleviate this problem was made by removing

records such that there was only one record per 60 seconds left in the data set. With such a long time between records, record features should not be dependent on each other. However, this test did not result in any significant changes in prediction accuracy (on the order of a couple of percent per class accuracy).

In summary, the models succeed in finding patterns in the vehicle data and classifying the functional classes to a degree of 85 % - 68 % per class accuracy (in the best case). Below, we give examples of some of the situations the models fail to predict correctly, and explain why these situations are difficult for the models to handle.

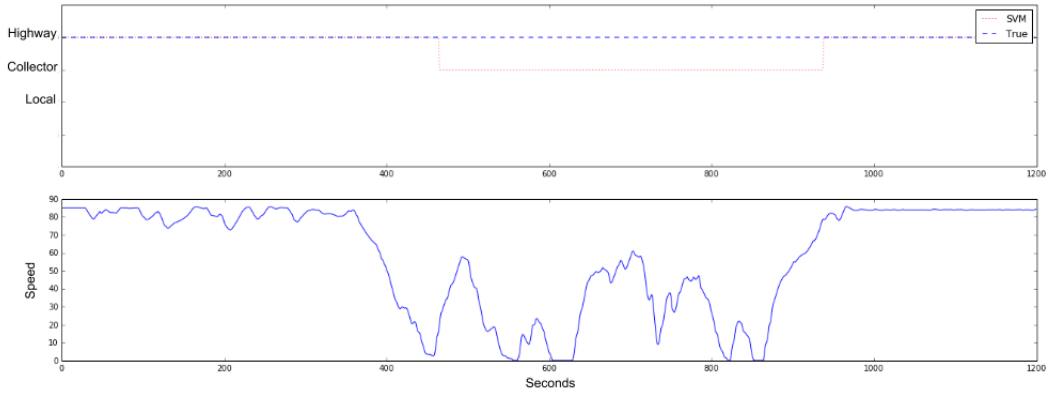


Figure 6.4: During about five minutes a vehicle is experiencing low speeds (bottom diagram) while driving on the highway (top diagram, dashed blue line), possibly due to traffic congestion or road construction work. This causes the model to misclassify these records as Collector (top diagram, red dotted line).

Firstly, there are several occurrences in the data where the vehicle is driving on the highway, but exhibits the patterns of driving on one of the functional classes lower in the hierarchy. An example of such a situation is depicted in Figure 6.4. Here, a vehicle is experiencing low speeds, possibly due to traffic congestion or road construction work. This causes the model to misclassify these records as Collector.

Secondly, the Baseline, Logistic Regression and SVM models use features including the mean of vehicle data aggregated over six minutes (see Section 3.1). When changing between two classes, vehicle data from both of those two classes will be included in the features, making it difficult for the model to predict such situations correctly (this issue is discussed in Section 4.5). Such a situation is presented in Figure 6.5, where a vehicle is changing from Collector class to Highway, increasing speed rapidly. However, since we are aggregating the mean speed over six minutes, the speed change is "spread out" in our feature. This makes it difficult for the model to decide when to change from Collector to Highway, resulting in misclassification during approximately one minute.

Thirdly, the difference in characteristics between the Collector class and the Local class seems to be rather vague. We can see in the confusion matrices (Table 6.10) that many of the Local records are misclassified as Collector records (27.5 % for the SVM model), and vice versa (9.8 % for the SVM model). Figure 6.6 shows a set of records belonging to the Local and Collector classes, exhibiting similar speed patterns, resulting in misclassification of the Local class.

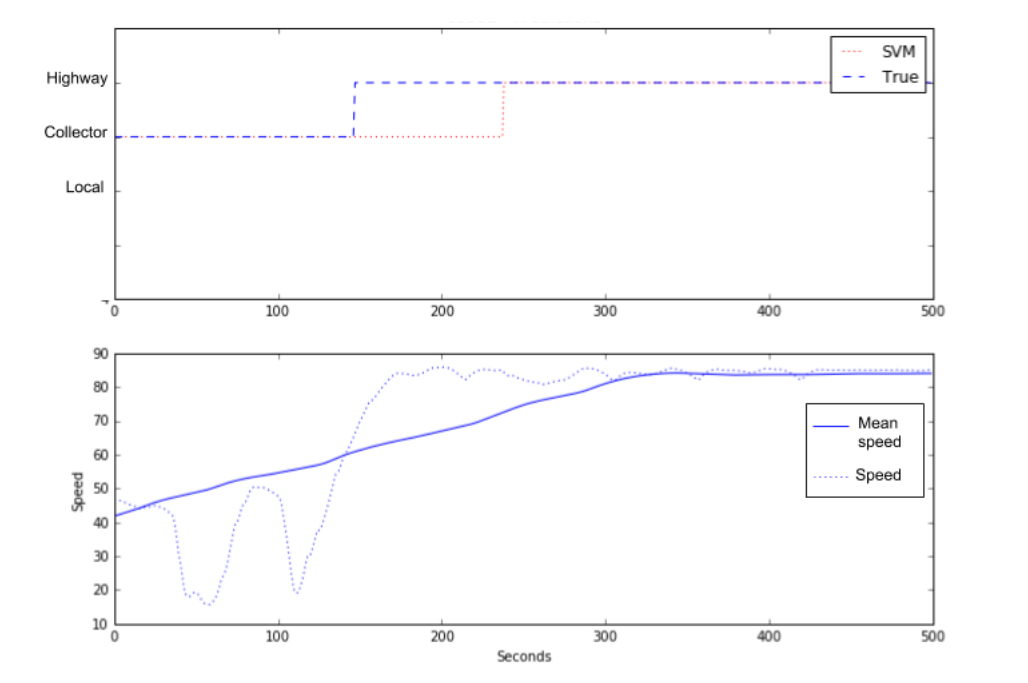


Figure 6.5: A vehicle moves from the Collector class to the Highway class (top graph, dashed blue line) increasing its speed (bottom graph, dotted line). However, since we use features with aggregated data (bottom graph, full line) the speed change is not directly detected by the model, misclassifying it as Collector (top graph, dotted red line)

.

6.2 InCity

In the problem of predicting the InCity-attribute, the SVM performs best, achieving an overall accuracy of 80.2 %. The other models reach 78.9 % (Baseline), 75.9 % (Logistic Regression), and 79.7 % (HMM). Also, the SVM performs well predicting both classes (81.5 % and 75.4 %), while the other models exhibit a more uneven result. The Baseline model and Logistic Regression model are relatively good at predicting *InCity=True* but bad at predicting *InCity=False*, and the other way around for the HMM-model.

The reason for this result is probably similar to why the SVM performed best in the Functional Class problem: it has the ability to describe non-linear relationships which the Baseline and Logistic Regression models cannot.

Furthermore, just as in the case of the Functional Class problem, there are some situations which all of the models fail to classify correctly. Two of the problems we saw when classifying functional class can be seen when classifying the InCity-attribute as well. Firstly, there is a problem of classifying records exhibiting low speeds outside the city (possibly due to traffic congestion or road construction work). We recognize this from the functional class case, where we experienced the same problem when classifying the Highway class. An example of this is presented in Figure 6.7. We can also see the problem of misclassification of records while changing between two classes, due to the fact that we are using features including vehicle data from a time window of six minutes.

In addition, we can see a new situation in the InCity problem which is difficult for

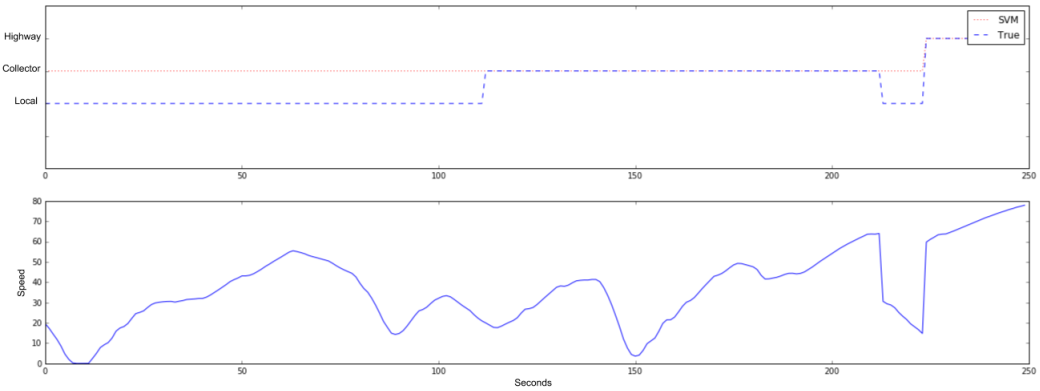


Figure 6.6: A set of records belonging to the Local and Collector classes (top graph, dashed blue line), exhibiting similar speed patterns (bottom graph), resulting in misclassification of the Local (top graph, red dotted line)

.

