Master of Science in Aerospace Engineering

Embedded Systems Major

Driving Profile Detection and Management

Project Report

Author:

Joseph MSAED

Supervisor:

Mr Thierry LEYDIER

March 31, 2023

Contents

List of Figures

Declaration of Authenticity

This assignment is entirely my own work. Quotations from literature are properly indicated with appropriated references in the text. All literature used in this piece of work is indicated in the bibliography placed at the end. I confirm that no sources have been used other than those stated.

I understand that plagiarism (copy without mentioning the reference) is a serious examinations offense that may result in disciplinary action being taken.

Date	Signature
March 31, 2023	[illegible]

Abstract

The overall behavior of road vehicles depends majorly on their driver's traits. The last several years witnessed digitalization growth and an increase of onboard data collection, which paved the way to Machine Learning applications. In this project, Unsupervised and Supervised learning approaches were investigated in order to detect driving profiles, because in reality, a clustering phase could be used to detect a pattern in an unlabeled data, then after a labeling phase, a classification phase would be mandatory to recognize a certain profile. At the end it was concluded that the Random Forest algorithm combined with Sliding window technique outperformed the other classical Machine Learning algorithms with an accuracy of 88%. In addition, Deep Learning was investigated, leading to a Neural Network with four layers achieving a testing accuracy of 83%.

Keywords: Driving Profile, Data Processing, Machine Learning, Supervised Learning, Clustering, Sliding Window, Deep Learning.

List of Acronyms:

Abbreviation	Definition
ML	Machine Learning
DBA	Driving Behavior Analysis
CAN	Controller Area Network
DS	Data Set
RF	Random Forest
NN	Neural Network
ADAS	Advanced Driver-Assistance Systems

1 Introduction

Driving behavior is a complex concept that represents a sequence of decisions and actions taken by a person while driving a vehicle. This concept can directly impact the car's global functioning, which depends also on the manufacturer's known specifications only. Not only the driver can affect the car performance, in particular energy consumption and gas emissions, but also his comfort and comfort of the passengers, and crucially road safety.
In fact, the World Health Organization (WHO) stated in a global status report on road safety that approximately 1.35 million fatalities occur yearly because of road accidents, in addition to millions more of injuries [1]. They estimated that car accidents are the major cause of death for the young people worldwide.

The recognition of drivers and their behaviors provides important and helpful insights for different stakeholders in the field to potentially find optimal solutions for the problems that are induced by the incertitude and diversity of driving styles.
Recently, driving profile analysis had an increasing potential in numerous fields like vehicle energy management, advanced driver-assistance systems (ADAS), development of automated vehicles, fleet management, intelligent transportation systems. And With this rises the demand of the ability to detect multiple driving profiles.

On the other hand, modern cars contain an increasing number of embedded sensors and processors, which lead to an abundance of data one can collect and use in Real-Time or later for the development of the automotive field. In addition, the field of Artificial Intelligence is advancing rapidly and especially Machine Learning, which relies heavily on data that is overflowing in modern cars. All of that offer a great opportunity in the driving profile detection and analysis domain, that is not fully exploited yet.

2 Context and Key Issues

The main motivation behind this project is that fact that enabling style and profile detection will unlock a potential feature that helps in energy consumption of a vehicle, especially in the age of electric cars where the recharging speed is slow, and the autonomy is a real concern. For example, we can imagine a scenario where a certain driver is aggressive on the pedals, consuming more energy in a result, so the recognition of this driver with this energy consuming style can help adapting his driving style to reach optimum energy consumption. Another application would be the automatic customization of a car depending on who's behind the wheel. Imagine the settings of the seating, air temperature, list of music, recently visited places list, etc., all changing automatically based on the driver recognition.

So the ultimate goal of driver profiling is to be able to implement it in Real-Time on board the car, so the computations and memory costs are major constraints to be considered later on. To make the driver profiling useful and efficient, it must be done by the processors inside the vehicle. In fact, if one is aiming to effectively improve each vehicle's energy management, the global system of driving style recognition and feeding it back to the powertrain must be flexible. Also, managing the Big Data generated from internal sensor might pose a challenge.

Like any other project involving Machine Learning, the preprocessing phase of the raw data can be very challenging, as making the dataset clean before feeding it to the ML model is very critical for a better performance. Plus, the raw data can be so inconsistent and erroneous, causing the detection task to be more contesting. Besides, the driving style can vary for the same driver depending on the external conditions and events like weather, road type, traffic, and even biometric signals of the driver. The challenge here is to process this data or to model those events and merge them with the internal vehicle data to get more reliable results. A simple example would be that a certain driver is cruising slower than usual, not due to the change of his driving style, but since it is raining, and he cannot go faster because of tires aquaplaning. Lastly, every set of input data has its own particularities in machine learning applications, and the pre-processing phase can differ from one set of data to another. Adding to that, some algorithms are more suited to a type of dataset compared to another.

All the motivations and potential applications stated above encouraged the kick-off of this project, which is a first phase towards unlocking all the potential of driver and style detection.

3 Related Works

The literature is full of research hovering around the concept of driving behavior analysis, which includes several tasks like driving style detection, driver identification (driver fingerprinting), driver inattention detection, driving event detection/prediction (two of them are tackled in the scope of this project). And Artificial Intelligence, especially Machine Learning, is a trendy and effective tool to tackle these tasks, as it mentioned heavily in research related to this topic.

Driving style refers to the way a driver operates a vehicle on the road, including their habits, attitudes, and behaviors while driving. It can be influenced by a variety of factors, such as the driver's personality, experience, skills, age, gender, the type of vehicle they are driving, and the road and weather conditions. The recognition and analyze of driving styles is proven that it has the potential to assist the drivers in managing their energy consumption. For instance, Yang et al. were able to improve energy consumption in hybrid vehicles by 7% and 5% for aggressive and normal drivers respectively [2]. Also, Magaña and Muñoz-Organero [3] introduced a driving assistant that gives recommendations for fuel saving based on customized drivers' styles. The system analyzes external conditions, and whether the driver's style is eco-friendly or not, and suggests therefore an optimal average speed. The study showed that this system can help to reduce fuel consumption by up to 11%.

On the other hand, the other DBA task that is tackled in the scope of this project is driver fingerprinting. It is simply guessing who is the person behind the wheel and labeling him with a unique identifier. Martinez et al. [4] proposed an architecture aiming to improve the security of ADAS systems by identifying in real-time drivers and detecting imposters. This method used a feedforward neural network data from CAN-bus and IMU accelerometers.

Several approaches and algorithms were implemented to recognize a driver or detect his driving style.
The simplest methodology has to be the Rule-based algorithms. It defines rules depending on fixed thresholds over an observed set of data. Murphey et al. [5] used this approach and determined the driving style by counting the amount of aggressive maneuvers. Similarly, Stoichkov [6] utilized this algorithm to detect 6 driving events. The rule-based algorithms showed poor accuracy mainly because of its limited number of parameters, showing inability to handle complex problem like ML.
Under the same approach, Fuzzy Logic maps were implemented, allowing to incorporate more parameters, and resulting in more robustness. This algorithm implemented by Syed et al. [7] estimated the ideal throttle and brake pedals operation and provided the driver haptic feedbacks. Note that in rule-based and fuzzy logic algorithms, there is a strong correlation between the quality of the classification and the thresholds choice, and a limit in the amount of data that can be processed.

Machine Learning algorithms take the biggest share of models in the DBA scheme. These algorithms can be grouped into different families.
Li Guofa et al. [8] adopted Random Forest, a well known algorithm in the Ensemble methods family, to

classify driving styles using selected features. Also, the latter algorithm was used in detecting driving events (Das et al. [9]), and drivers themselves. In fact, Chowdhury et al. [10] recognized drivers and reached the best results by applying Random Forest on smartphone's GPS data only. In addition to RF, other ensemble algorithms are present in the literature, like Gradient Boosting and Adaboost.
In the Instance-based family, the most promising algorithm is support vector machine (SVM). It was implemented by Wang et al. [11] in his semi supervised approach to detect styles, and by Moreira-Matias et al. [12] to recognize who is behind the wheel.
Decision tree algorithms were not mentioned a lot despite their easy implementation, due to their limitations in terms of features used.

Also, Deep Neural Networks, a subclass of ML models, have their fair share in the literature. Although it is computationally more expensive compared to classical ML algorithms, but they are better at extracting hidden features and critically temporal dependencies. Convolutional Neural Nets (CNN) have shown good results in time series classification, for the purpose of driving style detection (Bejani et al. [13]), and driver fingerprinting (Jeong et al. [14]).

All the methods above can be categorized as Supervised ML algorithms. However, Unsupervised ML algorithms were also used to categorize driving styles. In fact, Gace et al. [15] relied on k-means clustering to identify three types of trips using automotive and traffic context data sets. Another proposed unsupervised learning strategy to categorize 5 styles included Principal Component Analysis and Hierarchical Cluster Analysis with time-series motion data (Constantinescu et al. [16]). Feng et al. [17] classified robustly driving styles using Support Vector Clustering method. In this article, they were able to differentiate the variations in styles of the same driver and provided an objective classification.

Another implementation is to apply both Unsupervised and Supervised learning successively; unsupervised to specify the most efficient classification strategy and supervised learning to get better categorizing results. For example, Van Ly et al. [18] applied K-means (unsupervised) and Supported Vector Machine (supervised) for the classification.

4 Aims and Objectives

This project is considered as a first step towards unlocking all the potential application for recognizing drivers, and their driving behaviors. So the goal here is to detect who is behind the wheel using Machine Learning algorithms and try to find the best possible solution. To enable this task, lots of data are needed, the choice was to utilize data collected from the CAN-bus of the vehicle, since it is the most common present data, and it doesn't rely on external devices like smartphones.
Therefore, the first step is to acquire the sensors' data and perform pre-processing and features extractions, in order to clean it and make the most out of it before feeding it to the ML algorithms that rely heavily on the quality of the dataset in hand.
The second step is to apply an Unsupervised Learning approach as a way to detect some patterns in the dataset in hand, and then detect a particular style for each driver. Afterward, in a third phase, supervised learning algorithms are implemented aiming at identifying the driver. In this third step, classical ML techniques will be applied as well as Deep Learning and its Neural Networks modelling. All of this to find the best approach possible, giving the best results. Note that the second and third steps, unsupervised and supervised approaches respectively, are implemented using different datasets.

Semester 2: Unsupervised Learning Approach

5 Machine Learning Types

Since this project is going to evolve around utilizing some Machine Learning techniques, the first step was to familiarize with this vast environment. Machine learning is a subfield of Artificial Intelligence (AI) that involves the development of algorithms and statistical models that enable computers to automatically learn and improve from experience, without being explicitly programmed. The primary goal of machine learning is to enable machines to identify patterns and make predictions or decisions based on data inputs.

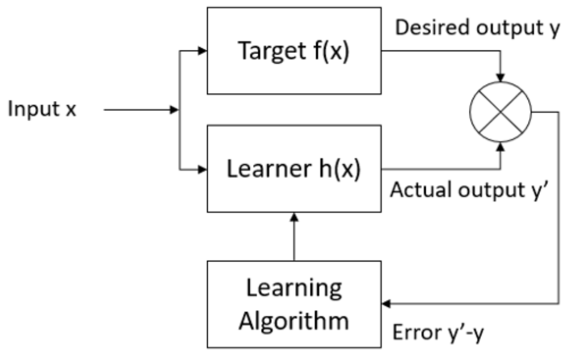


Figure 1: General formulation of ML

There are four main types of machine learning algorithms:

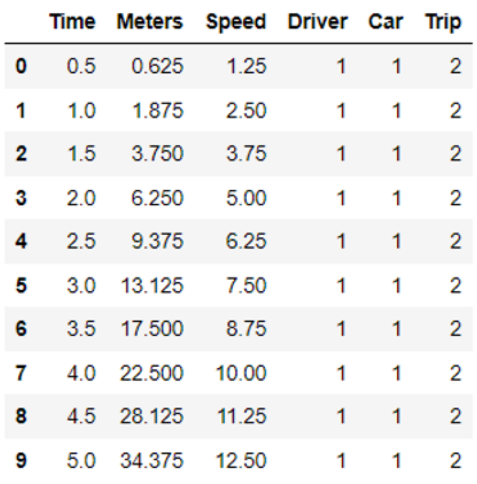
- **Supervised Learning:** it is the most common type of machine learning. During the training phase, the training set, containing the inputs, is labelled with the desired output. Also, there is a strong learning signal here.
- **Unsupervised Learning:** contrary to the supervised learning, the training sets are not labelled, leaving the system searching for common traits between them, and evolving based on internal knowledge.
- **Semi-supervised Learning:** as its name suggests, it combines a large amount of unlabeled training data sets and a small amount of labelled data.
- **Reinforcement Learning:** in this paradigm, there is a learner called agent, and the goal is to take a correct "action" for each environment "situation". In fact, the environment will reveal itself to the agent in the form of states, and the agent will have influence on the environment by taking actions, and then agent receive a "reward" for a correct combination of (state, action) before the next state.

Besides, one should not forget about a major subfield of ML called Deep Learning. It is focused on developing algorithms inspired by the structure and function of the human brain, known as artificial neural networks. These algorithms are designed to analyze and interpret complex data sets, such as images, videos, and text, and extract useful features and patterns that can be used for various tasks, including classification, regression, and clustering. Thus, neural networks can handle both supervised and unsupervised learning.

6 Dataset Exploration

Data is the holy grail of Machine Learning, without it, it is impossible to train models and get results. That is why is it critical to first analyze the presented data and understand what information can be extracted from it.

During the first period of the project, a set of data was provided (it will be referenced as **DS1** from now on). The dataset DS1 was given in a 'csv' format, which can be easily imported to a Python environment using the *Pandas* library. *Pandas* is a fast and powerful data analysis and manipulation library integrated in Python. DS1 contains 511,595 different data points divided into 6 columns, with a sampling rate of $2\ Hz$ as seen in the figure below.



	Time	Meters	Speed	Driver	Car	Trip
0	0.5	0.625	1.25	1	1	2
1	1.0	1.875	2.50	1	1	2
2	1.5	3.750	3.75	1	1	2
3	2.0	6.250	5.00	1	1	2
4	2.5	9.375	6.25	1	1	2
5	3.0	13.125	7.50	1	1	2
6	3.5	17.500	8.75	1	1	2
7	4.0	22.500	10.00	1	1	2
8	4.5	28.125	11.25	1	1	2
9	5.0	34.375	12.50	1	1	2

Figure 2: A sample of the first 10 elements of the provided dataset.

In DS1, the sampling period is constant and equal to 0.5 seconds. It contains only one metric, which is the vehicle's velocity in m/s. Also, each data point is linked to a $Driver$, Car and a $Trip$. We can notice that when a certain path is over (driver reaches his destination for example), $Time$ and $Meters$ columns are set back to 0.

In total, there are 8 drivers (labeled from 1 to 8), 4 cars, and 2 trips. The car 1 is driven by D1 and D5, car 2 by D2 and D6, car 3 by D3 and D7, and car 4 by D4 and D8. Also, note that all drivers take both trips, which could be interpreted as two different trajectories.

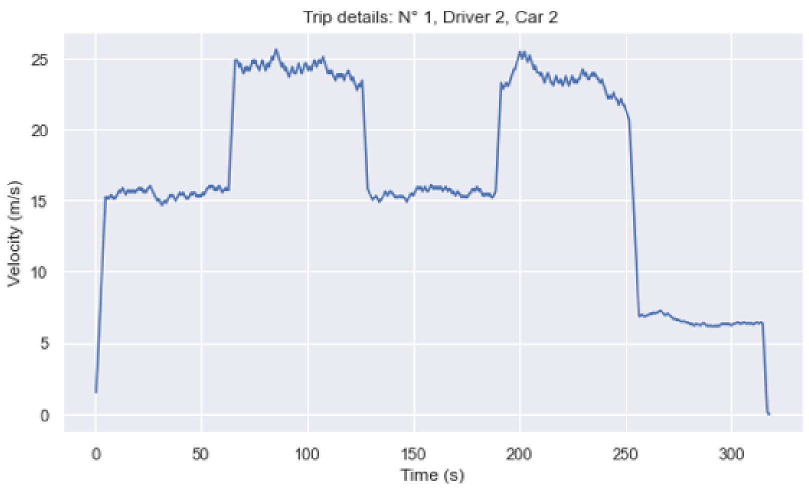


Figure 3: Velocity vs time plot of one trajectory of Trip 1 taken by Driver 2.

Figure 3 shows one sample of the speed trend in one trip. For this trajectory, most of the time the velocity gradient is small, but alters significantly in six occasions.

7 Dataset Expansion

Due to the fact that the current dataset only contains one feature, in this section the goal is to extract more hidden information present inside the dataset. The first that comes to mind is to calculate the instantaneous acceleration at every time step. It is well known as the first derivative of the velocity, expressed in m/s^2. The following formula was used to create an 'Acceleration' column:

$$a_{T_i} = \frac{V_{T_i} - V_{T_{i-1}}}{T_i - T_{i-1}}$$

Then, positive and negative accelerations were separated to two different columns to differentiate between applying the gas and the brakes pedal.

In addition to the acceleration, one can also extract the Jerk at every point, which is the rate of change of the acceleration (the second derivative of the velocity) expressed in m/s^3. It was used by some researchers [5] to analyze the driver's behavior.

$$Jerk_{T_i} = \frac{a_{T_i} - a_{T_{i-1}}}{T_i - T_{i-1}}$$

8 Time Series Segmentation

There are now 5 time series (Velocity, positive and negative acceleration, positive and negative jerk). It is illogical to set the whole driver path as one sample, because firstly it is too large, and secondly if the chronology of braking changes a bit, the time series will be classified differently, which is irrelevant. Hence, the obvious idea was to split the time series, where the Car and $Trip$ numbers are the same, into smaller chunks based on a criterion.

The main segmentation algorithms are: [19]

- The Sliding Window Algorithm
- The Top-Down Algorithm
- The Bottom-Up Algorithm

The use of the Sliding Window algorithm was investigated. In fact, this technique requires choosing a fixed size window and an overlap percentage. Sliding window alone with its fixed frame size as hyperparameter could be a potential solution.
But the idea of time-series segmentation base on detected driving events (inspired by [17]) seemed a more interesting solution at this stage.

9 Events Detection

To help in detecting the longitudinal driving events during a cruise, the speed variance was used. For every data point (except the first ones, since variance is chosen to be calculated for a duration of 4 seconds). Pandas' $rolling()$ and $var()$ functions were used with a frame size equal to 8 (4 seconds), and the variance threshold was chosen to be equal to 1 after several trials.

```
df.loc[df['T'] == df['T'].shift(minperiods-1),
'Speed_var'] = df['Speed'].rolling(min_periods = minperiods,
                                  window = window_size,
                                  center = True).var()
```

The second step in the section was to define the longitudinal events that can occur during a cruise:

- **Accelerating:**
  - if the acceleration is positive and the speed variance is above the threshold.
  - if the acceleration is positive for 2 seconds or more, no matter what the speed variance is.
- **Braking:**
  - if the acceleration is negative and the speed variance is above the threshold.
  - if the acceleration is negative for 2 seconds or more no matter what the speed variance is.
- **Stopping:**
  - if the velocity is equal to zero.
- **Maintaining speed:**
  - if the speed variance is below the chosen threshold.