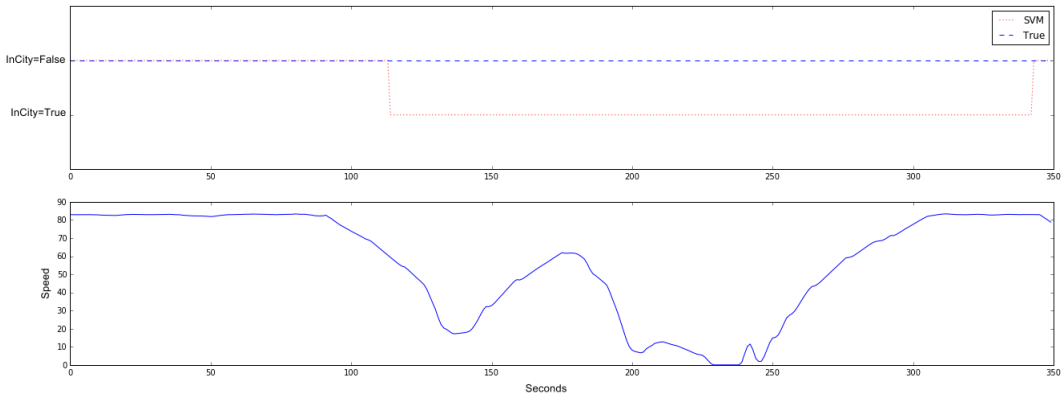


Figure 6.7: The models fail in classifying records outside the city exhibiting low speeds (possibly due to traffic congestion or road construction work).

.

the models to handle. The InCity attribute specifies if a vehicle is driving inside the borders of a city (see Section 2.2). Driving inside a city's borders often implies high frequency of obstacles such as red lights, crossings, etc., which results in relatively slow driving and high variation in speed. However, there could also be highways inside a city's borders, resembling the constraints of roads outside city borders. Driving on a highway inside a city's border often leads to misclassification, which is depicted in Figure 6.8. 86 % of the records classified as *InCity=False*, but actually belonging to *InCity=True* where marked with the functional class Highway label.

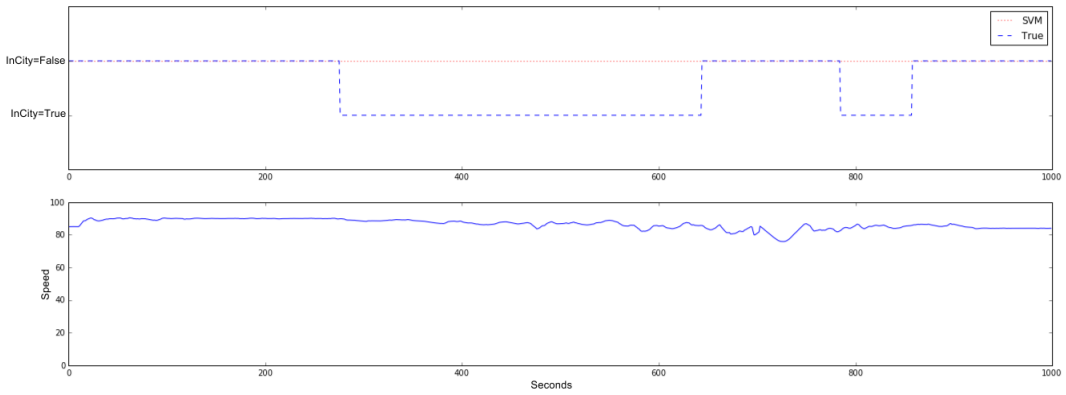


Figure 6.8: Driving on a highway inside a city's borders, which this diagram depicts, leads to misclassification.

.

Predicted	InCity=0	InCity=1
True		
InCity=0	**79.4**	20.6
InCity=1	22.9	**77.1**

(a) Baseline model

Predicted	InCity=0	InCity=1
True		
InCity=0	**75.6**	24.4
InCity=1	23.2	**76.8**

(b) Logistic Regression

Predicted	InCity=0	InCity=1
True		
InCity=0	**82.8**	17.2
InCity=1	24.1	**75.9**

(c) SVM

Predicted	InCity=0	InCity=1
True		
InCity=0	**81.6**	18.4
InCity=1	27.2	**72.8**

(d) HMM

Figure 6.9: Confusion matrices for the different models classifying the InCity problem

Predicted	Higway	Collector	Local
True			
Highway	**88.7**	10.3	1.0
Collector	26.2	**69.2**	4.6
Local	4.6	42.8	**52.6**

(a) Baseline model

Predicted	Higway	Collector	Local
True			
Highway	**87.5**	11.3	1.2
Collector	20.3	**68.8**	10.9
Local	4.9	30.8	**64.3**

(b) Logistic Regression

Predicted	Higway	Collector	Local
True			
Highway	**85.4**	13.6	1.0
Collector	9.4	**80.8**	9.8
Local	4.0	27.5	**68.5**

(c) SVM

Predicted	Higway	Collector	Local
True			
Highway	**91.6**	5.8	2.6
Collector	47.2	**36.1**	16.7
Local	11.8	22.1	**66.1**

(d) HMM

Figure 6.10: Confusion matrices for the different models classifying the Functional Class problem

Chapter 7

Conclusion and discussion

In this thesis, we have applied two driving context classification tasks to four different pattern recognition models. The problems were to classify (1) whether a vehicle is driving in a city or not, and (2) the functional class of the road the vehicle is driving on. The results presented in Chapter 6 show that the models reach the same level of performance when it comes to overall prediction accuracy (76 % - 80 % for classification task (1) and 84 % - 86 % for task (2)). In this chapter, we elaborate on these results and discuss how the performance of the framework can be improved further.

7.1 Model comparison

In this thesis, we have applied our classification tasks to several models with different characteristics and complexity. The results show that there is no significant difference in overall prediction performance between the models (76 % - 80 % for the city task and 84 % - 86 % for the functional class task). In general, the models seem to reach similar conclusions, dividing the data set in roughly the same way. Our conclusion from this result is that the relationship between classes and vehicle data is rather simple. It can be described almost as well with a simple model such as the Baseline model in this thesis, as with a more complex, non-linear model such as the SVM. Applying the problem to another complex model, such as a Neural Network, will thus probably not result in a significant improvement of classification accuracy.

The data we have worked with is sequential, i.e. consecutive records in the data are highly dependent on each other, and we have used two different approaches to handle this. In the case of the Baseline, Logistic Regression, and SVM models, which make their class predictions based solely on vehicle data features, we have aggregated vehicle data from a time window surrounding a given record, creating features by applying mean and variance on that data (described in Chapter 3). The HMM model on the other hand captures the sequential nature of the data "natively". Specifically, using a Markov chain approach, it classifies the driving context at time t_0 based not only on the vehicle data parameters at that point in time, but also the previous predicted class and the probability for changing class. This means that we can use "clean" vehicle data parameters, skipping the pre-processing phase of creating aggregated data features.

In section 4.5 we discussed the potential advantages with the HMM model. For instance, the HMM model has the ability to detect short-term feature changes,

performing better in situations when a vehicle shifts driving context class. However, as the results show, the aggregation approach was more successful than the HMM approach. Specifically, in the functional class task, the HMM failed in predicting the Collector class (see Section 6.1). This could be a result of the fact that we are working with classes with some ambiguity in their definitions (see more discussion on this in Section 7.5 below). The classes share characteristics, confusing the HMM model. The following example clarifies this reasoning. Consider if we had only one feature: the vehicle speed, and two driving context classes: Main Road, which is defined as roads where we reach speeds over 50 km/h, and Highway, which is defined as roads where we reach speeds higher than 80 km/h. The Highway class is included in the Main Road class, and therefore it would be hard for the HMM model to classify Highway correctly. If the Collector class share characteristics with the Highway class and the Local class in this way, that could explain why the HMM performs badly in this case. Another explanation to why the HMM performs badly in predicting the Collector class could be that the emission probabilities are faulty. For instance, we assume features follow a Gaussian distribution, which might not actually be the case.

As the results show, the sliding window feature aggregation approach is relatively successful. Applying the sliding window functions on the vehicle data seems to extract the essence of the vehicle data values for the given driving context class, enabling the models to make correct predictions. However, the results also show that predicting situations where a vehicle shifts between driving context classes, which was discussed in Section 4.5, is in fact a problem leading to faulty predictions.

7.2 Features

We have selected vehicle data parameters for our features based on what has been used in a previous study classifying driving context ([5]), and based on what parameters intuitively are dependent on the driving context. Then, we have created features using rather simple pre-processing functions (mean, variance and max). We used a uniform sliding window size when applying these functions on the features. This approach can be made more elaborate. For instance, our data set contains a large number of vehicle data parameters, and it is possible that there is some parameter we have not tested that better distinguishes driving contexts from each other that the ones we have used. Furthermore, it is not certain the same window size is the best for all features. A given window size might be too small/large to capture the information a feature needs to describe a certain pattern, and this size might differ between features. Further investigation of vehicle data parameters, sliding window functions, and unique window sizes per feature might lead to a better prediction result.

7.3 Dynamic driving context

As presented in Section 1.1, driving context can be divided into static context (road infrastructure) and dynamic context (traffic density, weather conditions, etc.). In this thesis we focus solely on classifying static driving context classes. The main reason for this is that we use a supervised learning approach, requiring a data set

labeled with the classes we want to learn. The data set we use does not contain labels for dynamic driving context, and therefore, we cannot learn such classes.

However, the data set we use is recorded in a real traffic setting, where traffic congestion and bad weather occur. Records labeled with class Highway could have been recorded both in situations when traffic flowed seamlessly and when there was traffic congestion (Figure 6.4 on page 43 shows an example of this). These differences in dynamic driving context conditions confuses our model. For instance, driving in a traffic jam on the Highway is more similar to driving on a road with Local functional class characteristics, than to regular driving on a Highway, which leads to misclassification of such situations.

One solution for this could be to try to find labels for the dynamic constraints in our data set. For instance, we could assume there is a higher probability of a traffic jam inside the borders of a city than outside, and say that the InCity label represents high frequency of traffic jams as well as high frequency of crossings, red lights, etc. Another solution would be to add a time aspect to our driving context classes. Assuming that traffic congestion is most probable during rush hours (a couple of hours during the morning and the afternoon), we could create labels such as *City - Rush hour* and *City - Non-Rush hour* based on the time of day the record was recorded. Such labels would then include both static road infrastructure constraints as well as dynamic traffic density constraints.

Furthermore, this problem could also be solved by collecting dynamic context constraints from other sources than the data set used in this thesis. For instance, there is the *UDRIVE* (eUropean naturalistic Driving and Riding for Infrastructure and Vehicle safety and Environment) project ([3]), where a data set containing naturalistic vehicle data as well as external video data is created. From video data, we could identify high-traffic density situations as well as bad weather conditions, and label vehicle data accordingly.

7.4 Using the GPS signal

The framework we created in this thesis bases its driving context classification solely on vehicle data signals. However, there is another signal, the current GPS coordinates, which is continuously logged by the vehicle, and which could be used when classifying driving context. The driving context of a given road segment is a property that rarely changes. Moreover, vehicles frequently access the same road segments. We can utilize this information by, when classifying a certain road segment, look at what this road segment has been classified as previously. This could for example mitigate the problem of misclassification of Highway segments due to high traffic density or road construction.

For instance, a vehicle data record being recorded during a road construction work situation at a Highway could lead to the framework misclassifying that segment as a Collector or a Local class road. However, previous predictions for that segment, when there was no road construction work going on should have been more accurate, producing a correct prediction of the Highway class. This information could be included in the classification by weighing in previous predicted driving context at the given GPS coordinate. A database of previously predicted driving contexts could be stored in the vehicle locally (containing only that specific vehicles predictions), or at a data center (containing the accumulated predictions of a fleet of vehicles).

7.5 Well-defined classes

One significant factor in succeeding with a classification problem is that the classes we are predicting are well defined. Well defined classes lead to an unambiguous data set, which enables the pattern recognition framework to find the underlying patterns that are unique for a given class. For instance, consider the famous pattern recognition problem of classifying images of hand-written digits. Any literate person can differentiate between a 2 and a 3, which enables creating a clear data set without ambiguities between classes. Noisy data, such as mislabeled images, or images that do not represent digits can easily be sorted out. Ideally, our driving context classes should be as well defined (in terms of driving context constraints) as classes of digits. However, this does not seem to be the case.

For instance, it is clear that driving in the center of a city, where traffic lights, stop signs, and crossings are frequent, puts constraints on the vehicle that are not present when driving on the highway. These are two well-defined situations that should be possible to distinguish between looking at vehicle data. We chose the InCity map attribute from our data set to represent this relationship. However, the InCity map attribute specifies if a road is located inside the borders of a city. As discussed in Section 6.2, this means both roads in a city center as well as highways can be labeled with the InCity attribute, as long as the roads are located inside the borders of a city. In terms of vehicle data and driving context constraints, this introduces ambiguities in the class definitions: the constraints on a highway inside and outside a city's borders does not differ. Such noise in the data is impossible for a model to predict correctly, irrespective of how elaborate classification hypotheses the model can represent. Figure 6.8 on page 46 shows how the model misclassifies such situations.

The functional classification scheme also seems to experience some ambiguities. Functional classification definitions can differ some between countries. A road classified as Collector Road in Germany might differ some from a road classified with the same class in the UK. If a Collector Road in Germany have more similarities with a Highway in the UK than with a Collector Road in the UK, the framework might confuse Collector Road with Highways, and vice versa. Furthermore, even inside the borders of a country, the functional classification scheme can be rather arbitrary. [19] states that "*The process of determining the correct functional classification of a particular roadway is as much an art as it is a science*". Figure 6.6 on page 45 shows us that the Local Road and Collector Road class exhibit the same vehicle data patterns, resulting in misclassification.

The first solution that comes to mind for these ambiguities in driving context classes is to choose another set of classes that better represent and distinguishes different driving contexts. However, in the data set we use, no other global driving context classes are available. Therefore, further investigation is needed on whether there exist more well-defined global driving context classes than the ones used in this thesis. Moreover, the data set we use contain a fairly large number of local driving context classes that was not tested in this thesis. It might be that the local driving context classes have more distinct characteristics, and therefore are more well suited for recognition.

7.6 Application

The framework produced in this thesis is not perfect, and in this chapter we have discussed some possible ways to improve it. However, we would like to finish this chapter by reconnecting to one of the applications that was mentioned in the introduction of this thesis, namely driving style evaluation, and discuss whether the framework has reached a level of performance such that it can be used in such an application.

Driving context is an important parameter in driving style evaluation calculations. Specifically, when interpreting a driving style evaluation score, in order to understand the evaluation, a driver should have access to driving context considered in his/her evaluation calculations. However, driving style evaluation is not a critical task, i.e. if the framework commits an error, the outcome will not be catastrophic. The worst outcome of this is that a high error ratio from the framework leads to users not trusting it. Our framework predicts driving context with an accuracy of around 70 to 85 %, depending on the driving context class. We believe this level of performance is good enough that the framework has the potential of being used in a driving style evaluation context. Furthermore, our framework produces probabilities of how certain it is of its classification. Such probability measurements can be used to decide when to trust the framework, e.g. we can define a threshold of how certain the framework must be on its output in order for us to trust it.

Bibliography

[1] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[2] C. D'Agostino, A. Saidi, G. Scouarnec, and L. Chen. Learning-based driving events classification. In *Proceedings of the 16th International IEEE Annual Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 1778–1783, The Hague, The Netherlands, October 2013.

[3] R. Eenink, Y. Barnard, M. Baumann, X. Augros, and F. Utesch. Udrive: the european naturalistic driving study. In *Transport Research Arena 2014*, Paris, France, 2014.

[4] J. Engström. Real-time recognition of driver behaviour. Master's thesis, University of Sussex, Brighton, UK, 2001.

[5] J. Engström and T. Victor. Real-time recognition of large-scale driving patterns. In *2001 IEEE Intelligent Transportation Systems Conference Proceedings*, pages 1020–1025, Oakland, CA, USA, August 2001.

[6] J. Engström and T. Victor. System and method for real-time monitoring of driver behaviour patterns, 2005. US Patent US-6879969.

[7] J. Engström and T. Victor. Real-time distraction countermeasures. In M. Regan, J. Lee, and K. Young, editors, *Driver Distraction: Theory, Effects and Mitigation*. CRC Press, 2008.

[8] Eurofot. Eurofot web page. `http://www.eurofot-ip.eu/`. Accessed: 2016-08-19.

[9] G.D. Forney. The viterbi algorithm. In *Proceedings of the IEEE*, volume 61, pages 161–174, March 1973.

[10] Here. Here web page. `https://company.here.com/here/`. Accessed: 2016-08-19.

[11] J. S. Hickman and R. J. Hanowski. Evaluating the benefits of a low-cost driving behavior management system in commercial vehicle operations. Technical Report FMCSA-RRR-10-033, U.S. Department of Transportation, Federal Motor Carrier Safety Administration, 2010.

[12] HMM-Learn. Hmm-learn github repository. `https://github.com/hmmlearn/`. Accessed: 2016-08-19.

[13] Progressive Insurance. Snapshot web page. `https://www.progressive.com/auto/snapshot/`. Accessed: 2016-08-21.