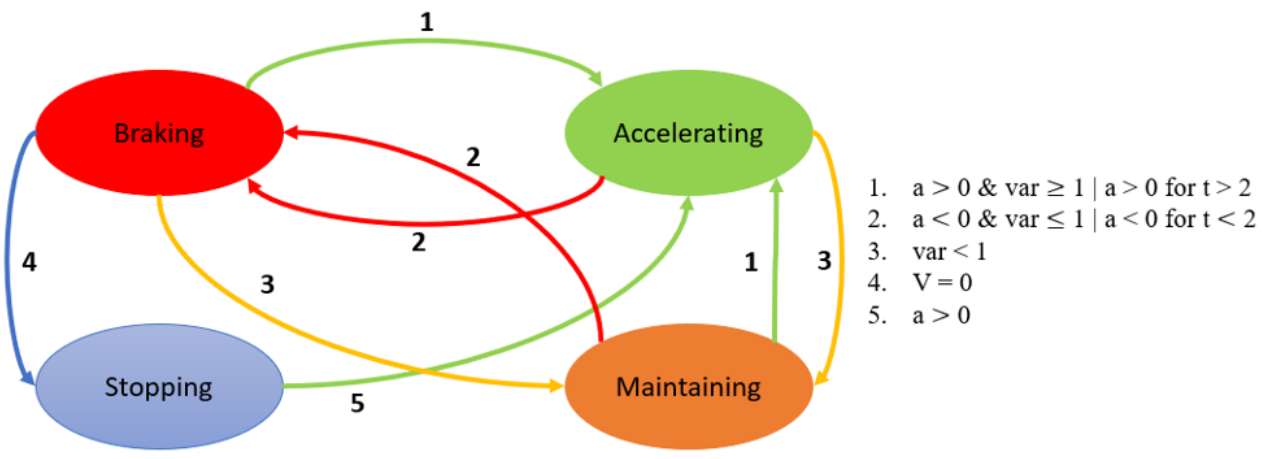
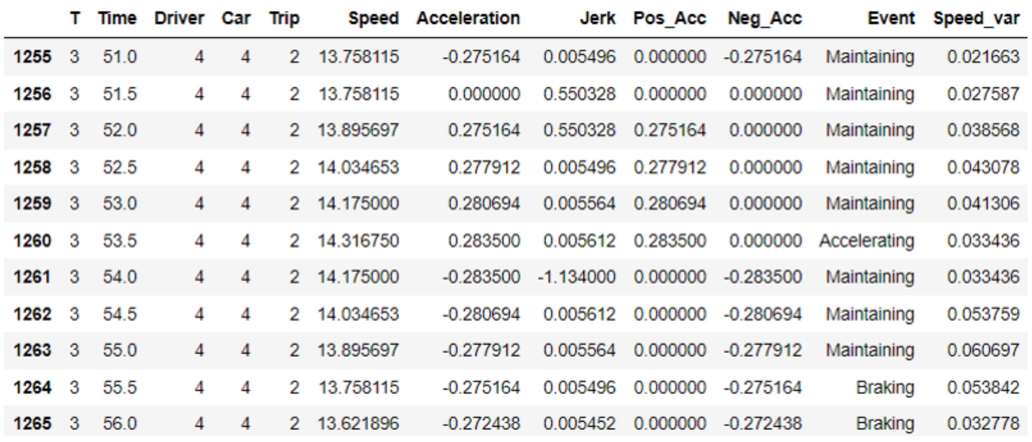

Braking
Accelerating
Stopping
Maintaining
1
2
2
3
4
1
3
5
1. a > 0 & var ≥ 1 | a > 0 for t > 2
2. a < 0 & var ≤ 1 | a < 0 for t < 2
3. var < 1
4. V = 0
5. a > 0

Figure 4: Schematic showing the transitions between events.



	T	Time	Driver	Car	Trip	Speed	Acceleration	Jerk	Pos_Acc	Neg_Acc	Event	Speed_var
1255	3	51.0	4	4	2	13.758115	-0.275164	0.005496	0.000000	-0.275164	Maintaining	0.021663
1256	3	51.5	4	4	2	13.758115	0.000000	0.550328	0.000000	0.000000	Maintaining	0.027587
1257	3	52.0	4	4	2	13.895697	0.275164	0.550328	0.275164	0.000000	Maintaining	0.038568
1258	3	52.5	4	4	2	14.034653	0.277912	0.005496	0.277912	0.000000	Maintaining	0.043078
1259	3	53.0	4	4	2	14.175000	0.280694	0.005564	0.280694	0.000000	Maintaining	0.041306
1260	3	53.5	4	4	2	14.316750	0.283500	0.005612	0.283500	0.000000	Accelerating	0.033436
1261	3	54.0	4	4	2	14.175000	-0.283500	-1.134000	0.000000	-0.283500	Maintaining	0.033436
1262	3	54.5	4	4	2	14.034653	-0.280694	0.005612	0.000000	-0.280694	Maintaining	0.053759
1263	3	55.0	4	4	2	13.895697	-0.277912	0.005564	0.000000	-0.277912	Maintaining	0.060697
1264	3	55.5	4	4	2	13.758115	-0.275164	0.005496	0.000000	-0.275164	Braking	0.053842
1265	3	56.0	4	4	2	13.621896	-0.272438	0.005452	0.000000	-0.272438	Braking	0.032778

Figure 5: A chunk example of the current data frame with events assigned to every point.

10 Feature Extraction

10.1 Features Selection and Calculation

The idea was to extract features for every detected event segment. Note that some events would last more than the others. Thus, four statistical features were calculated for the speed, acceleration (positive and negative), and jerk along each event duration:

- **Mean**
- **Standard deviation**: the spread of a group of numbers from the mean (square root of the variance).
- **Maximum value**
- **Minimum value**

Instead of iterating over the all the rows and performing the calculations where the event is the same, the efficiency and strength of *Pandas* was exploited. Firstly, a column called *evchange* was add, and every time there is a change in an event, its value inside the current row is incremented by 1:

```
df['evchange'] = df['Event'] != df['Event'].shift()
df['evchange'] = df['evchange'].cumsum() - 1 # to enumerate from 0
```

Secondly, the four parameters were calculated for every event segment very rapidly and in four lines of code using *groupby* function:

```
df_mean = df.groupby(['evchange','Event']).mean()
df_sd = df.groupby(['evchange','Event']).std()
df_max = df.groupby(['evchange','Event']).max()
df_min = df.groupby(['evchange','Event']).min()
```

Finally, the four new dataframes were concatenated into one consisted of 19 columns and, 30834 rows (the total number of event change in the dataset).

10.2 Features Analysis

When preparing features to be fed to a clustering algorithm, one must think of the 'Collinearity' problem. It is defined as a high level of correlation between two variables. In fact, when two variables collinear, they correspond to technically the same concept. The latter is now represented twice is the data, therefore gets double the weight of the other features, so the final results will most likely be asymmetrical in the direction of the double weighted feature.

As you can see in Figure 35 (appendix 26.1), we are not particularly interested by the rectangle where the color is light (positive correlation), or dark (inversely proportional). Instead, we are more interested in using features that have a correlation close to zero.
There are several methods to overcome this inconvenience:

- Variable elimination
- Factor analysis
- Variable index

Luckily, there is a way to solve this dilemma using the Principal Component Analysis method, which can be imported from 'Sklearn' Python library.

11 Principal Component Analysis (PCA)

It is a statistical technique whose main goal is to convert high dimensional data to lower dimensional data by identifying the features that contain the maximum information about the dataset. PCA combines the input variables in a way and eliminates the least important variables while maintaining precious parts of all the variables. In addition, it is a projection based method which transforms the data by projecting it onto a set of orthogonal axes [17].

The features are selected based on variance that they cause on the output, so the first component will be the feature that causes the highest variance, and so on. Note that the resulting principal components won't be correlated.

To conclude, PCA regroups input features in a manner to preserve the most valuable information and drops the least important features to form linearly independent PCA components.

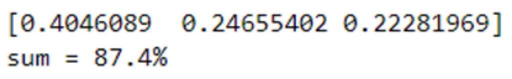
One can summarize several advantages of such Dimensionality Reduction method:

- Easier visualization of higher dimension data with 2D or 3D plots (reduction to 2 and 3 components respectively).
- Reduction of required time and memory storage.
- Removal of the multi-collinearity.
- Lower threat of overfitting (the more the features, the higher the risk of overfitting [20]).

The breakdown of PCA can be found in the Appendix 26.2.

It was interesting to be able to see a representation of the data, which is one of the reasons to perform PCA. Thus, we set the number of components to three, and we checked if it is acceptable using the Explained Variance.
The Explained Variance Ratio informs about the percentage of variance (i.e. information) corresponding to the principal components. To avoid the overfitting of the model, ideally the total variance of the components must be higher than 80%.

```python
print(pca.explained_variance_ratio_)
print("sum = {:.1f}%".format(100*sum(pca.explained_variance_ratio_)))
```



```
[0.4046089  0.24655402 0.22281969]
sum = 87.4%
```

Figure 6: Execution of the previous code.

For 3 principal components, 87.4% of the information is conserved, which is sufficient at an initial stage. As a result, 3 new uncorrelated features are now ready for some unsupervised learning.

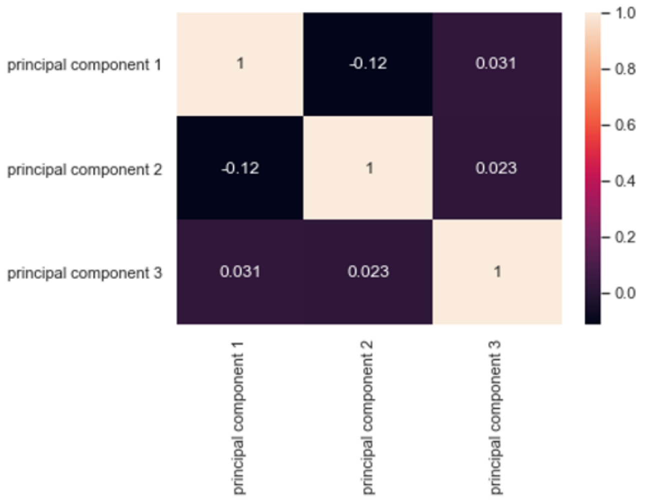
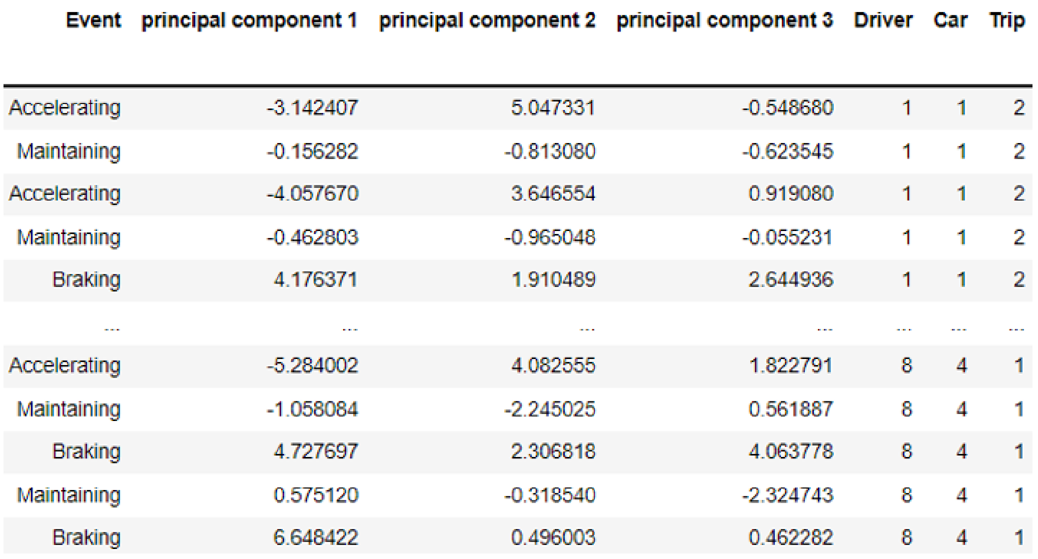


Figure 7: Correlation matrix of the new variables after PCA.

A new data frame was constructed with the new three components as columns:



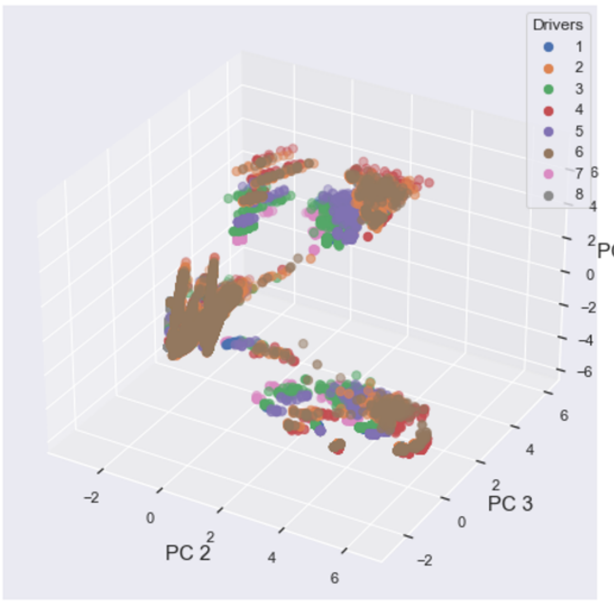
Event	principal component 1	principal component 2	principal component 3	Driver	Car	Trip
Accelerating	-3.142407	5.047331	-0.548680	1	1	2
Maintaining	-0.156282	-0.813080	-0.623545	1	1	2
Accelerating	-4.057670	3.646554	0.919080	1	1	2
Maintaining	-0.462803	-0.965048	-0.055231	1	1	2
Braking	4.176371	1.910489	2.644936	1	1	2
...	...	...	...	...	...	...
Accelerating	-5.284002	4.082555	1.822791	8	4	1
Maintaining	-1.058084	-2.245025	0.561887	8	4	1
Braking	4.727697	2.306818	4.063778	8	4	1
Maintaining	0.575120	-0.318540	-2.324743	8	4	1
Braking	6.648422	0.496003	0.462282	8	4	1

Figure 8: The obtained data frame after PCA.

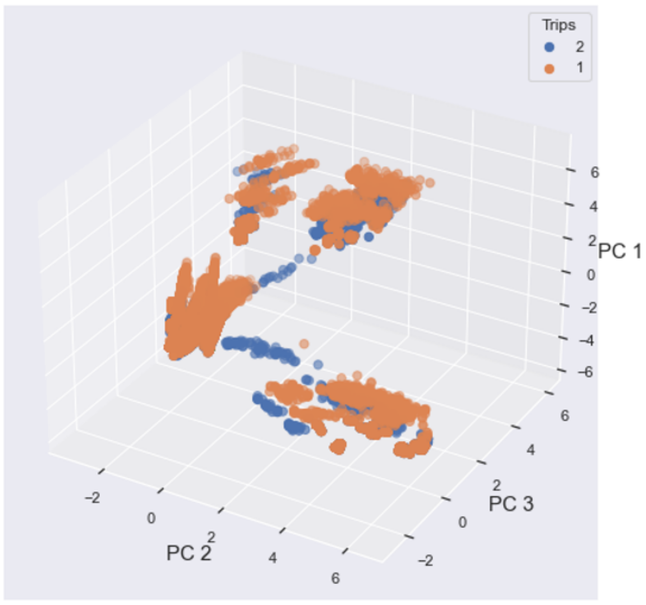
The following step is to detect some patterns, hence identifying the driving styles of the 8 drivers.

12 Data Visualization

DS1 was split into 80% for training set and 20% for the testing set as recommended by many experts.



(a) Drivers distribution.



(b) Trip numbers distribution

Figure 9: The training set visualized in 3D space.

As you can see in Figure 9a, the 8 drivers are distributed on what appearing to be several congestions of points, a first signal that the behavior of one driver can change during the same route.

Also, one can read from Figure 9b that the points are equivalently distributed in both groups. In other words, the behaviors of the drivers do not change drastically based on the trip.

13 Clustering

Clustering is an unsupervised ML approach that combines objects in a way that the ones grouped in the same cluster have similarities [21]. In other words, the data point will have similar aspects to the other points in its cluster, but different traits to the ones in other groups.
Clustering is considered one of the types of data modelling. So, its main objective is to discover structures and patterns in a dataset, which reduces complexity and facilitates interpretation. Note that the measurements of similarities between objects may be distance, correlation, cosine similarity, or something else depending on the context.

There are different types of clustering, such as:

- **Partitioning methods:**
  Techniques that divide the data set into k subsets, where k is a predefined number by the user.
- **Hierarchical clustering:**
  In this technique each is treated as an independent cluster at first. Then, in continuously executes two steps: identify two clusters that are closest together and merge the two most similar ones. The 'loop' breaks when all the clusters are merged.
- **Fuzzy clustering:**
  The main difference between this method and the previous ones is that data points in fuzzy clustering can be present in more than one cluster. To find the optimal location of a data point, it uses the least squares, and hence this location may be in a probability space between several clusters.

- **Model-based clustering:**
  It uses probabilistic distributions to create the clusters. The data is considered coming from a mixture of density, and each cluster is modeled by a Gaussian distribution.

- **Density based clustering:**
  It recognizes clusters based on considering that a cluster in a data space is an adjacent region of high point density and separated by regions of low point density.

13.1 K-means Clustering

It is one of the famous, easy to implement partitioning clustering algorithms. Its objective is to classify the data set naturally into k different clusters [22]. Like any other method, it has some advantages and drawback. Starting with the pros:

- Easy to implement and computationally fast.
- Has a time complexity of $O(n.k.t)$, with n number of data objects, k number of clusters and t the number of iterations in the algorithm (developed below).
- Easy to understand and interpret.
- Produce tighter clusters.

For the disadvantages, here are the main ones:

- The number of clusters k must be chosen in advance, and it is difficult to predict its value.
- Sensitive to scaling.
- The results vary with different representations of the data.

13.1.1 Algorithm

After placing k centroids randomly between the data points as an initialization phase, the k-means algorithm is based on an iteration over two steps until the stopping criterion is met.

1. **Data Assignment:**
   By relying on square Euclidean distance, each data point is assigned to the closest centroid. Each centroid and its assigned points form a cluster.
   Let d be the Euclidean distance defined as:

   $$d(x_i, y_i) = \left[\sum_{i=1}^{n}(x_i - y_i)^2\right]^{1/2}$$

   With x and y two points with n coordinates. Hence, the data assignment is based on the following:

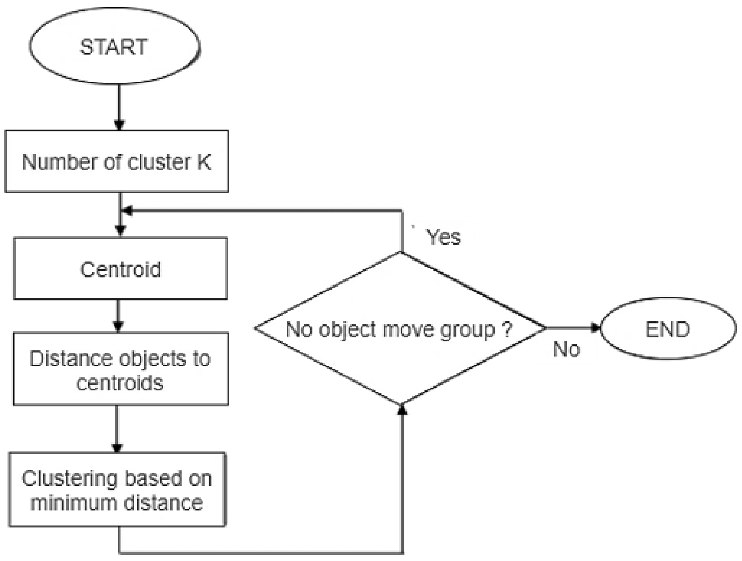
   $$argmin_{c_i \in C}\ d(c_i, x)^2$$

   With C the set containing the k centroids, and x each point in the dataset.