[14] Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization, Geneva, CH, December 2015.

[15] G. King and L. Zeng. Logistic regression in rare events data. *Political Analysis*, 9:137–163, 2001.

[16] J. Löfstedt and M. Svensson. Balthazar – a fuzzy expert for driving situation detection. Master's thesis, Department of Information Technology, Lund University, 2000.

[17] K. Murphy. *Machine Learning: a Probabilistic Perspective*. MIT Press, 2013.

[18] A. Niculescu-Mizil and R. Caruana. Predicting good probabilities with supervised learning. In *ICML '05 Proceedings of the 22nd international conference on Machine learning*, Bonn, Germany, 2005.

[19] U.S. Department of Transportation. Highway Functional Classification: Concepts, Criteria and Procedures. Technical Report FHWA-PL-13-026, U.S. Department of Transportation, Washington, DC, 2013.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[22] M. D. Richard and R. P. Lippmann. Neural network classifiers estimate bayesian a-posteriori probabilities. *Neural Computation*, 3(4):461–483, December 1991.

[23] F. Sagberg, Selpi, G. F. Piccinini, and J. Engström. A review of research on driving styles and road safety. *Human Factors*, 57(7):1248–1275, November 2015.

[24] S. M. Savaresi, V. Manzoni, A. Corti, and P. De Luca. Estimation of the driver-style economy and safety via inertial measurements. In G. Meyer and J. Valldorf, editors, *Advanced Microsystems for automotive applications*, pages 121–129. Springer, 2010.

[25] M. Schmidt, N. Le Roux, and F. Bach. Minimizing finite sums with the stochastic average gradient. *Math. Program.*, 2016.

[26] MyDrive Solutions. Mydrive solutions web page. `http://www.mydrivesolutions.com/`. Accessed: 2016-08-21.

[27] E. Tivesten and M. Dozza. Driving context and visual-manual phone tasks influence glance behavior in naturalistic driving. *Transportation Research Part F: Traffic Psychology and Behaviour*, 26:1369–8478, September 2014.

[28] I. Van Schagen, R. Welsh, A. Backer-Grøndahl, M. Hoedemaeker, T. Lotan, A. Morris, F. Sagberg, and M. Winkelbauer. Towards a large-scale european naturalistic driving study: main findings of prologue. Technical report, SWOV Institute for Road Safety Research, Leidschendam, The Netherlands, 2011.

[29] T. Wu, C. Lin, and W. C. Ruby. Probability estimates for multi-class classification by pairwise coupling. *The Journal of Machine Learning Research*, 5:975–1005, 2005.

[30] L. Zettlemoyer. Lecture 8 - kernels and kernelized perceptron. Machine Learning Course (CSE446) Winter Quarter, University of Washington, Seattle, 2015.

Appendix A

Code

A.1 Baseline model

A.1.1 baseline.py

```
import numpy as np
import sklearn.metrics

class BaselineModel:

    beacons_ = []

    def __init__(self):
        pass

    def fit(self, df, features, nr_labels):
        """ Train model, i.e. find values for beacons. """

        beacons = []
        for cls in range(nr_labels):
            class_data = df.loc[df.label == cls]
            beacon = class_data[features].mean().as_matrix()
            beacons.append(beacon)

        self.beacons_ = beacons


    def predict(self, dataframe, features, nr_labels):
        """ Predict a set of samples and return prediction accuracy
        and confision matrix. """

        df = dataframe.copy()
        # for each sample in the test set, compute the distance to
        # all the beacons and put in a column named by the class
        # label
        for cls in range(nr_labels):
            df[str(cls)] = (df[features] - self.beacons_[cls]).apply(
                np.square).apply(np.sum, axis=1).apply(np.sqrt)

```

```
        labels = [str(lab) for lab in range(nr_labels)]
        df['pred'] = df[labels].idxmin(axis=1)
        df.pred = df.pred.astype(float)

        return df.pred.as_matrix()


    def score(self, df, features, nr_labels):
        """ Predict a set of samples and return prediction accuracy
        and confision matrix. """

        # for each sample in the test set, compute the distance to
        # all the beacons and put in a column named by the class
        # label
        for cls in range(nr_labels):
            df[str(cls)] = (df[features] - self.beacons_[cls]).apply(
                np.square).apply(np.sum, axis=1).apply(np.sqrt)

        labels = [str(lab) for lab in range(nr_labels)]
        df['pred'] = df[labels].idxmin(axis=1)
        df.pred = df.pred.astype(float)

        accuracy = (df.label == df.pred).sum() / len(df)
        confusion_matrix = sklearn.metrics.confusion_matrix(
            df.label, df.pred)
        confusion_matrix = confusion_matrix / np.sum(
            confusion_matrix, axis=1)[:,None]
        return accuracy, confusion_matrix

    def get_confidences(self, df, nr_labels):
        """ Calculate the confidence probabilities of all the
        classes. The confidence is represented by the ratio of
        correctly predicted samples belonging to a class versus the
        total number of samples belonging to that class.

        :param df: A dataframe with all the samples and the
        predictions for these samples specified in a "pred"-column.
        :return: A confidence for the given class.
        """

        confidences = []
        for cls in range(nr_labels):
            total_class_records = (df.label == cls).sum()
            correctly_predicted_records = (
                (df.label == cls) & (df.label == df.pred)).sum()
            confidences.append(
                correctly_predicted_records / total_class_records)

        return confidences
```

A.1.2 baseline-runner.py

```
import pandas as pd
import logging
import mmlutils
import sklearn.preprocessing
import baseline
import constants

LOG_FILE = 'log.log'

ORIGINAL_SAMPLE_RATE = 10
RESAMPLE_RATE = 10
FILTER_RATE = 1

VEHICLE_DATA_PARAMS = ['mVehicleSpeed', 'SteeringWheelAngle',
                       'mEngineSpeed', 'mBrakePedalPos',
                       'mGearSelected', 'mAccelPedalPos']

WINDOW_SIZE = 6

FC = False

if FC:
    NR_LABELS = 3
    TRAIN_DATA_DIR = constants.FC_TRAIN_DATA_DIR
    TEST_DATA_DIR = constants.FC_TEST_DATA_DIR
    FEATURES = constants.FC_FEATURES
else:
    NR_LABELS = 2
    TRAIN_DATA_DIR = constants.CITY_TRAIN_DATA_DIR
    TEST_DATA_DIR = constants.CITY_TEST_DATA_DIR
    FEATURES = constants.CITY_FEATURES

global logger
logger = mmlutils.setup_logging(LOG_FILE)

# read files
train_dict = mmlutils.read_files(TRAIN_DATA_DIR)
test_dict = mmlutils.read_files(TEST_DATA_DIR)

# prepare data
train_dict = mmlutils.prepare_dict(train_dict, ORIGINAL_SAMPLE_RATE,
               RESAMPLE_RATE, FILTER_RATE, VEHICLE_DATA_PARAMS,
               FEATURES, WINDOW_SIZE, FC)
training_set = pd.concat(train_dict.values(), ignore_index=True)
test_dict = mmlutils.prepare_dict(test_dict, ORIGINAL_SAMPLE_RATE,
               RESAMPLE_RATE, FILTER_RATE, VEHICLE_DATA_PARAMS,
               FEATURES, WINDOW_SIZE, FC)
test_set = pd.concat(test_dict.values(), ignore_index=True)

logger.debug('(1) Training and (2) Test set records')
```

```
logger.info(mmlutils.get_nr_records_str(training_set, 'label'))
logger.info(mmlutils.get_nr_records_str(test_set, 'label'))

# do standard scaling on data
scaler = sklearn.preprocessing.StandardScaler()
training_set[FEATURES] = scaler.fit_transform(training_set[FEATURES])
test_set[FEATURES] = scaler.transform(test_set[FEATURES])

# train model, i.e find values for beacons
clf = baseline.BaselineModel()
clf.fit(training_set, FEATURES, NR_LABELS)
logger.info('Beacons: {}'.format(clf.beacons_))

# get result
accuracy, confusion_matrix = clf.score(test_set, FEATURES, NR_LABELS)
logger.info('Test set results: \nAccuracy: {} '
            '\nConfusion Matrix: \n{}\n'.format(
              accuracy, confusion_matrix))

print('Done!')
```

A.2 Logistic Regression/SVM

```
import pandas as pd
import sklearn.svm
import sklearn.grid_search
import sklearn.linear_model
import logging
import mmlutils
import numpy as np
import sklearn.externals
import constants

# specifies if we are classifying the functional class or the InCity
# problem
FC = False
# specifies if we are using a SVM or a Logistic Regression model
SVM = False

WINDOW_SIZE = 6
CROSSVAL_FOLDS = 4

LOG_FILE = 'log.log'

ORIGINAL_SAMPLE_RATE = 10
RESAMPLE_RATE = 10

VEHICLE_DATA_PARAMS = ['mVehicleSpeed', 'SteeringWheelAngle',
                       'mEngineSpeed', 'mBrakePedalPos',
                       'mGearSelected', 'mAccelPedalPos']
```

```
28
29 CLASSIFIER_FILE_NAME = ''
30
31 if FC:
32     CLASSIFIER_FILE_NAME = CLASSIFIER_FILE_NAME + 'fc_'
33     FEATURES = constants.FC_FEATURES
34
35     TRAIN_DATA_DIR = constants.FC_TRAIN_DATA_DIR
36     TEST_DATA_DIR = constants.FC_TEST_DATA_DIR
37
38     NR_LABELS = 3
39
40 else:
41     CLASSIFIER_FILE_NAME = CLASSIFIER_FILE_NAME + 'city_'
42     FEATURES = constants.CITY_FEATURES
43
44     TRAIN_DATA_DIR = constants.CITY_TRAIN_DATA_DIR
45     TEST_DATA_DIR = constants.CITY_TEST_DATA_DIR
46
47     NR_LABELS = 2
48
49 if SVM:
50     CLASSIFIER_FILE_NAME = CLASSIFIER_FILE_NAME + 'svm'
51     FILTER_RATE = 10
52     C_VALS = [1e-5, 1e-4, 0.001, 0.01, 0.1, 1, 10]
53     KERNELS = ['rbf', 'poly']
54
55 else:
56     CLASSIFIER_FILE_NAME = CLASSIFIER_FILE_NAME + 'lr'
57     FILTER_RATE = 1
58     C_VALS = [1e-6, 1e-5, 1e-4, 0.001, 0.01, 0.1, 1, 10, 100]
59
60 global logger
61 logger = mmlutils.setup_logging(LOG_FILE)
62
63 # read files
64 train_dict = mmlutils.read_files(TRAIN_DATA_DIR)
65 test_dict = mmlutils.read_files(TEST_DATA_DIR)
66
67 # prepare data
68 train_dict = mmlutils.prepare_dict(train_dict, ORIGINAL_SAMPLE_RATE,
69                 RESAMPLE_RATE, FILTER_RATE, VEHICLE_DATA_PARAMS,
70                 FEATURES, WINDOW_SIZE, FC)
71 test_dict = mmlutils.prepare_dict(test_dict, ORIGINAL_SAMPLE_RATE,
72                 RESAMPLE_RATE, FILTER_RATE, VEHICLE_DATA_PARAMS,
73                 FEATURES, WINDOW_SIZE, FC)
74
75 crossval_folds = mmlutils.create_cross_validation_folds(
76     train_dict, CROSSVAL_FOLDS)
77 logger.info('Crossvalidation␣folds:␣\n{}'.format(
78     [list(d.keys()) for d in crossval_folds]))
```

```
79
80  # do crossvalidaion on set of hyper paramters to find optimal hyper
81  # paramter value
82  total_accuracies = []
83  local_accuracies = []
84  conf_matrices = []
85
86  if SVM:
87      for kernel in KERNELS:
88          for c in C_VALS:
89              logger.info('\n\nKernel: {}, C: {}'.format(kernel, c))
90
91              # train and test model
92              classifier = sklearn.svm.SVC(
93                  verbose=True, cache_size=1000,
94                  class_weight='balanced', kernel=kernel, C=c)
95              accuracy, conf_matrix = mmlutils.get_accuracy_cv(
96                  crossval_folds, FEATURES, classifier, logger=logger)
97
98              # record results
99              local_accuracy = conf_matrix[NR_LABELS-1][NR_LABELS-1]
100             total_accuracies.append(accuracy)
101             conf_matrices.append(conf_matrix)
102             local_accuracies.append((local_accuracy, c, kernel))
103
104 else:
105     for c in C_VALS:
106         logger.info('\n\nC: {}'.format(c))
107
108         # train and test model
109         classifier = sklearn.linear_model.LogisticRegression(
110             solver='sag', class_weight='balanced', C=c)
111         accuracy, conf_matrix = mmlutils.get_accuracy_cv(
112             crossval_folds, FEATURES, classifier, logger=logger)
113
114         # record results
115         local_accuracy = conf_matrix[NR_LABELS-1][NR_LABELS-1]
116         total_accuracies.append(accuracy)
117         conf_matrices.append(conf_matrix)
118         local_accuracies.append((local_accuracy, c))
119
120 # find parameters for the max value
121 max_param_tuple = max(local_accuracies, key=lambda x: x[0])
122 logger.info('Measured local accuracies: {}'.format(local_accuracies))
123 logger.info('Parameters that yeilded maximum accuracy: {}'.format(
124     max_param_tuple))
125 logger.info('Measured accuracies: {}'.format(total_accuracies))
126 logger.info('Measured confidence matrices: {}'.format(conf_matrices))
127
128 # get result on test set
129 if SVM:
```

```
    classifier = sklearn.svm.SVC(verbose=True, cache_size=1000,
    class_weight='balanced', kernel=max_param_tuple[2],
    C=max_param_tuple[1])
else:
    classifier = sklearn.linear_model.LogisticRegression(
        solver='sag', class_weight='balanced', C=max_param_tuple[1])

training_set = pd.concat(train_dict.values(), ignore_index=True)
test_set = pd.concat(test_dict.values(), ignore_index=True)
accuracy, conf_matrix = mmlutils.get_accuracy(
    training_set, test_set, FEATURES, classifier)
# sklearn.externals.joblib.dump(classifier, CLASSIFIER_FILE_NAME + '.pkl
    ')
logger.info('Test set results: \nAcc: {} '
    '\nConfusion Matrix: \n{}'.format(accuracy, conf_matrix))

print('Done!')
```