2. **Centroid update:**
   At this stage, each centroid has its assigned points. Therefore, the new centroid of these points is recomputed using the mean of all the present points in the cluster. Let X_i be the set containing the points assigned to the centroid c_i.

   $$c_i = \frac{1}{|X_i|}\sum_{x_i \in X_i} x_i$$

These two steps are repeated until there is no change in the centroids' positions (the stopping criterion is met).


START
Number of cluster K
Centroid
Distance objects to centroids
Clustering based on minimum distance
No object move group ?
Yes
No
END
Figure 10: Flowchart of the K-means clustering algorithm.

13.1.2 Optimal value of K: Elbow Method

Before executing the k-means clustering algorithm, it is essential to decide the optimal number of clusters into which the data may be grouped. The Elbow Method is one of the most well-known methods to determine the optimal value of k that should be chosen before performing the k-means clustering.

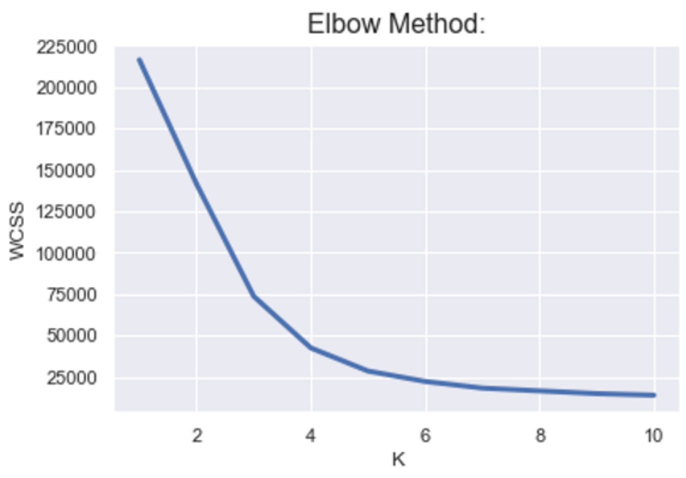


Figure 11: Plot showing the squared error vs the chosen number of clusters.

In this method, k is iterated between 1 and 10. And for every value k, the 'Within-Cluster Sum of Square' is calculated. The latter is the sum of square distance between each point of the cluster and its centroid.

As seen in Figure 11 above, the trend looks like an elbow and the optimal K according to it should be 4. But when combining this method with the Silhouette Score (developed below), one can see that $k = 3$ was the better choice, and it is still in the 'elbow region' of the plot.

13.1.3 Python Code

To perform the k-means method, the 'KMeans' library from 'sklearn.cluster' is imported, then the following code is executed with a chosen number of clusters k equal to three:

```
kmeans = KMeans(n_clusters=3,
                init = 'k-means++',
                random_state = 42)
kmeans.fit(X_train_np)
y_kmeans = kmeans.predict(X_train_np)
```

With *X_train_np* being an array of 3-by-1, the algorithm assigns for each of those arrays, which represents a point in the Principal Components dimension space (in this case 3 dimensions), a cluster number and stores those numbers in *y_kmeans* array.

13.2 Hierarchical Clustering

It was interesting to apply another famous type of clustering which is the Agglomerative Hierarchical Clustering, which is called a bottom-up approach.

13.2.1 Algorithm

At the start, every data point is considered as a cluster on its own. So, there are n clusters if n is the number of points in the dataset. Then, a new cluster is formed by joining the closest two clusters (distance calculation), thus the number of clusters becomes $n-1$. The previous step is repeated until there is one bug cluster.

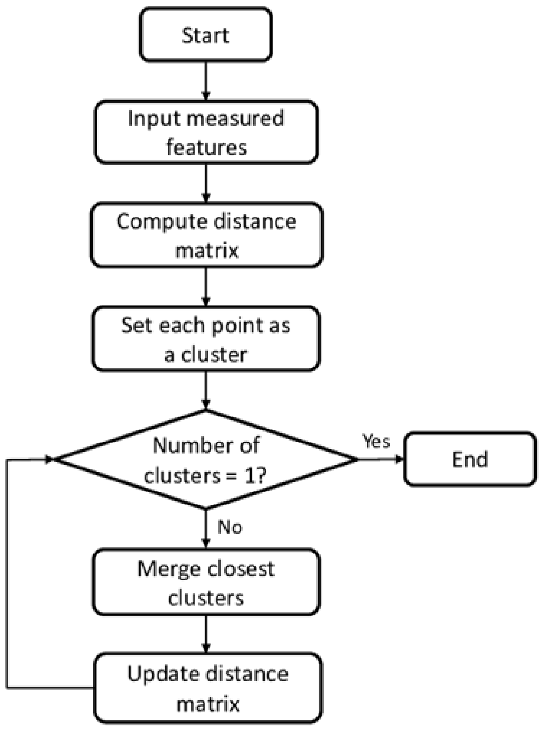


Figure 12: The algorithm of agglomerative hierarchical clustering.

13.2.2 Python Code

```
cluster = AgglomerativeClustering(n_clusters=3,
                affinity = 'euclidean',
                linkage = 'ward')
cluster.fit_predict(X_train_np)
```

The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of clusters that minimize this criterion. ‘ward’ minimizes the variance of the clusters being merged (from Scikit-learn). And the affinity is used to calculate the linkage. Since we are used ‘ward’, there is no choice but to use ‘Euclidean’ affinity.

14 Results and Analysis

14.1 Silhouette Method

The Silhouette score is a cluster validity measure [23]. It measures how similar a data point is to its cluster compared to other clusters. The Silhouette value for each point is defined as follows:

$$S_i = \frac{b_i - a_i}{max(a_i, b_i)}$$

Where a_i measures the similarities of the point i to its own cluster (mean intra cluster distance), and b_i is the distance between the point i and the nearest cluster which it doesn't belong to.

For the Python implementation, the importation of 'silhouette_score' from ‘sklearn.metrics’ is needed. Then, the score is calculated for $k = 2$ to $k = 10$.

```
sil_score = []
for k in range(2, 11):
    kmeans = KMeans(n_clusters = k,
            init = 'k-means++',
            random_state = 42).fit(X_train_np)
    labels = kmeans.labels_
    sil_score.append(silhouette_score(X_train_np, labels, metric = 'euclidean'))
```

Note that the Silhouette coefficient is defined only for $2 < k < n_{samples} - 1$, that's why the iteration starts from 2.

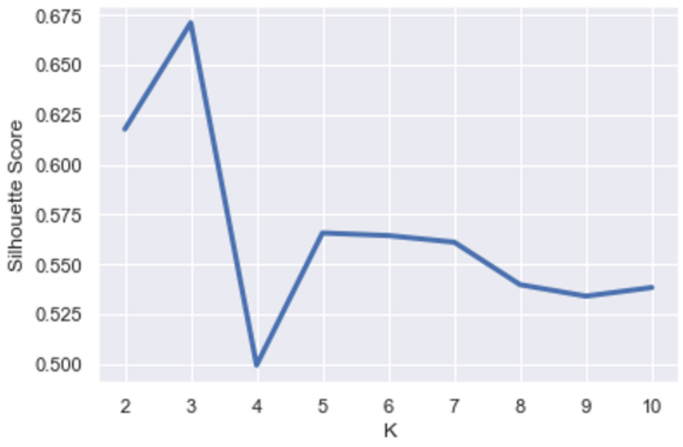


Figure 13: Silhouette Coefficient values for different number of clusters.

It is clear that for this application, that the number of clusters predefined earlier $(= 3)$ was the best choice with the highest score.
One should note that the Silhouette is a validation metric and can be combined with the Elbow Method. A combination of both is considered a strong tool to take more assertive decisions about the K value.

14.2 K-means Clustering Results

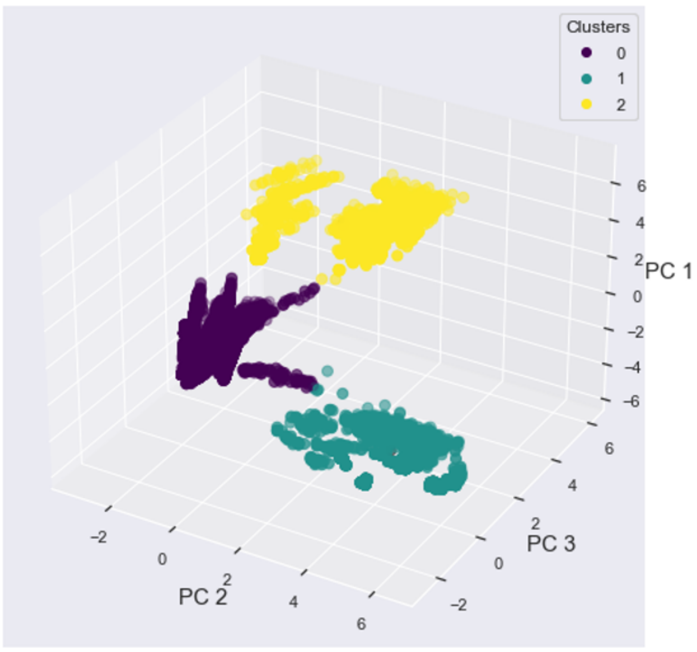
After running the k-means clustering algorithm, with k equals to three as input, the following 3D scatter was obtained with a clear distribution of data points over the three clusters.

The clusters seem fairly distributed on the data points. But the real problem here is that at this point we have no idea what is does every cluster represent in terms of meaning, one of the major disadvantages of the PCA.

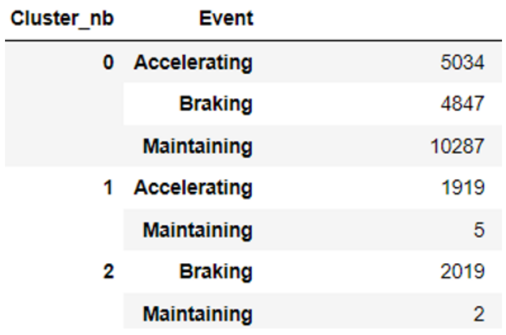
The table in Figure 14 shows the number of data points in each of the clusters. They are grouped by driving events, on which the original dataset was segmented, and the features were extracted.

Note that the points belonging to the $Stop$ event, where the velocity is 0, were taken out of the dataset before performing PCA and the k-means.

The results assert that the algorithm detected a pattern. For example, cluster 2 is a $Braking$ cluster, meaning that it contains data points where there were heavy braking zones. Also, it contains 2 'Maintaining speed' events out of 2021 (0.099%) which are obviously False Negative and do not ideally belong to this cluster.
Similarly, cluster 1 is an $Accelerating$ cluster, signaling heavy acceleration zones with 5 $Maintaining$ False Negative points.



(a) 3D visualization of the clusters.



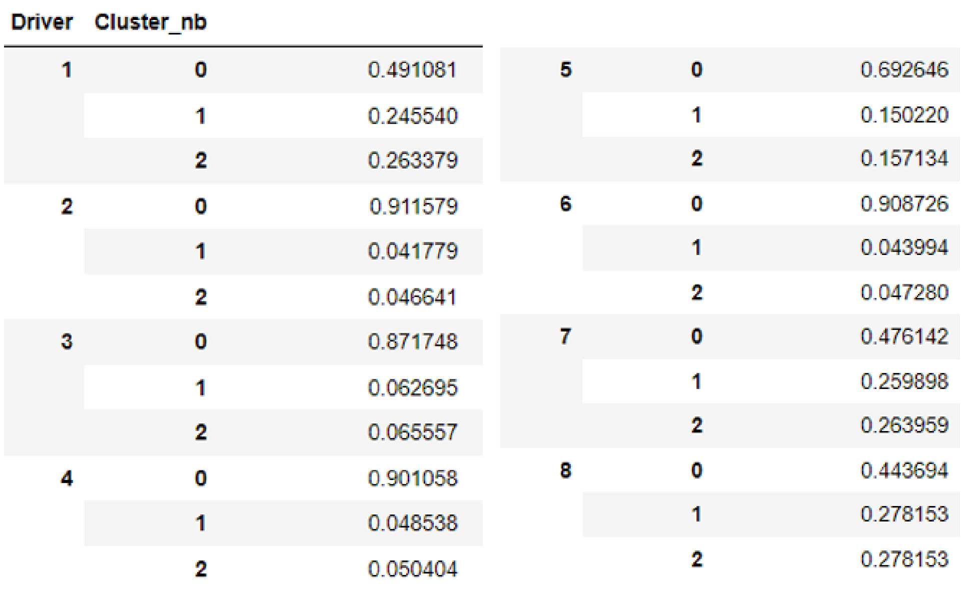
Cluster_nb	Event	
0	Accelerating	5034
	Braking	4847
	Maintaining	10287
1	Accelerating	1919
	Maintaining	5
2	Braking	2019
	Maintaining	2

(b) Distribution of data points belonging to events.

Figure 14: Results of the K-means clustering with k=3.

On the other hand, the cluster 0 is the largest one in terms of points density. It contains in total 20,168 out of 24,113 points (83%), of which the majority are $Maintainingspeed$ zones. This cluster seems to contain all the zones where the driver is either maintaining his velocity or he is doing slight accelerations and braking.

The table in Figure 15 shows the distribution of the 8 drivers on the clusters. As you can see, all drivers are present in all the three clusters, but in different proportions. For example, the driver 1 is 49.1% of the time in cluster 0 (stable speed), 24.5% in the heavy acceleration cluster and 26.3% in the braking zone (cluster 2). Also, we see similar behaviors for the drivers 7 and 8. For those three drivers, approximately half of the time they are maintaining speed, with small accelerations and decelerations, and the other half doing some heavy positive and negative accelerations. Therefore, these three drivers seem to be aggressive, especially if they are compared with driver 2. The latter possesses 91.1% of his proper data points in the 'Maintaining speed' cluster, with the rest distributed equally between clusters 1 and 2 (4.1% and 4.6% respectively). Hence, one can say that D2 is a calm driver, spending most of his cruise time without applying heavily on the acceleration and braking pedals.



Driver	Cluster_nb	
1	0	0.491081
	1	0.245540
	2	0.263379
2	0	0.911579
	1	0.041779
	2	0.046641
3	0	0.871748
	1	0.062695
	2	0.065557
4	0	0.901058
	1	0.048538
	2	0.050404
5	0	0.692646
	1	0.150220
	2	0.157134
6	0	0.908726
	1	0.043994
	2	0.047280
7	0	0.476142
	1	0.259898
	2	0.263959
8	0	0.443694
	1	0.278153
	2	0.278153

Figure 15: Table showing the distribution of the 8 drivers on the different clusters.

Taking a closer look, the drivers can be joined in three groups based on their proportions in each one of the clusters (Figure 15):

- **Group 1:** Contains D1, D7 and D8. It can be labelled as an **Aggressive** group, where the time of maintaining speed does not surpass 50% of the total cruise time.
- **Group 2:** Contains D2, D3, D4 and D6. It can be considered as **Calm** or **Eco-friendly** driving style group, and the percentage of time the driver is not applying heavily on any pedal is more than 85%.
- **Group 3**: Contains D5, who is 69% of the time in cluster 0. So, he can be considered as more aggressive than the ones in group 2 but less than the group 1 drivers. Hence, D5 can easily be classified as a **Normal** Driver.

After this analysis, some subjective thresholds can be assigned to roughly be able to classify new drivers based on their proportion in Cluster 0, where it is dominated by maintaining events, and small accelerations and braking.

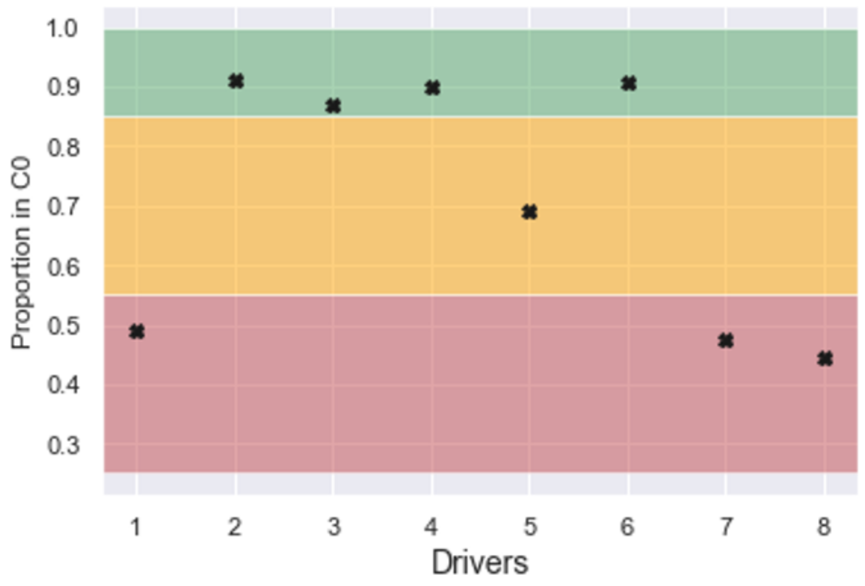


Figure 16: Distributing the drivers on 3 'driving zones': Calm (Green), Normal (Orange), Aggressive (Red).

The chosen 3 zones are as follows:

- 'Calm style': above 85% of belonging to C0.
- 'Normal style': between 55% and 85%.
- 'Aggressive style': below 55%.

The total number of data points belonging to 'Trip1' is larger than the 'Trip2' points. But the two trips are similarly distributed in all the clusters, meaning that their type of path is sort of similar and didn't affect the results.

14.3 Agglomerative Hierarchical Clustering Results

In a hierarchical clustering method, it is a good practice to plot the clustering dendograms to see the distances between clusters (which are equal to the number of data points at the beginning), and thus decide the number of clusters to feed as input to the Agglomerative Clustering model.

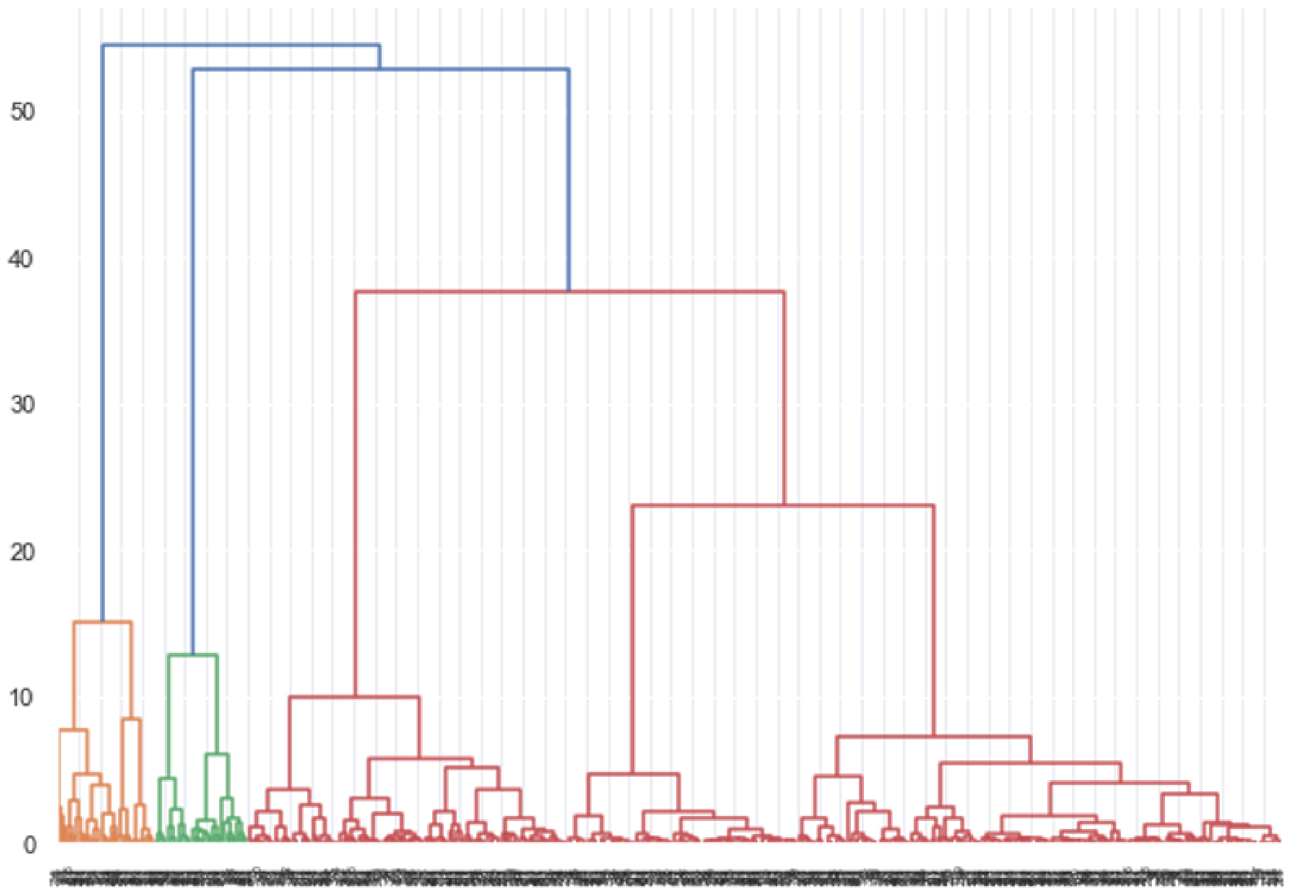
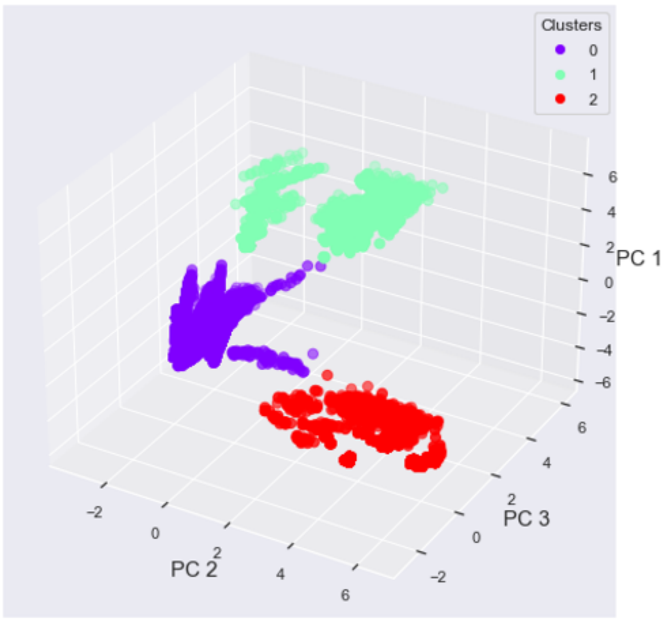
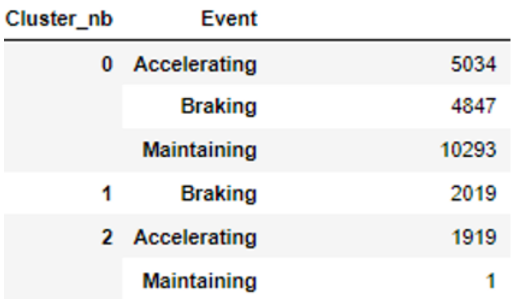


Figure 17: Hierarchical Clustering Dendograms of a chunk of the data.

The vertical height of a dendogram represents the Euclidean distance between two clusters. One can decide three as number of clusters as recommended by the dendograms color distribution, or the choice can be 4 or even 6 clusters. For the comparison to be valid, a number of clusters equal to 3 was chosen.



(a) Obtained clusters distribution



Cluster_nb	Event	
0	Accelerating	5034
	Braking	4847
	Maintaining	10293
1	Braking	2019
2	Accelerating	1919
	Maintaining	1

(b) Events distribution.

Figure 18: Hierarchical clustering results.

The results obtained from the agglomerative clustering are quite similar to the ones obtained from the k-means clustering. But from Figure 18b, one can see that there is only one-off point belonging to the 'Accelerating' and 'Braking' clusters, compared to a total of 7 for k-means.

Also, one should note that the k-means clustering algorithm was significantly faster than the hierarchical clustering. In addition, the Silhouette score of the latter is 0.67 for 3 clusters, very close to k-means' (Figure 13).

Semester 3: Supervised Learning Approach

In this section, the goal shifted from trying to detect a certain particular behavior of a driver into detecting the identity of the person behind the wheel (also known as driver fingerprinting). This was supported with a new labeled dataset, which made the implementation of supervised machine learning possible.

15 Dataset Exploration

The work was based on Ocslab driving dataset that can be found online [24]. This dataset was used for a driver classification challenge track in the 2018 Information Security R&D dataset challenge in South Korea. It consists of 51 variables taken from different sensors on the Controller Area Network (CAN) bus. The retrieval of this dataset was done using the Onboard Diagnostics 2 (OBD-II) and CarbigsPare as OBD-II scanner, at a sampling rate of 1 Hz [25].

The dataset currently in use has the following characteristics:

- A driving time of about 23 hours.
- A driving length of 46 kilometers.
- A dimension of 94380 x 54 (94380 different records and 51 possible features).
- A number of drivers equal to 10 designated from 'A' to 'J'.
- A size of 16.7 MB.

In a first step, after examining all 51 potential features, only 26 that seemed the most useful and representative for the goal of profile detection were selected. The full dataset variables descriptions can be found in Table 5 (appendix 26.3).

The list of all selected features is in Table 6 in appendix 26.4. These features can be classed into 3 categories: Engine, Fuel, and Transmission related.

16 Data Pre-processing

16.1 Sliding Window

The sliding window method is widely used to solve time-series supervised learning problems, by converting the latter into the classical supervised learning problem. So, the main advantage of this method consists of allowing us to apply any classical supervised learning algorithm. The sliding window method build a window classifier h_w, which maps a frame of width w into a single output value [26].

Let's consider $d = (w-1)/2$ as half-length of a frame centered at time c, so for each feature, h_w generates y_c from the frame $[x_c - d, ..., x_c, ..., x_c + d]$. In addition, other parameters will be introduced, which are the stride s and the overlap ratio between 2 consecutive frames r. Thus y is generated every $s = w \times (1-r)$. One cannot conclude about the best values for frame width w and the overlap ratio r, therefore they will be tuned based on the best results later on.

The frame length is chosen at first to be $\mathbf{w = 4}$ seconds and $\mathbf{r = 0.5}$, meaning that the window moves $s = 2$ seconds at each step. And since the sampling rate is 1 Hz, the width is equal to 4 frames and the stride to 2 frames. This choice is represented in Figure 19, where each windowing step is represented by a color.

In conclusion, all the 26 continuous signals go through a framing function with two parameters w and r before being fed to the feature extraction stage.

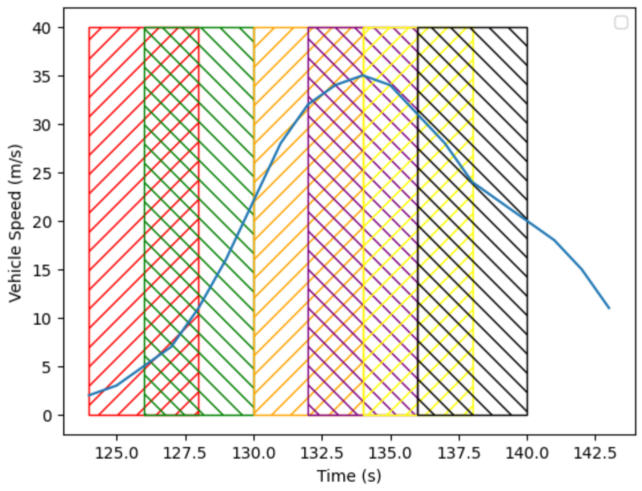


Figure 19: Sliding window illustration on the vehicle speed of driver A.

16.2 Features Extraction

From each small frame, some statistical features were extracted, as they represent the general behavior of each variable along the frame;

- Maximum: $argmax\ x_i \quad for\ i \in [c-d; c+d]$.
- Minimum: $argmin\ x_i \quad for\ i \in [c-d; c+d]$.
- Mean: $(\sum_i x_i)/w$
- Standard deviation: $\sqrt{(\sum_i (x_i - \mu)^2)/w}$
- Median: found by ordering the samples in the frame by ascending order and finding the exact middle. It is a better measure than mean in terms of central tendency of the set.

Hence, from each windowing execution, h_w outputs five features (y_{max}, y_{min}, y_{μ}, y_{std}, and y_{med}) for each of the 26 time series, resulting in a **130-D** problem.

16.3 Data Normalization

Since the scale of each feature is different, a normalization step is needed before training the classifier. Normalizing the features means scaling them into the range $[0, 1]$. Note that this type of scaling is chosen over standardization because the latter is used on data having Gaussian distribution, which is not necessarily the case of the current dataset.

After splitting the dataframe into training and testing sets, the method *MinMaxScaler()* from *sklearn preprocessing* is used.

```
# Split the dataset into training and testing
df_Train, df_Test = train_test_split(df3, test_size=0.2, shuffle=True)
x_train = df_Train.loc[:,features].to_numpy()
y_train = np.ravel(df_Train.loc[:, target].to_numpy())
x_test = df_Test.loc[:,features].to_numpy()
```

```
y_test = np.ravel(df_Test.loc[:, target].to_numpy())

scaler = MinMaxScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
```

16.4 PCA Testing

Since the number of features is significant, one can think about using PCA in order to reduce the dimension of the problem, benefiting from all its advantages. But at the same time, one should not compromise with the accuracy of the model. Therefore, since we cannot know at first sight the optimal number of principal components, the latter hyperparameter is variated, and a base Random Forest classifier model is trained using each time a different number of principal components. The goal is to observe if it is beneficial to reduce the dimension, and if so, to which dimension.

In Figure 20, one can see that the model starts to reach an acceptable accuracy above 30 principal components. And above 40, the accuracy stays stable around 80%.
On the other hand, the training time was represented just to confirm that reducing the dimension of the problem does indeed decrease the training time of the model.

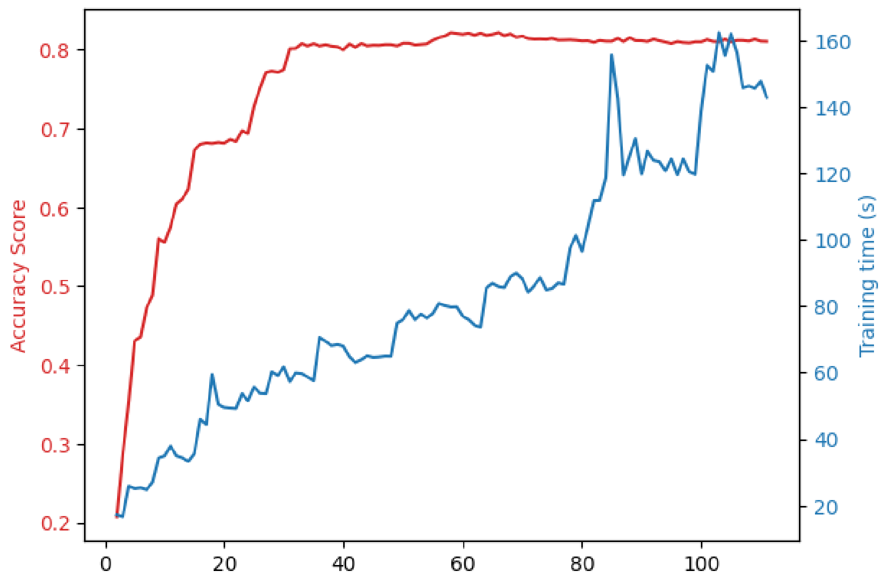


Figure 20: Plot showing the evolution of the model's accuracy and training time vs number of principal components.

Since decreasing the number of components did not save a major amount of time while keeping a maximal accuracy, we chose to stick with the original dimension and to not use PCA here.

17 Classification

17.1 Multi-class Classification

Classification problems can be separated into two families; binary and multi-class classification. Since binary classification can only separate between two classes, this is clearly the case of multi-class (multinomial) classification, more specifically a 10-class problem (10 different drivers in DS2.

There are a lot of algorithms that are conceptualized to be binary classifiers. But a way to turn around it is by using one of the two different strategies that arise when performing multi-class classification, in order to transform an N-class problem into multiple binary classification problems:

- One-vs-Rest (OvR): the idea here is to train k binary classifiers, each one of them as a detector of one of the k classes. Then the predicted class will be the one whose classifier has the highest score.
- One-vs-One (OvO): this strategy consists of training binary classifiers that are able to distinguish between a pair of classes. Thus, one needs $k \times (k-1)/2$ classifiers for k-class problem.

Scikit-learn can detect if the user is trying to use a binary classifier for a multi-class problem, so it implements automatically OvR (except SVM where it runs OvO, because SVM scales poorly with the size of the training sets, hence it is faster to train many classifiers on small training sets (OvO) than training few classifiers on large training sets.

17.2 Classification Algorithms

To classify the 10 drivers based on the selected features, the following machine learning algorithms were considered:

- **Random Forest:**
  It is an Ensemble of Decision Trees trained via the bagging method in general [27]. It combines the outputs of these multiple trees to attain a single result.
  An Ensemble is a combination of individual models that creates a more powerful new model.
  To simplify, a decision tree starts with a root node. The outgoing branches from the root node then feed into the internal decision nodes. Based on the available features, the nodes form homogenous subsets, which are called leaf nodes, on the basis of comparison. The leaf nodes represent all the possible outcomes within the dataset.
  So each tree in this Ensemble is trained on a random subset of features (bagging), to decrease the correlation between trees. While in a classical decision tree, the algorithm considers all the possible features splits.

- **Support Vector Machine:**
  It is a supervised ML algorithm used for linear and non-linear classification, and regression. Its goal is to find a hyperplane in an N-dimensional space that segregates the data points. The dimension of the hyperplane is always $N-1$, with N the number of used features. The algorithm searches for nearest points from the decision boundaries for all the classes (these points are called support vectors). Then the algorithm tries to maximize the margins (distances) between the support vectors and the potential hyperplane, and that's how the latter is found.
  As mentioned before, its multi-class support is handled according to OvO scheme.

- **K Nearest Neighbors:**
  It is a supervised non-parametric[1] and lazy[2] classifier, that uses proximity to make predictions about the grouping of data points. It relies on majority voting to assign a label for a data point, meaning it will decide based on the label that is most frequent around this data point. KNN is very simple, easy to understand and versatile.

- **AdaBoost:**
  Boosting is a technique used for Ensemble modelling, it was first presented by Freund and Schapire

[1]No assumption for underlying data distribution.
[2]There is no need for training of the model as all the data points used at the time of prediction.

in 1977 [28]. It fits a first model, then builds a second one that predicts accurately where the first's performance is poor. Hence, the combination of these two should perform better. So, this procedure continues by adding iteratively a new model that improves the performance of the previous combination. Thus, AdaBoost or Adaptative Boosting is a technique used as an Ensemble method. It builds a model and gives similar weights to all data points, then it assigns higher weights to the wrongly classified points. Then, the next model will be training while taking into consideration those higher weights, thus giving more importance to the previous false classification, hoping to decrease the errors.

- **Gradient Boosting:**
  Gradient Boosting is a type of Machine Learning Boosting, it consists of choosing the best possible next model by minimizing the overall prediction error. It actually fits a new model to the residual errors made by the previous predictor model.

18 Model Training

To train any estimator, Scikit-learn provide a similar method. Also, because training time is an important metric, it was recorded systematically using the time module. Note that if a *random_state* parameter exists for a certain estimator, it will always be set to the same number to enable reproducibility.

```
start = time.time()
estimator.fit(x_train, y_train)
finish = time.time()
training_time = finish - start
```

18.1 Random Forest

The first Random Forest model's parameters were kept to default, except $random\_state$.

```
RandomForest_0 = RandomForestClassifier(n_estimators=100, criterion='gini',
                  max_depth=None, min_samples_split=2, min_samples_leaf=1,
                  min_weight_fraction_leaf=0.0, max_features='sqrt',
                  max_leaf_nodes=None, min_impurity_decrease=0.0,
                  bootstrap=True, oob_score=False, n_jobs=None,
                  random_state=123, verbose=0, warm_start=False,
                  class_weight=None, ccp_alpha=0.0, max_samples=None)
```

Its most important hyperparameters are the following:

- *n_estimators*: the number of decision trees in the forest. $Default = 100$.
- *criterion*: the function to measure the quality of each split. It can be "gini" (for the Gini impurity) or "log_loss" or "entropy" (for the Shannon information gain). $Default = gini$.
- *max_depth*: the maximum depth of each individual tree. The bigger the depth, the more the chance of overfitting. But the latter is not a problem because of the existence of several trees. $Default = None$.
- *min_samples_split*: the minimum samples required to split an internal node. $Default = 2$.
- *max_features*: the number of random features to include when looking for the best node splitting. $Default = "sqrt"$ (square root of the number of features).

- *bootstrap*: whether bootstrap samples are used when building trees or no. If False, the whole dataset is used to build each tree. $Default = False$.
- min_samples_leaf: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least *min_samples_leaf* training samples in each of the left and right branches. $Default = 1$.

18.2 Support Vector Machine

SVM uses a mathematical function defined in its parameters, which is the kernel. The goal of the kernel function is to map the original non-linear data points into a higher dimensional space, where they become separable by the decision boundary. The chosen kernel was 'RBF' (Gaussian Radial Basis Function), which is usually preferred for non-linear data. The formula of RBF depends on γ which is one of the parameters. For now, it is kept in its default value 'scale', meaning it will be equal to $1/(n\_features \times X.var())$. This parameter in addition to others will be tuned in a following phase.
Hence, the first SVM model is initialized in the following way:

```
Svm_0 = svm.SVC(kernel='rbf',
                C=1000,
                class_weight='balanced',
                random_state=123)
```

The value 'balanced' was passed to *class_weight* parameter to let SVC automatically adjust the weights of the different classes, in a way that a more present class doesn't influence more the training of the estimator. The *C* parameter tells the SVM optimization how much the user wants to avoid misclassifying each training example. For large values of *C*, the optimization will choose a smaller margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly.

18.3 K Nearest Neighbors

The main parameter to choose here is the number of neighbors k (*n_neighbors*). Research has shown that there is no optimal solution that fits all data sets. But one has to keep in mind that for a small k, noise will have a higher influence on the result, whereas a large k will make the algorithm computationally expensive. Generally, an odd number of neighbors is chosen for an even number of classes.
An iteration over several values of k was done to observe its effect on the accuracy of the model.