A.3 HMM

```
import numpy as np
import hmmlearn
import hmmlearn.hmm
import pandas as pd
import mmlutils
import matplotlib.pyplot as plt
import sklearn.metrics
import shelve
import constants


def _modify_transmat_distribution(transmat, move_ratio):
    """ Modifies a transition matrix such that the probability of
    staying in a state is increased (and moving to another state is
    decreased). """
    for i in range(len(transmat)):
        steal_sum = 0
        for j in range(len(transmat)):
            if j != i:
                steal = move_ratio * transmat[i][j]
                transmat[i][j] = transmat[i][j] - steal
                steal_sum = steal_sum + steal
        transmat[i][i] = transmat[i][i] + steal_sum


def _low_pass(df, sample_rate, features):
    """ Applies a sliding window mean on a feature in a
    data frame. """

    WINDOW_SIZE = 6
```

```
31
32      for feat in features:
33          df[feat] = pd.rolling_mean(df[feat],
34              mmlutils.nr_window_records(WINDOW_SIZE, sample_rate),
35              center=True)
36
37      return df
38
39  def _get_transition_counts(ser, i, j):
40      """ Computes the number of transitions from value i to value j in
41       a pandas series."""
42
43      func = lambda x: x[0] == i and x[1] == j
44      return pd.rolling_apply(ser, 2, func).sum()
45
46
47  def _create_covariance_matrix(varaiances):
48      """ Creates a diagonal covariance matrix. """
49
50      covariance_matrix = np.zeros((len(varaiances), len(varaiances)))
51      for i in range(len(varaiances)):
52          for j in range(len(varaiances)):
53              if i == j:
54                  covariance_matrix[i, j] = varaiances[i]
55
56      return covariance_matrix
57
58
59  def _fix_sum_to_one(matrix):
60      """ Ensures that all the values in a row of a transition matrix
61      sums up to one. """
62
63      for row_index in range(len(matrix)):
64          row = matrix[row_index]
65          missing = 1 - sum(row)
66          row[row_index] = row[row_index] + missing
67
68      return matrix
69
70
71  def _get_init_probabilities(trips, nr_labels):
72      """ Compute the probabilities of the initial class. """
73
74      initial_class_probs = np.zeros(nr_labels)
75      for trip in trips:
76          this_inital_class = trip.iloc[0].label
77          initial_class_probs[this_inital_class] = \
78          initial_class_probs[this_inital_class] + 1
79
80      return initial_class_probs / len(trips)
81
```

```
82
83  def _get_nr_transitions(trans_mat):
84      """ Calculate the total number of transitions from a matrix of
85      inter class transitions. """
86
87      sum = 0
88      for i in range(len(trans_mat)):
89          for j in range(len(trans_mat)):
90              if i != j:
91                  sum = sum + trans_mat[i][j]
92
93      return sum
94
95
96  def _get_transition_matrix_list(class_sequence, nr_labels):
97      """ Creates a matrix with the number of inter class transitions
98      from a sequence of classes. """
99
100     trans_mat = np.zeros((nr_labels, nr_labels))
101     for i in range(len(class_sequence)-1):
102         src = class_sequence[i]
103         dest = class_sequence[i+1]
104         trans_mat[src][dest] = trans_mat[src][dest] + 1
105
106     return trans_mat
107
108
109 def _get_transition_matrix_df(trips, nr_labels):
110     """ Compute transistion matrix"""
111
112     transition_matrix = np.zeros((nr_labels, nr_labels))
113     total_nr_records_per_class = np.zeros(nr_labels)
114
115     for trip in trips:
116
117         # count nr records per class
118         for cls in range(nr_labels):
119             total_nr_records_per_class[cls] = \
120                 total_nr_records_per_class[cls] + \
121                 (trip.label == cls).sum()
122
123         # count nr transitions
124         for i in range(nr_labels):
125             for j in range(nr_labels):
126                 # get count of the transitions between
127                 # classes i and j
128                 transition_matrix[i, j] = transition_matrix[i, j] + \
129                     _get_transition_counts(trip.label, i, j)
130
131     # normalize each row of transitions counts
132     for row in range(len(transition_matrix)):
```

```
133         transition_matrix[row] = \
134             transition_matrix[row] / total_nr_records_per_class[row]
135 
136     transition_matrix = _fix_sum_to_one(transition_matrix)
137 
138     return transition_matrix
139 
140 
141 def _get_gaussian_parameters(df, nr_labels, features):
142     """ Computes the mean and the variance of the feature variables
143     conditional to each class. """
144 
145     means_per_class = []
146     covariances_per_class = []
147     for cls in range(nr_labels):
148         class_samples = df.loc[df.label == cls]
149         means = class_samples[features].mean()
150         means_per_class.append(means.as_matrix())
151         covariances = class_samples[features].cov().as_matrix()
152         covariances_per_class.append(covariances)
153 
154     return means_per_class, np.array(covariances_per_class)
155 
156 
157 def _train_model(training_set_dict, features, nr_labels):
158     """ Train the model, i.e find start probabilities, transition
159     probabilities, and emission probability parameters. """
160 
161     # compute probability distribution paramters ...
162     initial_class_probs = _get_init_probabilities(
163         training_set_dict.values(), nr_labels)
164     print('Initial␣class␣probabilities:␣\n', initial_class_probs)
165 
166     transition_matrix = _get_transition_matrix_df(
167         training_set_dict.values(), nr_labels)
168     print('Transition␣matrix:␣\n', transition_matrix)
169 
170     # compute probability distributions
171     means, variances = _get_gaussian_parameters(pd.concat(
172         training_set_dict.values()), nr_labels, features)
173     print('Means:␣\n', means)
174     print('Variances:␣\n', variances)
175 
176     # initialize model
177     np.random.seed(42)
178     model = hmmlearn.hmm.GaussianHMM(n_components=nr_labels,
179         covariance_type='full', verbose=True)
180     model.startprob_ = initial_class_probs
181     model.transmat_ = transition_matrix
182     model.means_ = means
183     model.covars_ = variances
```

```
184
185     return model
186
187 def _predict(test_set_dict, features):
188     """ Get prediction and measure accuracy on test set. """
189
190     # get prediction of a set of trips and accumulate to two lists
191     prediction = np.array([])
192     y_test = np.array([])
193     for trip in test_set_dict.values():
194         trip_id = str(trip.iloc[0].trip_id)
195         X_test = trip[features]
196         this_y_test = trip.label.as_matrix()
197         logprob, this_prediction = model.decode(X_test)
198         print('Logprob: \n', logprob)
199         db[trip_id] = this_prediction
200         prediction = np.append(prediction, this_prediction)
201         y_test = np.append(y_test, this_y_test)
202
203     return prediction, y_test
204
205 def _prepare_data(df_dict, features, filter_rate):
206     """ Preapare the data. """
207
208     ORIGINAL_SAMPLE_RATE = 10
209     RESAMPLE_RATE = 10
210     FILTER_RATE = 1
211
212     prepared_dict = dict()
213     for file, df in df_dict.items():
214         df = mmlutils.downsample(df, RESAMPLE_RATE, columns=FEATURES)
215         sample_rate = ORIGINAL_SAMPLE_RATE / RESAMPLE_RATE
216         df.SteeringWheelAngle = df.SteeringWheelAngle.abs()
217         df = mmlutils.filter_nth(df, FILTER_RATE)
218
219         mmlutils.create_labels(df, FC)
220         df = df.dropna(subset=FEATURES + ['label'])
221         df = df.reset_index(drop=True)
222         prepared_dict[file] = df
223
224     return prepared_dict
225
226
227 FC = True
228 if FC:
229     TRAIN_DATA_DIR = constants.FC_TRAIN_DATA_DIR
230     TEST_DATA_DIR = constants.FC_TEST_DATA_DIR
231     NR_LABELS = 3
232 else:
233     TRAIN_DATA_DIR = constants.CITY_TRAIN_DATA_DIR
234     TEST_DATA_DIR = constants.CITY_TEST_DATA_DIR
```

```
    NR_LABELS = 2

FEATURES = ['mVehicleSpeed', 'SteeringWheelAngle',
           'mEngineSpeed','mGearSelected', 'mAccelPedalPos']

LOG_FILE = 'log.log'

global logger
logger = mmlutils.setup_logging(LOG_FILE)
np.set_printoptions(suppress=True)

# read data
train_dict_raw = mmlutils.read_files(TRAIN_DATA_DIR)
test_dict_raw = mmlutils.read_files(TEST_DATA_DIR)

# prepare data
train_dict = _prepare_data(train_dict_raw, FEATURES, filter_rate)
test_dict = _prepare_data(test_dict_raw, FEATURES, filter_rate)

model = _train_model(train_dict, FEATURES, NR_LABELS)

prediction, y_test = _predict(test_dict, FEATURES)

accuracy = (prediction == y_test).sum()/len(y_test)
print('Accuracy: {}'.format(accuracy))

confusion_matrix = sklearn.metrics.confusion_matrix(
    y_test, prediction)
confusion_matrix = confusion_matrix / np.sum(
    confusion_matrix, axis=1)[:,None]
logger.info('Confusion matrix: \n{}'.format(confusion_matrix))

predicted_transmat = _get_transition_matrix_list(
    prediction, NR_LABELS)
predicted_nr_transitions = _get_nr_transitions(predicted_transmat)
predicted_transmat_ratio = predicted_transmat / np.sum(
    predicted_transmat, axis=1)[:,None]
logger.info('Predicted nr transitions: {} \n'
    'Transition matrix: \n{}'.format(
    predicted_nr_transitions, predicted_transmat_ratio))

real_transmat = _get_transition_matrix_list(y_test, NR_LABELS)
real_nr_transitions = _get_nr_transitions(real_transmat)
real_transmat_ratio = real_transmat / np.sum(
    real_transmat, axis=1)[:,None]
logger.info('Actual nr transitions: {} \n'
    'Transition matrix: \n{}'.format(
    real_nr_transitions, real_transmat_ratio))

print('Done!')
```

A.4 Utility functions

```
import pandas as pd
import numpy as np
import logging
import os
import random
import matplotlib.pyplot as plt
import sklearn.preprocessing

def downsample(df, n, columns=None):
  """ Reduces the number of samples in a pandas dataframe. Keeps
    every nth sample and removes the rest. The value of a sample p
    which we're keeping is calculated by taking the mean of p's n
    surrounding samples.

  :param df: The dataframe that should be downsampled
  :param n: Keep every nth sample. E.g. setting n to 3 would result
    in that every 3rd sample would be kept and every
  3+1th and 3+2th sample would be removed.
  :param columns: The columns to apply mean calculation on.
    :return: A downsampled data frame.
  """

  window_size = n
  if columns:
    # apply mean on given columns
    df[columns] = pd.rolling_mean(
           df[columns], window_size, center=True)

  # filter out every nth row
  df = df.iloc[::n, :]

  return df.reset_index(drop=True)


def filter_nth(df, n):
    """ Filters out all but every nth record from a dataset.

    :param df: The dataframe
    :param n: Keep every nth record.
    :return: The filtered dataframe
    """

    df = df.iloc[::n, :]
    return df.reset_index(drop=True)


def nr_window_records(window_minutes, sample_rate):
    """ Calculates the number of samples that should be included in a
    window of samples given the window size in terms
```

```
    of minutes.
    the window's time should be.

    :param window_minutes: Window size in terms of minutes
    :param sample_rate: Sample rate in dataset in terms of samples
    per second (Hz).
    :return: The number of samples corresponding to the given number
    of minutes.
    """

    return int(window_minutes * 60 * sample_rate)


def get_nr_records_str(df, label_column):
  """ Creates a string with the number of records for each unique
    label in a pandas dataframe

  :param df: The dataframe
  :param label_column: The column in the dataframe that contains
    the labels
  :return: A string
  """

  labels = df[label_column].dropna().unique()
  label_counts = \
       {label:(df[label_column] == label).sum() for label in labels}
  string = []
  for label, count in label_counts.items():
    string.append(
           'Number of label {} records: {}'.format(label, count))

  return '\n'.join(string)


def create_labels(df, column_value_mappings):
  """ Adds a label column to a pandas dataframe, and assigns each
    row a label given the column-value mappings in
    column_value_mappings.

  :param df: The dataframe
  :param column_value_mappings: A list of tuples describing the
    column-value mappings that will be used to create the labels.
    First entry should be a column name. Second entry shoul be a list
    of values. E.g. if [('column_1':[1, 2]), ('column_1':[0])] is
    given, all rows with value 1 or 2 in column_1 will be given label
    value 0, and all rows with value 0 in column_1 label 1.
  """

  LABEL_COLUMN_NAME = 'label'
  df[LABEL_COLUMN_NAME] = pd.Series(len(df) * np.nan)