As you can see in Figure 21, it is clear that the highest accuracy was attained with only one neighbor, and it became worst as the number of neighbor increased. Also, the figure shown the significant difference between training and predicting time of the models.

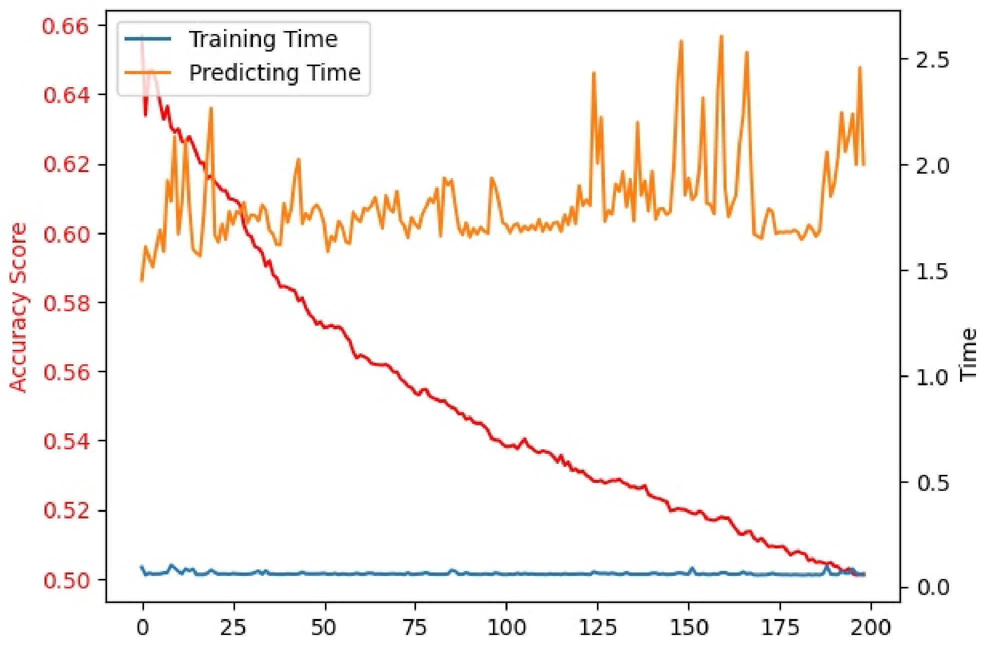


Figure 21: The evolution of the model accuracy, training and predicting time with respect to k.

```
KNN_dict = {}
for i in range(1, 200):
    KNN_dict[str(i)] = {}
    KNeighbors= KNeighborsClassifier(n_neighbors=i)
    start = time.time()
    KNeighbors.fit(x_train , y_train)
    stop = time.time()
    KNN_dict[str(i)]['training_time'] = stop - start
    start = time.time()
    y_pred = KNeighbors.predict(x_test)
    stop = time.time()
    KNN_dict[str(i)]['predicting_time'] = stop - start
    knn_acc = accuracy_score(y_test , y_pred)
    KNN_dict[str(i)]['accuracy'] = knn_acc
```

To conclude, this algorithm didn't show any potential in this application, especially if it is compared with Random Forest algorithm for example. This might be the case because it is sensitive to the quality of the data (If it is noisy for example), and does not work well with high dimensionality, as this will complicate the distance calculations. Also, it requires high memory because it needs to store all the training data.

18.4 AdaBoost

AdaBoostClassifier has *base_estimator* as first parameter, which is basically the base estimator from which the boosted Ensemble is built. It is kept to default, which is *DecisionTreeClassifier* with $max\_depth = 1$.

Other than that, the main hyperparameters are:

- *n_estimator*: the maximum number of estimators at which boosting is terminated. $Default = 50$.
- *learning_rate*: weight applied to each classifier at each boosting iteration, meaning a higher learning rate increases the contribution of each classifier. $Default = 1$.

The number of hyperparameters is relatively small for this classifier, meaning the hyperparameters tuning phase won't be as expensive as for the others.

```
AB_0 = AdaBoostClassifier(base_estimator=None,
                          n_estimators=50,
                          learning_rate=1.0,
                          algorithm='SAMME.R',
                          random_state=123)
```

18.5 Gradient Boosting

The Gradient Boosting classifier's parameters were initially kept to default to get a baseline idea of the performance. These parameters can be divided into three categories:

- Tree specific parameters: affect each individual tree. (*min_samples_split, min_samples_leaf, max_depth, min_weight_fraction_leaf, max_leaf_nodes, max_features*).
- Boosting parameters: affect the boosting operation. (*learning_rate, n_estimators, subsample*).
- Miscellaneous parameters: other parameters for overall functioning. (*loss, init, random_state, verbose, warm_start, presort*).

```
GB_0 = GradientBoostingClassifier(loss='log_loss', learning_rate=0.1,
                                  n_estimators=100, subsample=1,
                                  criterion='friedman_mse', min_samples_split=2,
                                  min_samples_leaf=1, min_weight_fraction_leaf=0,
                                  max_depth=3, min_impurity_decrease=0,
                                  init=None, random_state=123,
                                  max_features=None, verbose=1,
                                  max_leaf_nodes=None,
                                  warm_start=False, validation_fraction=0.1,
                                  n_iter_no_change=None, tol=1e-4,
                                  ccp_alpha=0)
```

19 Model Evaluation

19.1 Confusion Matrix

A confusion matrix is a useful tool for evaluating the performance of a classification model. It provides a clear and concise summary of how well the model is performing by comparing its predicted labels with the true labels of a set of data.

The confusion matrix is presented as a table, where the rows represent the actual (true) labels and the columns represent the predicted labels. The diagonal elements of the matrix represent the number of correct predictions, while the off-diagonal elements represent the misclassifications.

In essence, the confusion matrix allows us to easily identify the number of true positives, false positives, true negatives, and false negatives, giving a comprehensive view of the model's performance. This information is critical for understanding the strengths and weaknesses of a classifier and making informed decisions on how to improve its accuracy.

The four values contained in the matrix, which are used to calculate some metrics, can be defined as follows (in a binary classification context):

- True Positive (TP): when a model correctly predicts a positive class (the class in question).
- False Negative (FN): when a model incorrectly predicts a negative class (the class not in question). In other words, it predicts the positive class for a negative one.
- False Positive (FP): when a model incorrectly predicts a positive class, indicating that the item in question does not actually belong to that class.
- True Negative (TN): when a model correctly predicts a negative class.

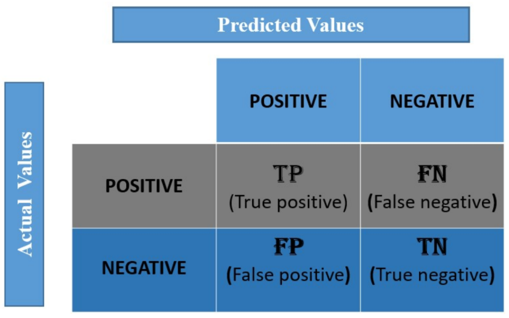


Figure 22: Confusion matrix in a binary classification problem.

19.2 K-Fold Cross Validation

K-fold cross-validation is a technique for evaluating the performance of machine learning models on a limited data sample. It is used for assessing how the statistical analysis generalizes to an independent dataset. This technique evaluates ML models by training them on of the input data, and then testing these model on unused complementary subset of the data.
Since one should never use the testing set during the training phase, cross-validation is applied only on the training DS.

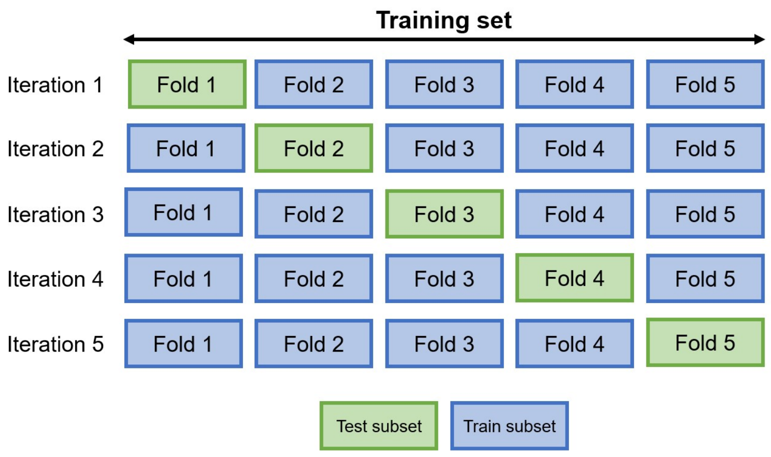


Figure 23: 5-fold cross-validation representation

The value of k should be chosen in a way that the train/validation chunks are large enough to be representative of the full dataset. The rule of thumb was followed here by using 5-fold cross validation, because it is not time expensive, and it is sufficient to validate the models' training.

At the beginning, all the training set is divided into k subsets ($k = 5$ in this case). Then in a first iteration out of k, the model is trained based on $k - 1$ subsets, then validated on the complementary chunk left. This procedure is repeated k times until all the chunks are used once for testing (see Figure 23). By doing that, it is guaranteed that every data point get an equal opportunity to be included in the validation subset.

For each iteration, the performance of the estimator in question is evaluated by calculating performance metrics such as:

- Accuracy: the proportion of correct predictions out of all the prediction made.
- Precision: the proportion of positive predictions that are actually correct. $P = TP/(TP + FP)$.
- Recall: also known as sensitivity, it is the ratio of positive instances that are correctly detected by the classifier. $TP/(TP + FN)$.
- F1_score: It is the harmonic mean of precision and recall. It is a good metric to use when the positive class is rare or when the cost of false positives and false negatives is different:

$$f1\_score = \frac{2}{1/precision + 1/recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

After cross validation phase, one can calculate the average of each metric, thus obtain a robust general results.

20 Results and Analysis

For each of the 5 classifiers mentioned above, 5-folds cross validation was implemented, and 4 scoring metrics were retrieved, as well as the training times. The code can be found in appendix 26.5.
Note that for the precision, recall, and f1 scores, the *average* parameter is set to 'weighted', meaning that we are aggregating the performance metrics across all classes, taking into account the relative importance of each class in the dataset. For example, if the driver 'A' has more samples than the other drivers, it will give more weight to the performance of this class in the overall evaluation.

Classifier	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**	**Training Time (s)**
Random Forest	81.6 ± 0.4	81.9 ± 0.4	81.6 ± 0.4	81.5 ± 0.4	25.8 ± 1.4
SVM	80.5 ± 0.3	80.5 ± 0.2	80.5 ± 0.3	80.5 ± 0.3	112.9 ± 9.7
20 Nearest Neighbors	60 ± 0.1	60.4 ± 0.1	60 ± 0.1	59.9 ± 0.1	0.07 ± 0.037
AdaBoost	32.4 ± 0.9	34.2 ± 1.0	32.4 ± 0.9	29.2 ± 1.1	14.8 ± 0.49
Gradient Boosting	72.5 ± 0.2	73 ± 0.3	72.5 ± 0.2	72.2 ± 0.2	481.6 ± 16.3

Table 1: Table showing the results obtained after the classifiers' cross validation training.

From the scoring table above, it is clear that both **Random Forest** and **Support Vector Machine** outperformed all the others in all metrics. But RF's training time is way faster than SVM's.
Inversely, AdaBoost algorithm with its default parameters performed very poorly. Most probably if we experimented more and focused on tuning its parameters, it will reach better accuracy, but it is highly unlikely that it will reach RF's scores.

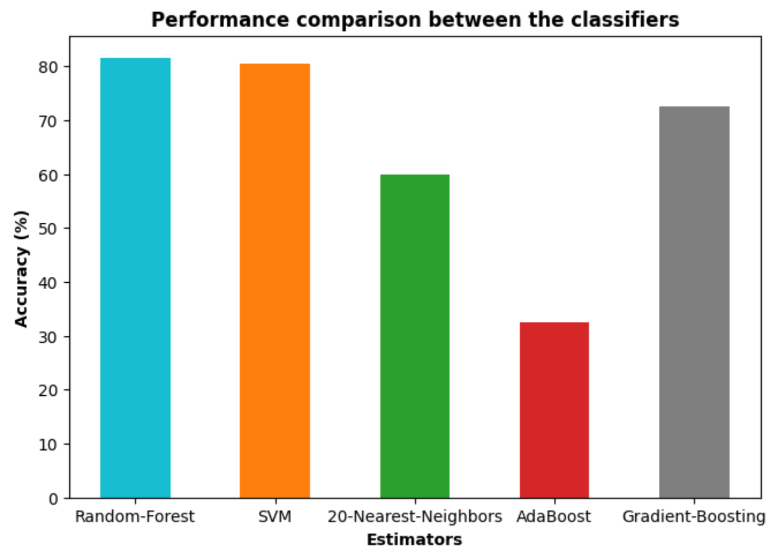


Figure 24: Accuracies of the classifiers (cross validation results)

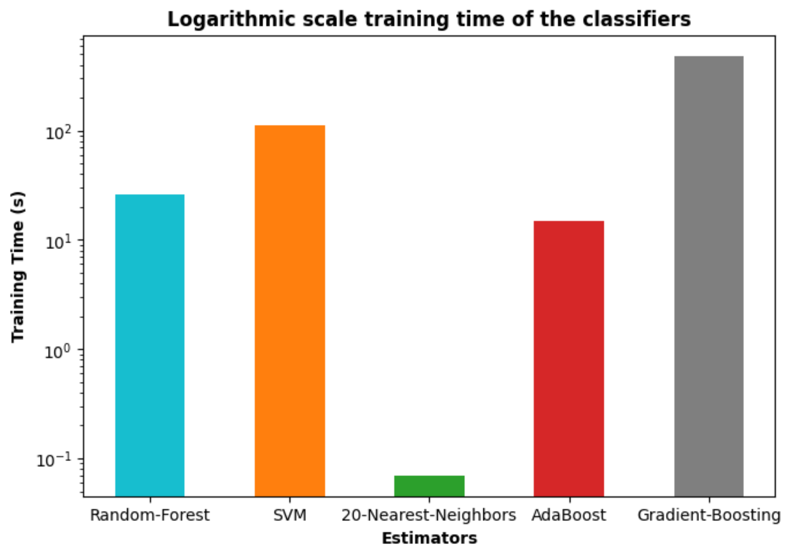


Figure 25: Logarithmic training times of the classifiers (cross validation results)

Notice that for Nearest Neighbor classifier, the training time is negligible (less than 1 second), because this algorithm's time cost is in his predicting phase rather than training (visualized in Figure 21). Also, Gradient Boosting's performance is acceptable ($accuracy = 72.5\%$), but it takes almost 20 times more than Random Forest to train, which is relatively poor.

As a consequence of analyzing the results above, **Random Forest** algorithm outperformed all the others in terms of scores and training time. ($accuracy = 81.6\%$, $training\ time = 25.8\ s$)

21 Hyperparameters Tuning

As RF was the winner between the other tested classifiers, it will be tuned to get even better performance out of it. Hyperparameter tuning is the process of selecting the optimal values of hyperparameters for ML model (RF classifier in our case).

The steps involved in hyperparameters tuning are the following:

1. Choose the appropriate algorithm for the model.
2. Decide the parameter space.
3. Decide the method for searching parameter space.
4. Decide the cross-validation method.
5. Decide the score metrics to evaluate the model.

The hyperparameters that will be tuned are the following:

- *n_estimators*
- *max_features*
- *max_depth*
- *min_samples_split*
- *min_samples_leaf*
- *bootstrap*

The tuning phase will be divided into two parts, one being a more broad search of parameters using *Sklearn.model_selection*'s *RandomizedSearchCV*, and one is a detailed search in a specific range of hyperparameters' values using *GridSearchCV*.

21.1 Randomized Search CV

In technical terms, *RandomizedSearchCV* will take the default Random Forest classifier, a set of hyperparameters, and a distribution of values for each hyperparameter as input. It then performs a randomized search by selecting a random combination of hyperparameters from the specified distributions, fitting the estimator on the training data with these hyperparameters, and evaluating the performance of the estimator on a validation set. This process is repeated for a specified number of iterations or until the search reaches a predefined limit. 50 iterations were made in our case, which allows for an efficient search over a large hyperparameter space without the need for exhaustive search, which can be computationally expensive.

```
# nb of trees in random forest
n_estimators = [int(x) for x in np.linspace(start=10, stop=500, num=15)]
# nb of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum nb of levels in tree
max_depth = [int(x) for x in np.linspace(start=10, stop =150, num=15)] + [None]
# Minimum nb of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum nb of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]

param_grid = {'n_estimators': n_estimators,
              'max_features': max_features,
              'max_depth': max_depth,
              'min_samples_split': min_samples_split,
              'min_samples_leaf': min_samples_leaf,
                'bootstrap': bootstrap}

RF_default = RandomForestClassifier(random_state=randomstate)

RF_RandomGrid = RandomizedSearchCV(estimator = RF_default,
                                   param_distributions = param_grid,
                                   n_iter = 50,
                                   cv = 5,
                                   verbose = 2,
                                   n_jobs = -1)

RF_RandomGrid.fit(X_train, y_train)

# Get the optimum found parameters
print(RF_RandomGrid.best_params_)
```

The Randomized search picked the following hyperparameters as optimal:

- *n_estimators* = **80**.
- *max_features* = **'sqrt'**.
- *max_depth* = **130**.
- *min_samples_split* = **2**.
- *min_samples_leaf* = **1**.
- *bootstrap* = **False**.

21.2 Grid Search CV

GridSearchCV method works similarly, but with only one difference; it performs an exhaustive search by evaluating the performance of the estimator with each possible combination of hyperparameters in the chosen grid, making it a powerful tool as it ensures that the best combination of parameters is selected. However, it can be computationally expensive and impractical for large hyperparameter spaces.
The strategy was to take few values around the ones picked by the previous phase and check if a better combination is possible.

```
# The hyperparameters search grid
param_grid = {'n_estimators': [60, 80 , 90, 150, 200],
              'max_features': ['sqrt'],
              'max_depth': [120, 130, 140],
              'min_samples_split': [2, 3, 4],
              'min_samples_leaf': [1, 2, 3],
              'bootstrap': [False]}
```

The grid above is consisted of 135 different combinations, and with 5-folds cross validation, the total of unique estimator fits is 675, which is computationally very expensive.
Finally, the picked hyperparameters values with *GridSearchCV* are:

- *n_estimators* = **200**.
- *max_features* = **'sqrt'**.
- *max_depth* = **120**.
- *min_samples_split* = **2**.
- *min_samples_leaf* = **1**.

- *bootstrap* = **False**.

Estimator	Accuracy (%)	Training time (s)	Improvement in accuracy (%)
Default RF	81.6	26	-
Best RandomizedSearch RF	84.22	34	3.19
Best GridSearch RF	84.76	73	3.87

Table 2: Random Forest Classifiers' performances after hyperparameters tuning.

After tuning the RF model, we reached an accuracy of **84.76%**, with an improvement of 3.87% which is considered a very satisfying result. One can remark that the training time has increased as well. This is due to the fact that *n_estimators* parameter, which controls the number of decision trees in the forest, has increased. Hence, it can be tuned further to compromise between performance, computational cost and overfitting.

22 Feature Analysis

The goal of feature analysis is to acquire understandings on the effectiveness of the chosen features. We took advantage of the fact that we have an RF classifier and used tree-based feature importance (*mean decrease impurity*). In reality, in an RF, each decision tree is constructed on a random subset of the features, which results in a diverse set of trees. The feature importance is then calculated based on how much each feature reduces the impurity in the decision tree. And *mean decrease impurity* is a measure of feature importance, which calculates the total reduction in impurity across all decision trees in the forest that results from splitting on a particular feature.