```

```
101   for label, mapping in enumerate(column_value_mappings):
102     column_name = mapping[0]
103     column_value_list = mapping[1]
104     df.ix[df[column_name].isin(
105             column_value_list), LABEL_COLUMN_NAME] = label
106
107
108 def exceed_thresh(ser, window, threshold):
109     """ Calculates how many times an attribute in a pandas Series
110     object exceeded a given threshold during a given time window.
111
112     :param ser: The panads Series
113     :param window: The time window.
114     :param threshold: The threshold
115     """
116
117     ret = ser > threshold
118     return pd.rolling_sum(ret, int(window), center=True)
119
120
121 def read_files(data_dir, columns=None, subset=None):
122     """ Reads a set of .csv files from a given directory. Retruns
123     all the files in a dictionary where the key is the file name and
124     the value is the pandas dataframe with the file contents.
125     """
126
127     if not subset:
128         input_files = [f for f in os.listdir(data_dir) if f.endswith('.
              csv')]
129     else:
130         input_files = [f for f in os.listdir(data_dir) if f.endswith('.
              csv') and f in subset]
131     input_dataframes = dict()
132     for file in input_files:
133         df = pd.read_csv(data_dir + file, usecols=columns)
134         input_dataframes[file] = df
135
136     return input_dataframes
137
138
139 def create_training_and_test_sets(dfs_dict, test_set_size):
140     """ Splits a dictionary of <filename:dataframe> entries into a
141     "test set dictionary" with one portion of the
142     <filename:dataframes>s and a "training set dictionary" with the
143     rest of the <filename:dataframes>s.
144
145     :param dfs_dict: The original dictionary of dataframes
146     :param test_set_size: Size of test set in terms of ratio (e.g. 0.2)
147     :return: (1) the dictionary of training dataframes and (2) the
           dictionary of test dataframes
148     """
```

```
149
150     input_files = list(dfs_dict.keys())
151     nr_files_in_test_set = int(test_set_size * len(input_files))
152     random.shuffle(input_files)
153     training_set_dict = {file:dfs_dict[file] for file in input_files[
            nr_files_in_test_set:]}
154     test_set_dict = {file:dfs_dict[file] for file in input_files[:
            nr_files_in_test_set]}
155
156     return training_set_dict, test_set_dict
157
158
159 def create_cross_validation_folds(dfs_dict, nr_folds):
160     """ Divides a set of dataframe into nr_folds equaly large
161     subsets. """
162
163     input_files = list(dfs_dict.keys())
164     random.shuffle(input_files)
165     folds_names = [input_files[i::nr_folds] for i in range(nr_folds)]
166     folds = []
167     for fold in folds_names:
168         folds.append({file:dfs_dict[file] for file in fold})
169
170     return folds
171
172
173 def setup_logging(log_file):
174     """ Sets up logging such that messages of at least level DEBUG
175     gets logged to a log file.
176
177     :param log_file: The log file.
178     :return: A logger.
179     """
180
181     # create logger
182     logger = logging.getLogger('root')
183     logger.setLevel(logging.DEBUG)
184
185     # file handler has level debug
186     fh = logging.FileHandler(log_file, mode='w')
187     fh.setLevel(logging.DEBUG)
188
189     # create formatter
190     formatter = logging.Formatter('%(asctime)s␣%(levelname)s:%(message)s'
            )
191     fh.setFormatter(formatter)
192     logger.addHandler(fh)
193
194     return logger
195
196
```

```
197 def drop_rows(df, cls, fraction):
198     """ Remove a protion of the rows belonging to the given class.
199 
200     :param df: The dataframe we want to delete rows from.
201     :param fraction: The fraction of samples we want to keep.
202     :param cls: The class we want to remove samples from.
203     :return: A dataframe were a fraction of the given class' samples
204     have been removed.
205     """
206 
207     # extract samples beloning to remove-class
208     remove_class_samples = df[df.label == cls]
209     if len(remove_class_samples) > 0:
210         df = df[df.label != cls]
211 
212         # remove fraction of samples
213         remove_class_samples = \
214             remove_class_samples.sample(frac=fraction)
215 
216         # re-include samples from remove-class to keep
217         df = df.append(remove_class_samples)
218 
219     return df.reset_index(drop=True)
220 
221 
222 def _get_coefficient(dataset_share, me_vs_hw_db_ratio,
223         hw_dataset_share):
224     """ Private method used by distribute data. Returns a the ration
225     of samples for a set of classes to keep in order to reach a
226     certain distribution. """
227 
228     return (me_vs_hw_db_ratio * hw_dataset_share) / dataset_share
229 
230 
231 def count_records(df, nr_labels):
232     """ Returns the number of samples of each class in the given data
233     frame. Class must be encoded in the 'label'-column of the data
234     frame.
235 
236     :param df: The dataframe to count records in.
237     :return: A list of numbers, where the count for class i can be
238     found at index i.
239     """
240 
241     return [(df.label == cls).sum() for cls in range(nr_labels)]
242 
243 
244 def distribute_data(df, distribution):
245     """ Removes samples from a dataset such that the distribution
246     over classes mirrors the given distribution.
247 
```

```
248     :param df: The dataframe
249     :param distribution: An array of floats encoding the distribution
250     we want to acheive. The float at index i represents the ratio of
251     class i samples.
252     """
253
254     distribution = np.array(distribution)
255
256     # calculate parameters
257     me_vs_hw_db_ratio = distribution / distribution[0]
258     dataset_nr_samples = count_records(df, len(distribution))
259     data_set_distribution = \
260         dataset_nr_samples / np.sum(dataset_nr_samples)
261
262     # get coefficients
263     coefficients = [_get_coefficient(data_set_distribution[cls],
264         me_vs_hw_db_ratio[cls], data_set_distribution[0])
265         for cls in range(1, len(distribution))]
266
267     # we cannot add new samples, so all coefficient must be below 1
268     coefficients = \
269         [coef if coef < 1.0 else 1.0 for coef in coefficients]
270
271     # modify dataframe
272     for cls, coef in enumerate(coefficients, 1):
273         df = drop_rows(df, cls, coef)
274
275     return df
276
277
278 def plot_feature_vs_classes(df, nr_rows=1, nr_columns=1,
279         subplot_index=1, feature='mVehicleSpeed'):
280     """ Plots a diagram of a subplot with a given feature and the
281     class.
282
283     :param df: The dataframe containing the data.
284     :param nr_rows: The number of rows in this set of subplots.
285     :param nr_columns: The number of columns in this set of subplots.
286     :param subplot_index: The index of this subplot.
287     :param feature: The feature to plot.
288     :return: The ax object of the subplot
289     """
290
291     FEATURE_LINE_STYLE = 'k-'
292     LABEL_LINE_STYLES = ['b--', 'g:', 'r-.', 'y--', 'm:', 'c-.']
293
294     nr_labels = len(df.label.unique())
295
296     # create subplot
297     ax = plt.subplot(nr_rows, nr_columns, subplot_index)
298     trip_id = str(df.trip_id.iloc[0])
```

```
299     ax.set_title(trip_id + ' - ' + feature)
300 
301     # plot feature
302     class_line_height = df[feature].max()
303     ax.plot(df[feature], FEATURE_LINE_STYLE, label=feature)
304 
305     # plot classes
306     for cls in range(nr_labels):
307         ax.plot((df.label == cls)*class_line_height,
308             LABEL_LINE_STYLES[cls], label=str(cls))
309 
310     ax.legend()
311 
312     return ax
313 
314 
315 def plot_feature_vs_class_set(dfs, limit=None, figsize=None):
316     """ Plots a subplot diagram with a given feature and the class
317     for a set of data frames. """
318 
319     if not limit:
320         limit = len(dfs)
321 
322     # set defualt figure size
323     if not figsize:
324         figsize = (20, limit*4)
325 
326     figure = plt.figure(figsize=figsize)
327 
328     # plot subplots
329     for i, df in enumerate(dfs, 1):
330         if i > limit:
331             break
332         plot_feature_vs_classes(df, nr_rows=limit, subplot_index=i)
333 
334     plt.show()
335 
336 
337 def get_nr_labels(df, label_column='label'):
338     """ Counts the number of different labels in a data frame. """