Top Features	Mean decrease in impurity
mean_Long_Term_Fuel_Trim_Bank1	0.064013
max_Long_Term_Fuel_Trim_Bank1	0.052928
min_Long_Term_Fuel_Trim_Bank1	0.052457
median_Long_Term_Fuel_Trim_Bank1	0.048648
mean_Maximum_indicated_engine_torque	0.021088
max_Maximum_indicated_engine_torque	0.020361
median_Maximum_indicated_engine_torque	0.019917
min_Maximum_indicated_engine_torque	0.018444
std_Long_Term_Fuel_Trim_Bank1	0.016902
min_Intake_air_pressure	0.014851

Table 3: The most important features.

The most important features were extracted (shown in Table 3 and Figure 26). All statistical features except standard deviation of *Long term fuel bank* and *Engine torque* take the lead in terms of importance.

The Long Term Fuel Trim Bank is a long-term adjustment made by the engine control module (ECM) to the fuel delivery system based on feedback from various sensors. It adjusts the fuel delivery system to maintain the air/fuel ratio at the optimal level by adding or subtracting fuel as needed.
Hence, this parameter depends hugely on driving style (as well as other factors like vehicle condition and fuel quality).

All the 130 features' importance can be found in appendix 26.6.

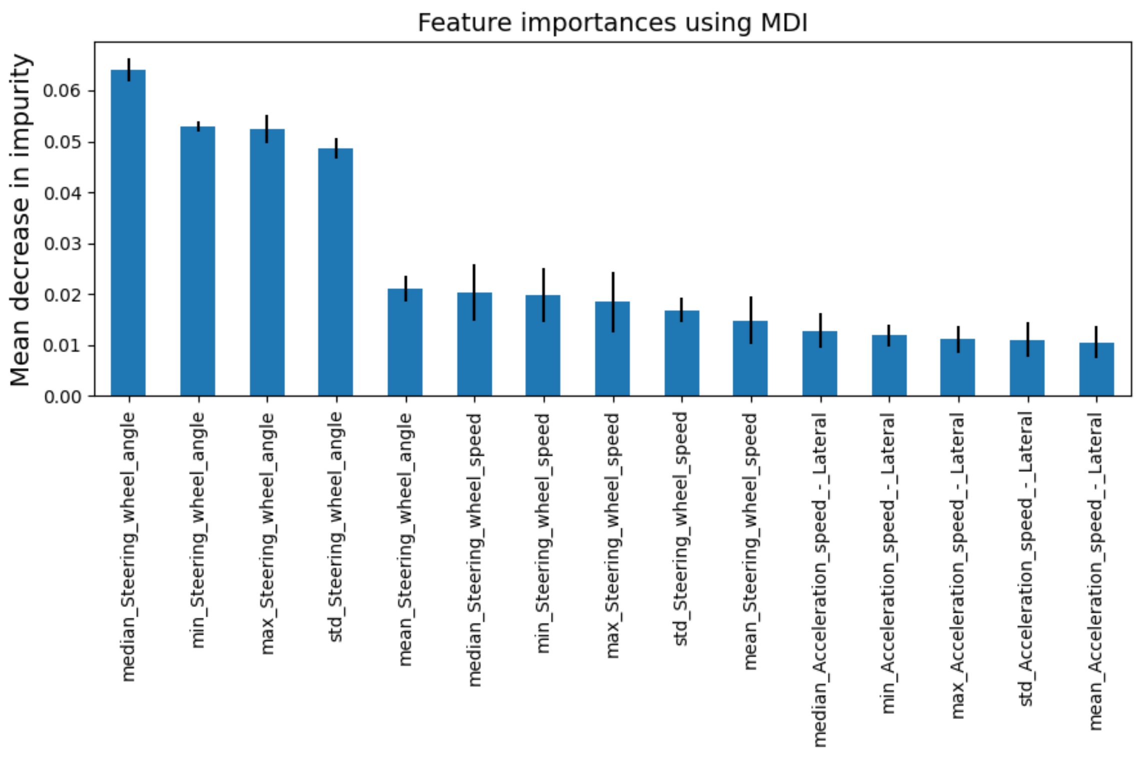


Figure 26: 15 most importance features.

23 Frame Size Analysis

One of the main hyperparameters of this time-series classification problem were sliding window's frame size w and overlap ratio r. Therefore, an experiment was conducted on both of them using the best found classifier (RF) to grasp its effect, especially the model's accuracy and training time. w was incremented from 2 to 20 seconds, and r took 3 values (0%, 25%, and 50%).

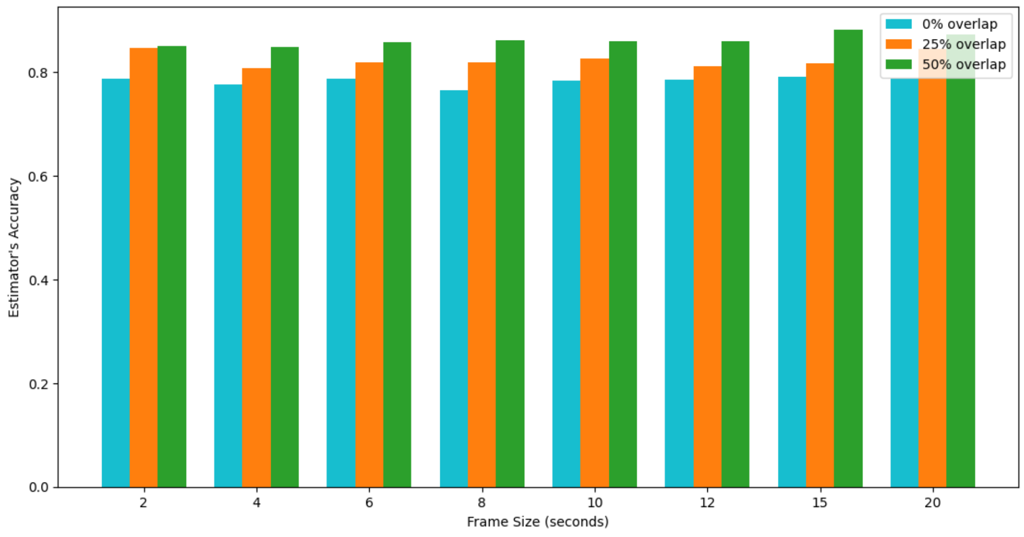


Figure 27: Bar plot showing the outcome of frame size and overlap variation on estimator's accuracy.

Looking at the consecutive frames overlapping percentages, one can conclude that the larger the percentage, the better the accuracy is. This can be linked to the fact that overlapping frames embed important

information related to temporal dependencies between consecutive frames. On the other hand, more overlapping means more data points, hence a slightly larger training time (Figure 28). For a similar reason, the wider the frame, the larger the training time. However, the accuracies don't alter a lot with the increase of frame length.
The best frame size in terms of accuracy was the 15 seconds frame with $r = 50\%$ ($acc \approx 88\%$, $training\ time = 71s$).

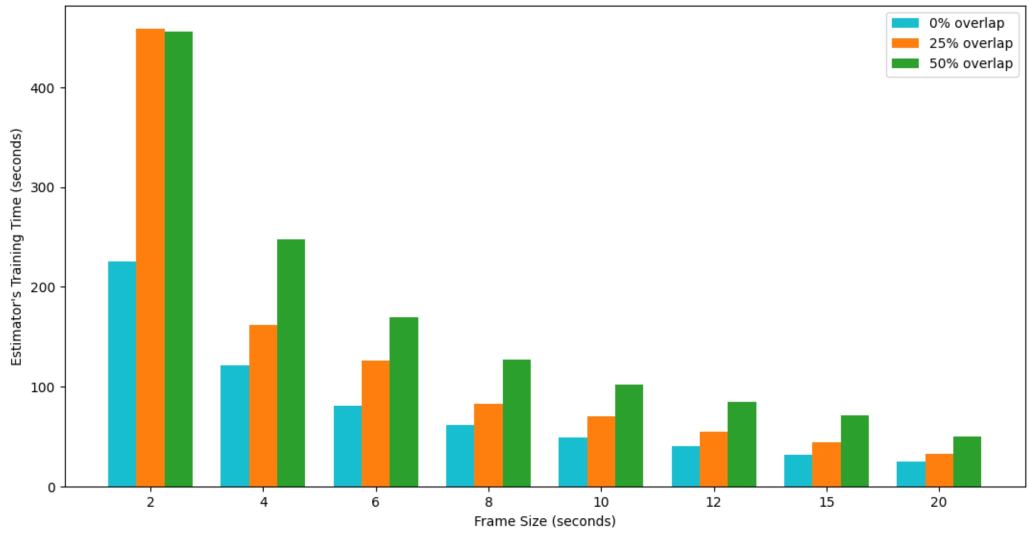


Figure 28: Bar plot showing the outcome of frame size and overlap variation on estimator's training time.

Finally, it was interesting to regroup the 5 statistical features (*Mean, Standard deviation, Minimum, Maximum,* and *Median*) and calculate their importance with the variation of the frame length.
Minimum and *Maximum* were not heavily affected by the increase of frame length. However, the *Standard deviation* and the *Median*'s importance were affected with the length's increase, the first one's value grew while the second's importance decreased (Figure 29).

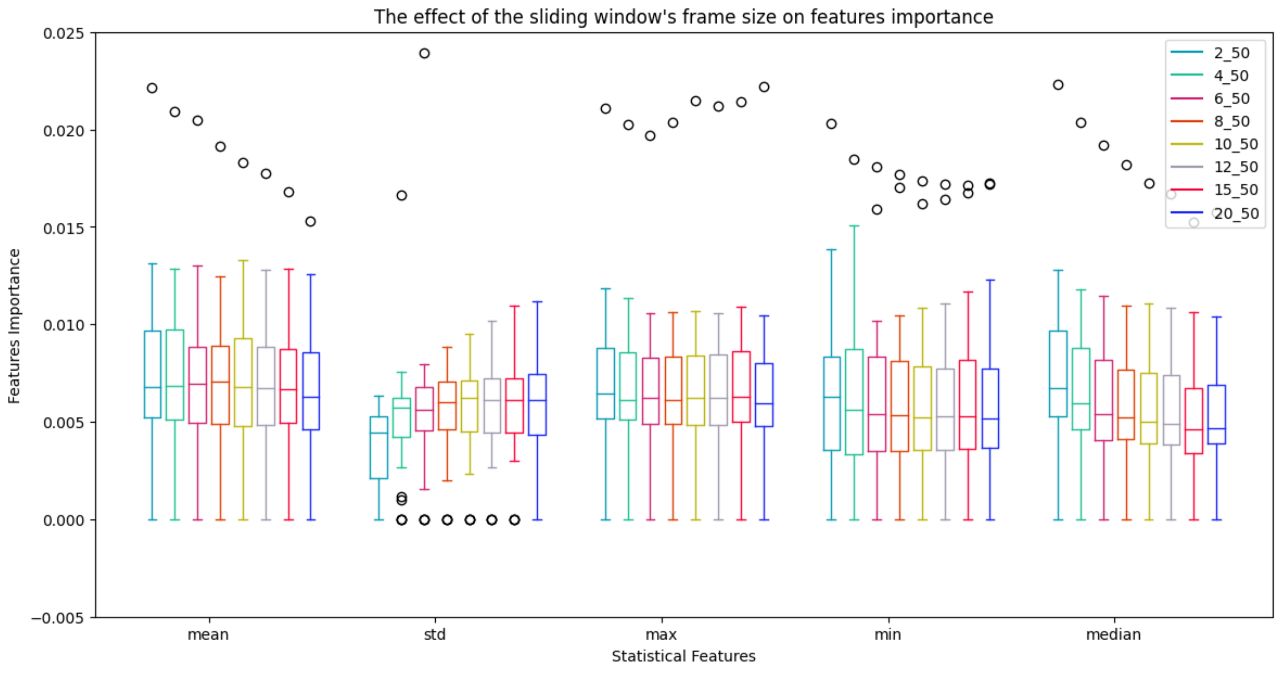


Figure 29: Box plot showing the effect of frame size variations on features.

24 Deep Learning

Artificial Neural Networks (ANNs) lie at the core of Deep Learning algorithms. ANNs are modeled after the structure and function of the human brain, with the nodes or artificial neurons representing biological neurons that communicate with each other.
They are composed of node layers, including an input layer, one or more hidden layers, and an output layer. Each artificial neuron is connected to others and has a weight and threshold associated with it. When an individual node's output surpasses the threshold value, it becomes active and sends data to the next layer in the network. Otherwise, it does not pass any information to the next layer.

The number of hidden layers (also known as the depth of the network), as well as the number of neurons per layer, are one of the hyperparameters of the network model. On the other hand, the number of neurons in the input layer should be equal to the number of features, and the output neurons are equal to the number of different classes in the dataset.

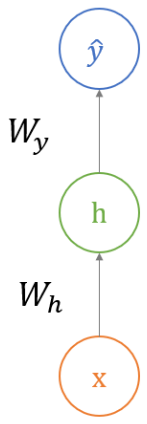


Figure 30: Fully connected neural network.

Let's take a look at an example of a fully connected network. In Figure 30, the input x is affected by an affine transformation W_h $(\sum_{j=1}^{n} w_i.x_j)$ followed by a non-linear transformation (known as activation function), resulting in h. Similarly, the hidden output h is subject to 2 consecutive linear and non-linear transformations, producing the final output $\hat{y}$.
This network can be represented mathematically, with f and g non-linear activation functions:

$$h = f(W_h.x + b_h)$$

$$\hat{y} = g(W_y.h + b_y)$$

The activation function is a critical component of the neural network because it introduces non-linearity, allowing the network to learn and model complex relationships between inputs and outputs. Without non-linearity, a NN would be limited to modeling linear relationships.
The activation of the output layer would usually depend on the use case, whereas the hidden layers can have activations like ReLU, sigmoid, hyperbolic tangent, soft(arg)max, etc.

To build and train the NNs, we used the **Pytorch** framework, which is an open source ML framework originally developed by Meta AI. It is based on multidimensional arrays called tensors, they similar to *Numpy*'s.

24.1 Training Phase

In this context, learning means finding which parameters (weights and biases) minimize a certain cost function.
A working of a neural net is based on 2 principles:

1. **Forward Pass:** when the input data is fed in the forward direction through the network.
2. **BackPropagation:** it is the core algorithm behind how neural networks learn.

During forward propagation, the outputs are calculated based on the current weights and biases. Then a loss function is used to calculate the error between the predicted and the actual output.

Then, the error is propagated back through the network in the Backpropagation, and the algorithm calculates the gradient of the loss function with respect to the weights and biases of each neuron. This gradient represents the direction and magnitude of the change that needs to be made to each weight and bias to minimize the error.
The weights and biases are then updated using the gradient descent algorithm, which makes adjustments in the direction of the negative gradient of the loss function.

To train any NN using Pytorch, there are always these 5 fundamentals step:

1. ***output = model(input)*:** does the forward pass through the network that generates the output.
2. ***J = loss(output, target)*:** calculates the training loss (error between predicted and actual targets).
3. ***model.zero_grad()*:** resets the gradient calculations, because we don't want them to be accumulated from before for the next pass.
4. ***J.backward()*:** does the backpropagation pass, computes $\nabla_x J$ for each variable x, and accumulates it into the gradient of each x ($x.grad \leftarrow x.grad + \nabla_x J$).
5. ***optimizer.step()*:** updates the model's parameters (a step in gradient descent).

24.2 The Neural Network Model

After several trials, a deep neural network was constructed with the following characteristics:

- 4 layers of neurons, of which 2 are hidden.
- $n = 130$ neurons in the input layer (equal to the number of features).
- $m = 10$ neurons in the output layer (equal to the number of classes).
- $h_1 = 400$ neurons in the first hidden layer.
- $h_2 = 200$ neurons in the second hidden layer.
- ReLU or Rectified Linear Unit activation function for both hidden layers (appendix 26.8).
- Softmax activation function for the output layer, which is commonly used for multi-class classification problems (appendix 26.8).

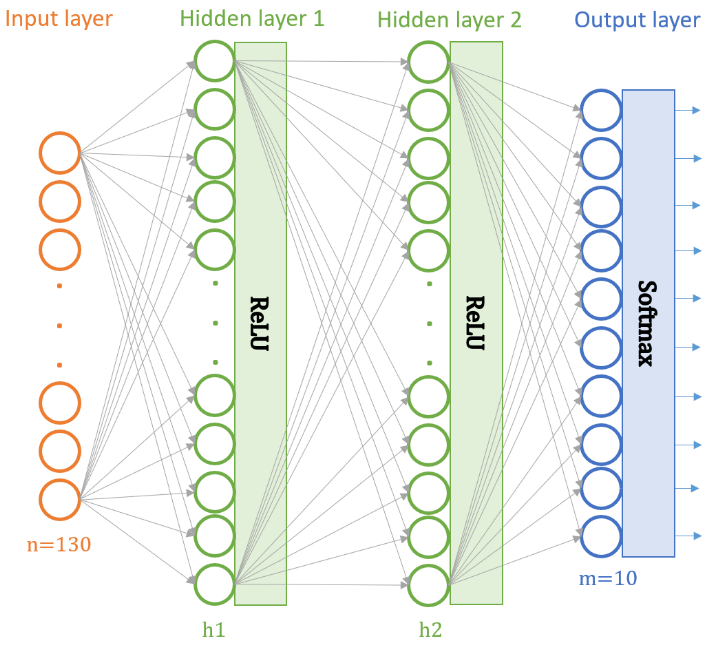


Figure 31: The architecture of the used deep neural network.

For the criterion, **CrossEntropyLoss** was utilized to calculate the loss. In addition, for the choice of optimizer, **Adam** which is an adaptive optimizer. It was preferred over gradient descend optimizers because the latter are proven not to be good with sparse data, and they have a high possibility to be stuck in local minima.

Note that before feeding the training DS into the neural net, it was scaled to take values in $[0, 1]$. Also, we used the method *DataLoader* from Pytorch to split the data into smaller batches. That allows the model to process multiple inputs and their corresponding labels simultaneously during training. By processing a small batch of data at a time, the model can update its parameters more frequently, which can lead to faster convergence. Additionally, it grants for more efficient use of memory during training.

On the other hand, the training DS was further split into training and validation set. The validation set's role was critical during the training phase. In fact, the training error keeps going down during training, whereas validation error goes down to a certain minimum then increases again. So the training will be stopped when validation loss starts increasing. This regularization technique is used to prevent overfitting and generalization loss.

The full training function can be found in the appendix 26.9.

24.3 Results and Analysis

At this stage, the frame length of the Sliding Window algorithm was fixed to **15 seconds**, with an overlap of 50%.

One of the hyperparameters is the batch size percentage with the respect to the training DS. So, a list of 8 percentages were tried to conclude which one is the better choice.

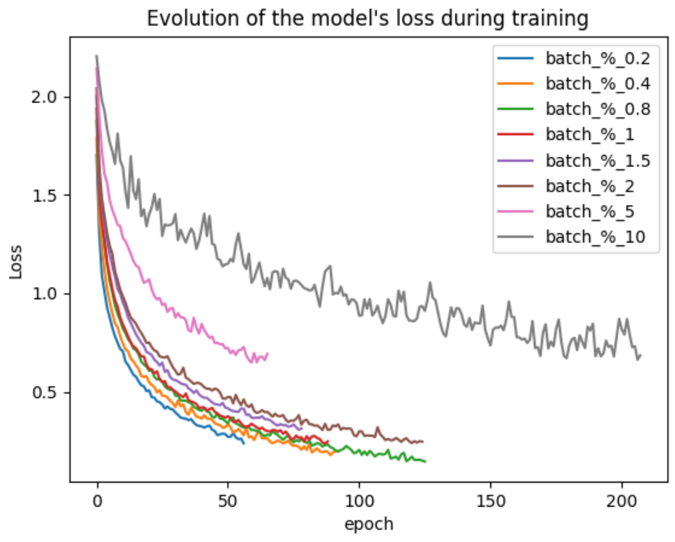


Figure 32: The evolution of training set's loss for different batch sizes.

Figure 32 represents how the loss converges during the training loop. Notice that the early stopping time due to validation' error increase differs from one batch size percentage to another.

Apparently, taking a batch size percentage of 0.8% of the training dataset achieves the highest accuracy (94.7%). It took the network 154 seconds and 126 epochs to converge $(1.2s/epoch)$.

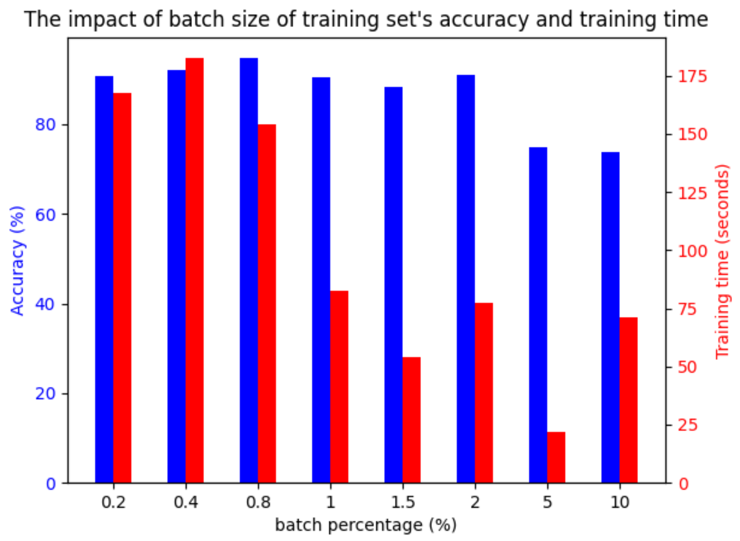


Figure 33: Bar plot showing the accuracies and training time for several batch sizes.

We can notice in Figure 33 that when using a large batch size, the accuracy is relatively low (around 74% for 5 and 10% sizes). Besides, for the lowest batch sizes, the accuracies are good, but the training times are higher than the rest.

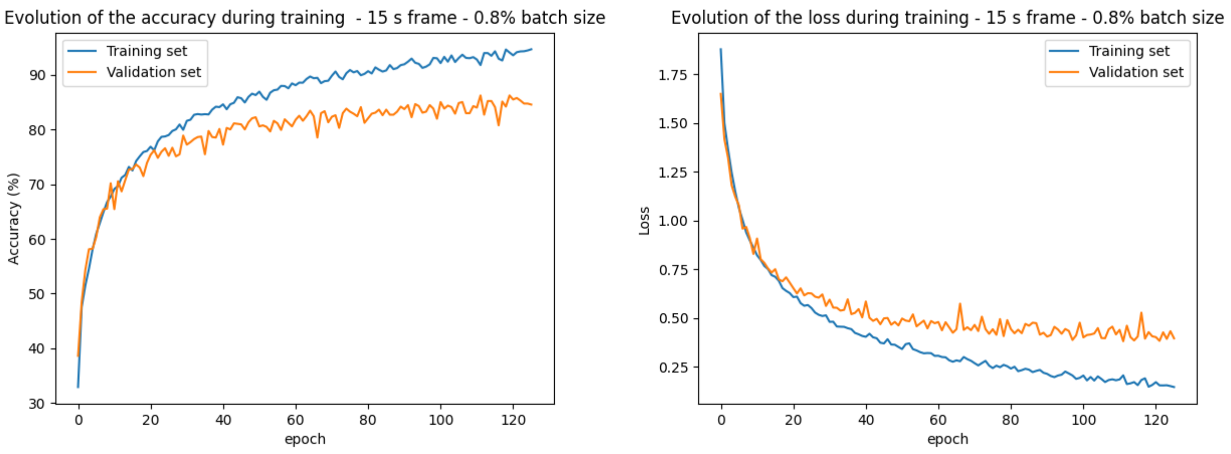


Figure 34: Accuracy and loss evolution of the model with 0.8% batch size

Table 4 shows how well our model generalized, as it achieved an accuracy of 83.15% on an unseen test dataset, which is considered a satisfying result.

	Training set	**Validation set**	**Test set**
Loss	0.25	0.43	0.45
Accuracy (%)	90.54	84.03	83.15

Table 4: Results of the 4 layers DNN model, with window length of 15s.

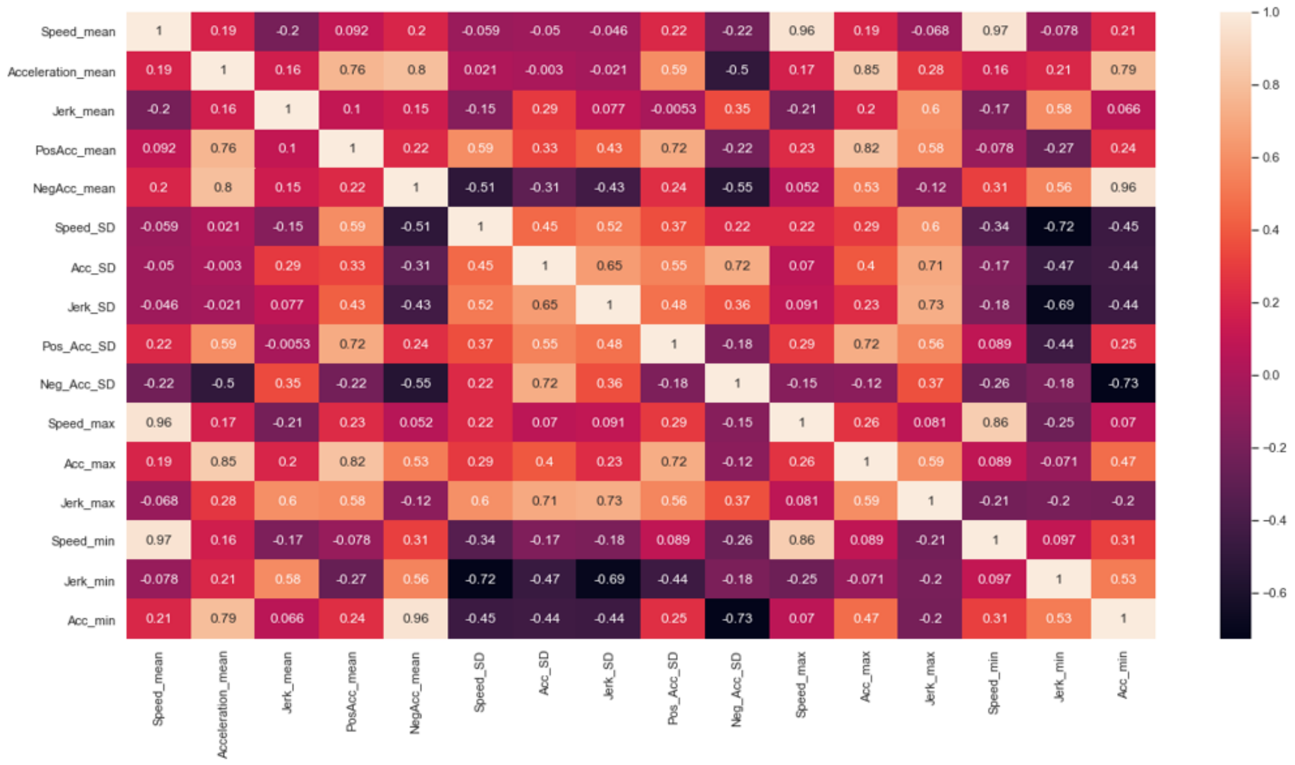
25 Conclusion and perspectives

This project proposed several Machine Learning models ready to recognize the driver behind the wheel based on data extracted from the vehicle's CAN-bus. Mainly, two models came on top, the Random Forest algorithm, a classical ML Ensemble algorithm, and a Deep Neural Network with two hidden layers. The first one achieved an accuracy of **88%** on an unseen testing set, whereas the second attained **83.15%**.
These models can be applied to interpret driving data, extract patterns and classify it, to enable relevant applications on board the commercial vehicles. Besides, it was emphasized on the importance of processing the raw data, and extracting some features before feeding it into any ML algorithm.

For future work, it is worth diving deeper into Deep Learning and tackle Recurrent Neural Networks, as researchers confirmed their impressive results with time-series problems. Also, the investigation of the use of other time-series segmentation methods or not, combined with RNN can present some interesting results. After that, it is necessary to try and test the models in a real car processors and enter an optimization and improvement phase as a way to enable real time applications, like enabling the ADAS to correct and 'teach' the driver more eco-friendly way of driving to decrease energy consumption.

26 Appendices

26.1 Correlation Matrix - Dataset 1



	Speed_mean	Acceleration_mean	Jerk_mean	PosAcc_mean	NegAcc_mean	Speed_SD	Acc_SD	Jerk_SD	Pos_Acc_SD	Neg_Acc_SD	Speed_max	Acc_max	Jerk_max	Speed_min	Jerk_min	Acc_min
Speed_mean	1	0.19	-0.2	0.092	0.2	-0.059	-0.05	-0.046	0.22	-0.22	0.96	0.19	-0.068	0.97	-0.078	0.21
Acceleration_mean	0.19	1	0.16	0.76	0.8	0.021	-0.003	-0.021	0.59	-0.5	0.17	0.85	0.28	0.16	0.21	0.79
Jerk_mean	-0.2	0.16	1	0.1	0.15	-0.15	0.29	0.077	-0.0053	0.35	-0.21	0.2	0.6	-0.17	0.58	0.066
PosAcc_mean	0.092	0.76	0.1	1	0.22	0.59	0.33	0.43	0.72	-0.22	0.23	0.82	0.58	-0.078	-0.27	0.24
NegAcc_mean	0.2	0.8	0.15	0.22	1	-0.51	-0.31	-0.43	0.24	-0.55	0.052	0.53	-0.12	0.31	0.56	0.96
Speed_SD	-0.059	0.021	-0.15	0.59	-0.51	1	0.45	0.52	0.37	0.22	0.22	0.29	0.6	-0.34	-0.72	-0.45
Acc_SD	-0.05	-0.003	0.29	0.33	-0.31	0.45	1	0.65	0.55	0.72	0.07	0.4	0.71	-0.17	-0.47	-0.44
Jerk_SD	-0.046	-0.021	0.077	0.43	-0.43	0.52	0.65	1	0.48	0.36	0.091	0.23	0.73	-0.18	-0.69	-0.44
Pos_Acc_SD	0.22	0.59	-0.0053	0.72	0.24	0.37	0.55	0.48	1	-0.18	0.29	0.72	0.56	0.089	-0.44	0.25
Neg_Acc_SD	-0.22	-0.5	0.35	-0.22	-0.55	0.22	0.72	0.36	-0.18	1	-0.15	-0.12	0.37	-0.26	-0.18	-0.73
Speed_max	0.96	0.17	-0.21	0.23	0.052	0.22	0.07	0.091	0.29	-0.15	1	0.26	0.081	0.86	-0.25	0.07
Acc_max	0.19	0.85	0.2	0.82	0.53	0.29	0.4	0.23	0.72	-0.12	0.26	1	0.59	0.089	-0.071	0.47
Jerk_max	-0.068	0.28	0.6	0.58	-0.12	0.6	0.71	0.73	0.56	0.37	0.081	0.59	1	-0.21	-0.2	-0.2
Speed_min	0.97	0.16	-0.17	-0.078	0.31	-0.34	-0.17	-0.18	0.089	-0.26	0.86	0.089	-0.21	1	0.097	0.31
Jerk_min	-0.078	0.21	0.58	-0.27	0.56	-0.72	-0.47	-0.69	-0.44	-0.18	-0.25	-0.071	-0.2	0.097	1	0.53
Acc_min	0.21	0.79	0.066	0.24	0.96	-0.45	-0.44	-0.44	0.25	-0.73	0.07	0.47	-0.2	0.31	0.53	1

Figure 35: Correlation matrix of all the calculated features.

26.2 Steps of Principal Components Analysis

In theory, the Principal Component Analysis can be broken down into five steps:

1. **Standardization:**
   Feature scaling through standardization is a critical step prior to the execution of the PCA, since the latter is very sensitive to the variances of the input variables. It is a requirement for the optimal performance of many ML algorithms, and skipping this step will most likely lead to the domination of some variables over the others, leading to biased results.
   The standardization incorporates scaling of the features so that to have the same properties as a standard normal distribution (mean equal to zero and standard deviation equal to one).
   Mathematically, it can be done through the following formula:

   $$x_{new} = \frac{x - mean}{standard\ deviation}$$

   In python, the features columns are stored in an array, then it is standardized using the 'StandardScaler' function from 'sklearn.preprocessing'. And a new data frame is reformed using this new scaled data.

```
X = df_final.loc[:, features].values
X = StandardScaler().fit_transform(X)
```

2. **Generating the covariance or correlation matrix for the whole dataset:**
   This matrix is the numerical representation of how much information is contained between two-dimensional space (related to two features) at a time.

3. **Calculating the eigenvectors:**
   After performing the eigen decomposition of the covariance matrix, the eigen vectors are obtained, which represents the direction of variance. In other words, the vector with the highest eigenvalue will give the direction of maximum variance and thus forming the first principal component (variable), and so on.

4. **Transforming Data:**
   This step is accomplished by performing the dot product of the data with the chosen eigenvectors to obtain the new data projections.

All of the above can be taken care of by importing 'PCA' from 'sklearn.decomposition' and implementing the following code:

```
pca = PCA(n_components = 3)
x = df_standardized.loc[:, features].values
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data=principalComponents,
    columns=['principal component 1','principal component 2','principal
        component 3'],
    Index=pd.MultiIndex.from_tuples(df_standardized.index.tolist(),
    names = ['T','evchange', 'Event']))
```

26.3 Ocslab Driving Dataset

#	Variable name	Description
1	Fuel_consumption	The instant value of fuel consumption
2	Accelerator_Pedal_value	Accelerator pedal opening angle percentage
3	Throttle_position_signal	Relative throttle position
4	Short_Term_Fuel_Trim_Bank1	Percentage of change in fuel injection over a short period of time
5	Intake_air_pressure	Used to calculate air density and the engine's air mass flow rate
6	Filtered_Accelerator_Pedal_value	ECU's filtered accelerator pedal opening angle percentage as determined by the accelerator position sensor
7	Absolute_throttle_position	Actual throttle position
8	Engine_soacking_time	Duration where a vehicle's engine is at rest before being started
9	Inhibition_of_engine_fuel_cut_off	The fuel cut-off control system is responsive to a brake switch signal and an engine speed signal having a value above a fuel recovery threshold to decrease the value of a fuel cut-off threshold to again perform the fuel cut-off even in the normal fuel recovery range. This value represents the inhibition of engine fuel cut off
10	Engine_in_fuel_cut_off	The inhibition of engine fuel cut off, i.e. fuel cut-off threshold
11	Fuel_Pressure	The pressure in the fuel system, which is the actual applied pressure by the injector
12	Long_Term_Fuel_Trim_Bank1	Percentage of change in fuel injection over a long period of time, averages the short trims
13	Engine_speed	The number of revolutions the crankshaft makes per minute, also known as RPM
14	Engine_torque_after_correction	The value after correcting the torque to which an engine is adjusted before a gear disengagement
15	Torque_of_friction	The torque caused by the frictional force that occurs when two objects in contact move
16	Flywheel_torque_interventions	The flywheel stores energy when torque is applied by the energy source, and it releases stored energy when the energy source is not applying torque to it. The value represent the flywheel torque after torque interventions
17	Current_spark_timing	The time to set the angle relative to piston position and crankshaft angular velocity that a spark will occur in the combustion chamber near the end of the compression stroke
18	Engine_coolant_temperature	The temperature of the engine coolant of the internal combustion engine
19	Engine_Idle_Target_Speed	The desired idle RPM in relation to the coolant temperature
20	Engine_torque	Represents the load an engine can handle to generate a certain amount of power to rotate the engine on its axis
21	Calculated_LOAD_value	Indicates a percentage of peak available torque
22	Min_indicated_engine_torque	Minimum Engine_torque value
23	Miax_indicated_engine_torque	Maximum Engine_torque value
24	Flywheel_torque	The flywheel's torque
25	Torque_scaling_factor(standardization)	Described as how flexible or how much force can be expressed in a given gear when the driver scales the gear
26	Standard_Torque_Ratio	Described as how flexible or how much force can be expressed in a given gear

27	Requested_spark_retard_angle_from_TCU	The transmission control unit (TCU) controls modern electronic automatic transmissions. This value computes the requested spark retard angle from TCU
28	TCU_requests_engine_torque_limit_(ETL)	Monitors the request to engine torque limits (ETL) by TCU
29	TCU_requested_engine_RPM_increase	Monitors the TCU requests related to the increasing of engine's RPM
30	Target_engine_speed_used_in_lock-up_module	Monitors the lock-up valve, used to shut off the signal pressure line of pneumatic actuators
31	Glow_plug_control_request	Monitors the request to check the glow plug
32	Activation_of_Air_compressor	Shows if the air compressor is in activation state
33	Torque_converter_speed	A particular kind of fluid coupling that is used to transfer rotating power from a prime mover
34	Current_Gear	The current selected gear
35	Engine_coolant_temperature.1	The temperature of the fluid inside the transmission
36	Wheel_velocity_front_left-hand	The speed of the front left-hand wheel
37	Wheel_velocity_rear_right-hand	The speed of the rear right-hand wheel
38	Wheel_velocity_front_right-hand	The speed of the front right-hand wheel
39	Wheel_velocity_rear_left-hand	The speed of the rear left-hand wheel
40	Torque_converter_turbine_speed_-_Unfiltered	A torque converter is a type of fluid coupling that is used to transfer rotating power from a prime mover, such as an internal turbine in this case
41	Clutch_operation_acknowledge	Signals when a clutch operation happens
42	Converter_clutch	Activates the torque converter clutch to prevent slipping at highway speeds
43	Gear_Selection	The current selected gear
44	Vehicle_speed	The current speed of the vehicle in m/s
45	Acceleration_speed_-_Longitudinal	The longitudinal acceleration of the vehicle in m/s^2
46	Indication_of_brake_switch_ON/OFF	Signals whether the brake indicator is on or off
47	Master_cylinder_pressure	The pressure of the master cylinder, a control device that converts non-hydraulic pressure into hydraulic one
48	Calculated_road_gradient	The slope of the currently traveled road
49	Acceleration_speed_-_Lateral	The lateral acceleration of the vehicle in m/s^2
50	Steering_wheel_speed	The current steering wheel speed
51	Steering_wheel_angle	The current steering wheel angle
52	Time(s)	The relative time when the sample was taken (in seconds)
53	Class	The driver identifier
54	PathOrder	To differentiate between the outward leg (1) and the return leg (2) of the round trip

Table 5: All variables present in the dataset [29].