339 
340     return len(df.label.unique())
341 
342 
343 def get_accuracy(training_set, test_set, feature_names,
344         classifier, logger=None):
345     """ Trains a Logistic Regression model and returns the overall
346     prediction accuracy and the accuracy per class. """
347 
348     # scale data
349     scaler = sklearn.preprocessing.StandardScaler()
```

```
350     X_train = scaler.fit_transform(training_set[feature_names])
351     y_train = training_set.label
352     X_test = scaler.transform(test_set[feature_names])
353     y_test = test_set.label
354 
355     # train model
356     print('Model - Fitting...')
357     classifier.fit(X_train, y_train)
358     print('Model - Scoring...')
359     accuracy = classifier.score(X_test, y_test)
360     print('Model - Done!')
361     confusion_matrix = sklearn.metrics.confusion_matrix(
362         y_test, classifier.predict(X_test))
363     accuracies_per_class = confusion_matrix / np.sum(
364         confusion_matrix, axis=1)[:,None]
365 
366     if logger:
367         logger.debug('Accuracy: {:.4f}'.format(accuracy))
368         logger.debug('Confusion matrix: \n{}'.format(
369             accuracies_per_class))
370     return accuracy, accuracies_per_class
371 
372 
373 def get_accuracy_cv(folds, feature_names, classifier, logger=None):
374     """ Runs crossvalidation on a dataset.
375 
376     :param folds: A dataset divided in a set of folds. Each fold is a
377     dict of {filename:dataframe} items.
378     """
379 
380     accuracies = []
381     accuracies_per_class = []
382 
383     if logger:
384         logger.debug('\nStarting {}-fold crossvalidation'
385             'for features:\n{}'.format(
386                 len(folds), '\n'.join(feature_names)))
387 
388     for i in range(len(folds)):
389         if logger:
390             logger.debug('Iteration {}...'.format(i))
391 
392         test_set_dict = folds[i]
393         test_set = pd.concat(
394             test_set_dict.values(), ignore_index=True)
395         test_set = test_set.dropna(subset=feature_names + ['label'])
396 
397         training_set_dicts = \
398             [folds[j] for j in range(len(folds)) if j != i]
399         training_set_dict = {file_name: data_frame \
400             for d in training_set_dicts \
```

```
401             for file_name, data_frame in d.items()}
402         training_set = pd.concat(
403             training_set_dict.values(), ignore_index=True)
404         training_set = \
405             training_set.dropna(subset=feature_names + ['label'])
406         if logger:
407             logger.debug('Training (1) and Test set (2) records')
408             logger.info(get_nr_records_str(training_set, 'label'))
409             logger.info(get_nr_records_str(test_set, 'label'))
410 
411         accuracy, accuracy_per_class = get_accuracy(training_set,
412             test_set, feature_names, classifier, logger)
413         accuracies.append(accuracy)
414         accuracies_per_class.append(accuracy_per_class)
415 
416     mean_accuracy = np.mean(accuracies)
417     mean_accuracy_per_class = np.sum(
418         accuracies_per_class, axis=0) / len(accuracies_per_class)
419 
420     if logger:
421         logger.info('CV accuracy: {}'.format(mean_accuracy))
422         logger.info('CV accuracy per class: \n{}'.format(
423             mean_accuracy_per_class))
424 
425     return mean_accuracy, mean_accuracy_per_class
426 
427 
428 def get_class_accuracies(conf_matrices):
429     """ Returns a list of the elements in the diagonal in the given
430     matrix. """
431 
432     return [[m[i][i] for m in conf_matrices] for i in range(3)]
433 
434 
435 def prepare_dict(df_dict, original_sample_rate, downsample_rate,
436         filter_rate, vehicle_data_params, features, window_size, fc):
437     """ Does some pre-processing on a set of trips.
438 
439     :param df_dict: A dictionary of <filename:dataframe> items. Each
440     dataframe in the dictionary will be processed.
441     :param original_sample_rate: The rate at which the data is
442     sampled in the database.
443     :param downsample_rate: We will keep every nth sample (keeping
444     the averge of the dropped samples), where n is given by the
445     downsample_rate parameter.
446     :param filter_rate: We will keep every nth sample (dropping the
447     rest of the samples), where n is given by the filter_rate
448     parameter.
449     :param vehicle_data_params: The original vehicle data signals
450     that are used to create the features.
451     :param features: The features we are using.
```

```
452     :param window_size: The size of the sliding window we are
453     aggregating features over
454     :param fc: A boolean specifying if we are solving the city or the
455     functional class problem.
456     :return: The processed dataframe
457     """
458
459     prepared_dict = dict()
460     for file, df in df_dict.items():
461         df = downsample(
462             df, downsample_rate, columns=vehicle_data_params)
463         sample_rate = original_sample_rate / downsample_rate
464         df = create_features(df, sample_rate, window_size)
465         create_labels(df, fc)
466         df = filter_nth(df, filter_rate)
467         df = df.dropna(subset=features + ['label'])
468         prepared_dict[file] = df
469
470     return prepared_dict
471
472
473 def create_labels(df, fc):
474     """ Adds a label column to a data frame.
475     :param df: The dataframe
476     :param fc: A boolean specifying if we are labeling the fc or the
477     functional class problem.
478     """
479
480     if fc:
481         df.ix[df.mIsMotorway_MAP == 1, 'label'] = 0
482         df.ix[df.mGetFC_MAP == 1, 'label'] = 0
483         df.ix[df.mGetFC_MAP == 2, 'label'] = 1
484         df.ix[df.mGetFC_MAP == 3, 'label'] = 1
485         df.ix[df.mGetFC_MAP == 4, 'label'] = 2
486     else:
487         df.ix[df.mIsInCity_MAP == 0, 'label'] = 0
488         df.ix[df.mIsInCity_MAP == 1, 'label'] = 1
489
490
491 def create_features(df, sample_rate, window_size):
492     """ Adds features to a data frame. Applies the functions of the
493     features.
494
495     :param df: The dataframe
496     :param sample_rate: The rate the data is sampled at
497     :param window_size: The size of the sliding window we are
498     aggregating features over
499     :return: The processed dataframe
500     """
501
502     # Steering wheel is negative when turning one direction and
```

```
503     # positive when turning the other direction. We are not
504     # interested in the direction, but only the absolute value
505     df.SteeringWheelAngle = df.SteeringWheelAngle.abs()
506 
507     # Direction indicator is encoded with 1 for one direction and 2
508     # for the other direction. We are only interested in whether the
509     # direction indicator is active or not.
510     df.mDirInd = df.mDirInd > 0
511 
512     # create features
513     df['acc_var'] = pd.rolling_var(df.mAccelPedalPos,
514         nr_window_records(window_size, sample_rate), center=True)
515     df['acc_mean'] = pd.rolling_mean(df.mAccelPedalPos,
516         nr_window_records(window_size, sample_rate), center=True)
517     df['dir_mean'] = pd.rolling_mean(df.mDirInd,
518         nr_window_records(window_size, sample_rate), center=True)
519     df['dir_var'] = pd.rolling_var(df.mDirInd,
520         nr_window_records(window_size, sample_rate), center=True)
521     df['gear_var'] = pd.rolling_var(df.mGearSelected,
522         nr_window_records(window_size, sample_rate), center=True)
523     df['gear_mean'] = pd.rolling_mean(df.mGearSelected,
524         nr_window_records(window_size, sample_rate), center=True)
525     df['speed_var'] = pd.rolling_var(df.mVehicleSpeed,
526         nr_window_records(window_size, sample_rate), center=True)
527     df['speed_mean'] = pd.rolling_mean(df.mVehicleSpeed,
528         nr_window_records(window_size, sample_rate), center=True)
529     df['steer_mean'] = pd.rolling_mean(df.SteeringWheelAngle,
530         nr_window_records(window_size, sample_rate), center=True)
531     df['steer_var'] = pd.rolling_var(df.SteeringWheelAngle,
532         nr_window_records(window_size, sample_rate), center=True)
533     df['engine_var'] = pd.rolling_var(df.mEngineSpeed,
534         nr_window_records(window_size, sample_rate), center=True)
535     df['engine_mean'] = pd.rolling_mean(df.mEngineSpeed,
536         nr_window_records(window_size, sample_rate), center=True)
537 
538     return df
```