26.4 Selected Features

Engine Related	Engine_speed
	Engine_torque_after_correction
	Engine_coolant_temperature
	Engine_torque
	Minimum_indicated_engine_torque
	Maximum_indicated_engine_torque
Transmission Related	Accelerator_Pedal_value
	Throttle_position_signal
	Filtered_Accelerator_Pedal_value
	Absolute_throttle_position
	Torque_scaling_factor(standardization)
	Standard_Torque_Ratio
	Current_Gear
	Wheel_velocity_front_left-hand
	Wheel_velocity_rear_right-hand
	Wheel_velocity_front_right-hand
	Wheel_velocity_rear_left-hand
	Vehicle_speed
	Acceleration_speed_-_Longitudinal
	Acceleration_speed_-_Lateral
	Steering_wheel_speed
	Steering_wheel_angle
Fuel Related	Intake_air_pressure
	Fuel_consumption
	Short_Term_Fuel_Trim_Bank1
	Long_Term_Fuel_Trim_Bank1

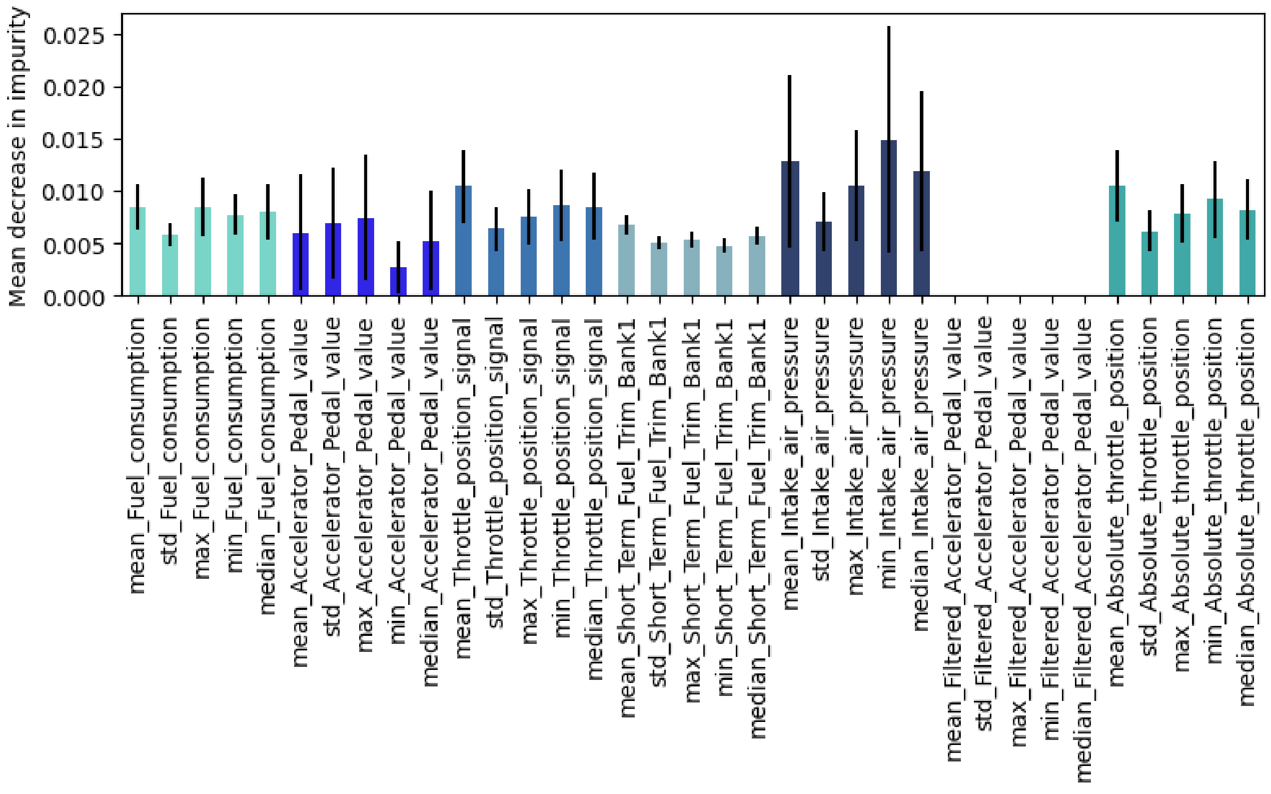
Table 6: Table showing all selected features.

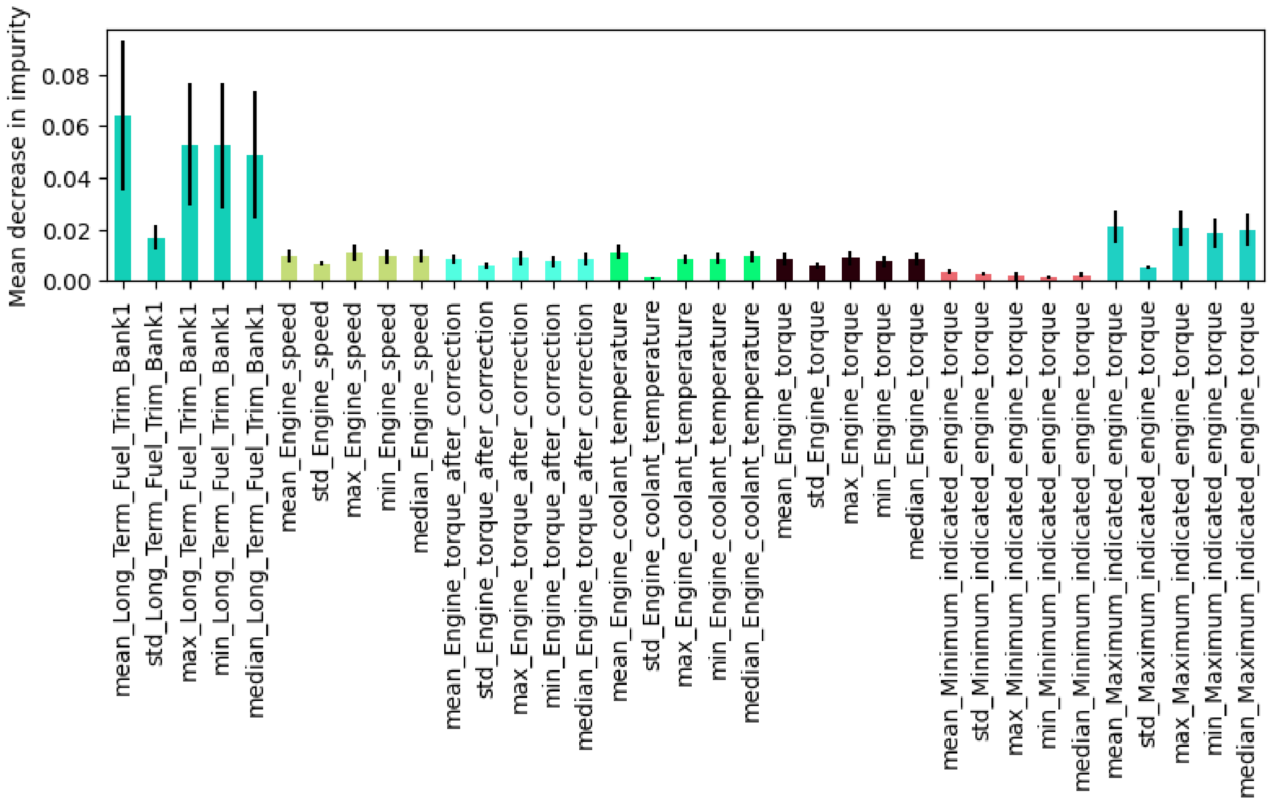
26.5 K-folds cross validation on multiple classifiers

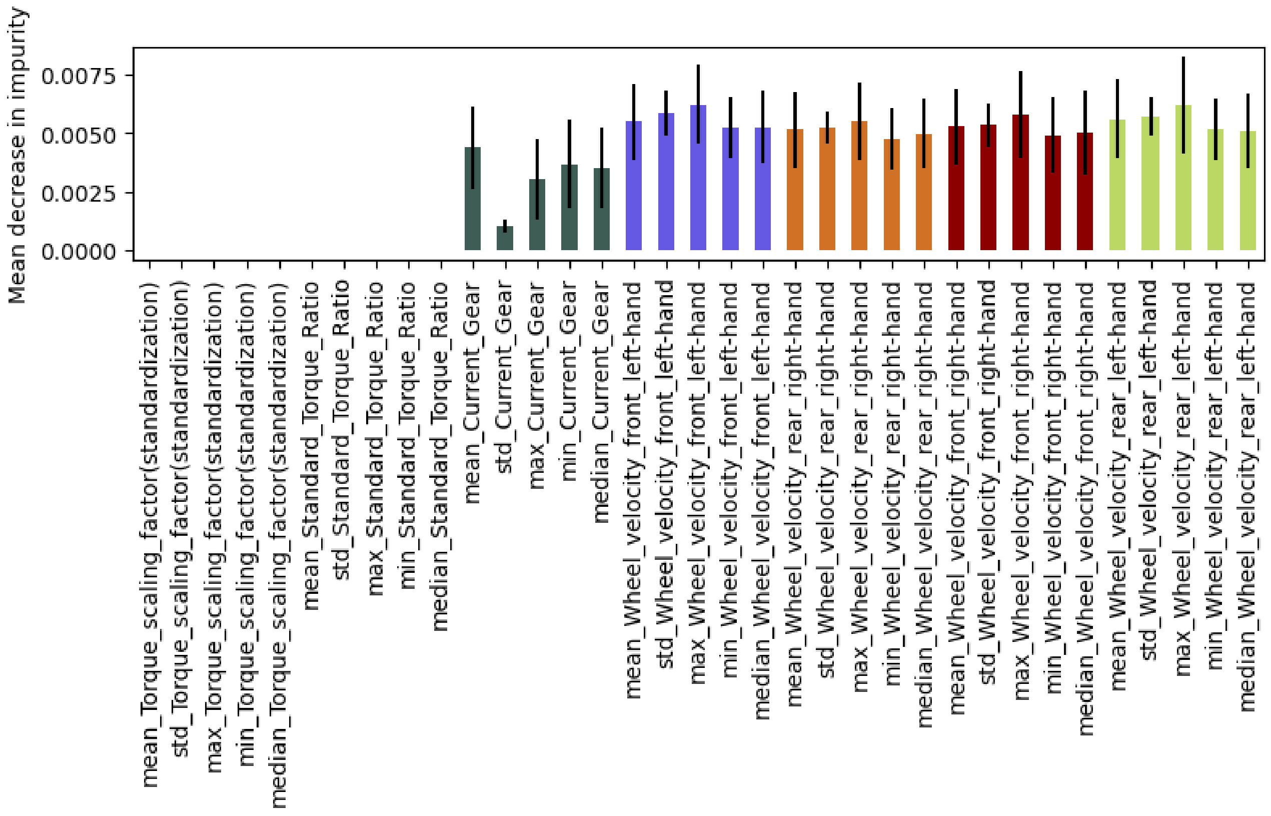
```
classifiers = [RandomForest_1,
               Svm_0,
               KNeighbors_0,
               AB_0,
               GB_0]
classifier_names = ["Random-Forest",
                    "SVM",
                    "20-Nearest-Neighbors",
                    "AdaBoost",
                    "Gradient-Boosting"]
# nb of folds for cross validation
k = 5
# The evaluation metrics
evaluation_metrics = ["accuracy_score",
                    "precision_score",
                    "recall_score",
                    "f1_score"]

# --------------------------------------------------------------------
# The dataset
target = ['Class']
features = df3.drop(['Time(s)', 'PathOrder']+target, axis=1).columns.tolist()
df_Train, df_Test = train_test_split(df3, test_size=0.2, shuffle=True)

# Cross validation on training set ONLY
X = df_Train.loc[:,features].reset_index(drop=True)
y = df_Train.loc[:,target].reset_index(drop=True)

scaler = MinMaxScaler()
kf = KFold(n_splits=k, shuffle=True, random_state=randomstate)


scores = {}
for i, clf in enumerate(classifiers):
    scores[classifier_names[i]] = {}
    for train_index, test_index in kf.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = np.ravel(y.iloc[train_index]),
           np.ravel(y.iloc[test_index])

        scaler.fit(X_train)
        X_train = scaler.transform(X_train)
        X_test = scaler.transform(X_test)

        start = time.time()
        clf.fit(X_train, y_train)
        stop = time.time()
        y_pred = clf.predict(X_test)

        for j, metric in enumerate(evaluation_metrics):
            score = None
            if metric == 'accuracy_score':
                score = accuracy_score(y_test, y_pred)
            elif metric == 'precision_score':
                score = precision_score(y_test, y_pred, average='weighted')
            elif metric == 'recall_score':
                score = recall_score(y_test, y_pred, average='weighted')
            elif metric == 'f1_score':
                score = f1_score(y_test, y_pred,average='weighted')

            if metric in scores[classifier_names[i]].keys():
                scores[classifier_names[i]][metric].append(score)
            else:
                scores[classifier_names[i]][metric] = [score]

        if 'training_time' in scores[classifier_names[i]].keys():
            scores[classifier_names[i]]['training_time'].append(stop - start)
        else:
            scores[classifier_names[i]]['training_time'] = [stop - start]
```

26.6 Features Importance

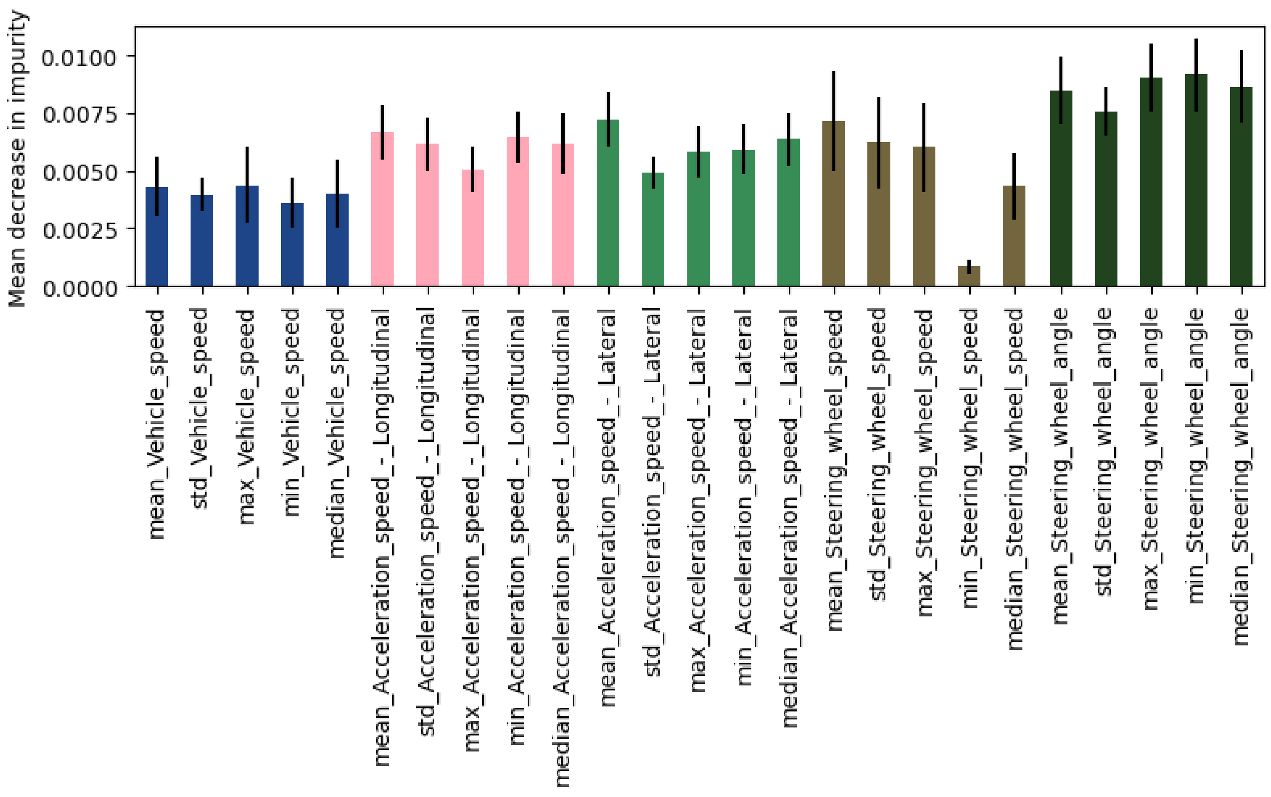
Here is the list of all 130 features used to classify driver from DS2.







Mean decrease in impurity
0.0000
0.0025
0.0050
0.0075
mean_Torque_scaling_factor(standardization)
std_Torque_scaling_factor(standardization)
max_Torque_scaling_factor(standardization)
min_Torque_scaling_factor(standardization)
median_Torque_scaling_factor(standardization)
mean_Standard_Torque_Ratio
std_Standard_Torque_Ratio
max_Standard_Torque_Ratio
min_Standard_Torque_Ratio
median_Standard_Torque_Ratio
mean_Current_Gear
std_Current_Gear
max_Current_Gear
min_Current_Gear
median_Current_Gear
mean_Wheel_velocity_front_left-hand
std_Wheel_velocity_front_left-hand
max_Wheel_velocity_front_left-hand
min_Wheel_velocity_front_left-hand
median_Wheel_velocity_front_left-hand
mean_Wheel_velocity_rear_right-hand
std_Wheel_velocity_rear_right-hand
max_Wheel_velocity_rear_right-hand
min_Wheel_velocity_rear_right-hand
median_Wheel_velocity_rear_right-hand
mean_Wheel_velocity_front_right-hand
std_Wheel_velocity_front_right-hand
max_Wheel_velocity_front_right-hand
min_Wheel_velocity_front_right-hand
median_Wheel_velocity_front_right-hand
mean_Wheel_velocity_rear_left-hand
std_Wheel_velocity_rear_left-hand
max_Wheel_velocity_rear_left-hand
min_Wheel_velocity_rear_left-hand
median_Wheel_velocity_rear_left-hand



Mean decrease in impurity
0.0000
0.0025
0.0050
0.0075
0.0100
mean_Vehicle_speed
std_Vehicle_speed
max_Vehicle_speed
min_Vehicle_speed
median_Vehicle_speed
mean_Acceleration_speed_-_Longitudinal
std_Acceleration_speed_-_Longitudinal
max_Acceleration_speed_-_Longitudinal
min_Acceleration_speed_-_Longitudinal
median_Acceleration_speed_-_Longitudinal
mean_Acceleration_speed_-_Lateral
std_Acceleration_speed_-_Lateral
max_Acceleration_speed_-_Lateral
min_Acceleration_speed_-_Lateral
median_Acceleration_speed_-_Lateral
mean_Steering_wheel_speed
std_Steering_wheel_speed
max_Steering_wheel_speed
min_Steering_wheel_speed
median_Steering_wheel_speed
mean_Steering_wheel_angle
std_Steering_wheel_angle
max_Steering_wheel_angle
min_Steering_wheel_angle
median_Steering_wheel_angle

26.7 DNN Model Class

```
class Model1(nn.Module):
    def __init__(self, n_inputs, n_outputs):
        super(Model1, self).__init__()

        # input to first layer
        self.layer1 = nn.Linear(n_inputs, 400)
        # init weights
        nn.init.kaiming_uniform_(self.layer1.weight, nonlinearity='relu')
        # second layer
        self.layer2 = nn.Linear(400, 200)
        nn.init.kaiming_uniform_(self.layer2.weight, nonlinearity='relu')
        # third layer and output
        self.layer3 = nn.Linear(200, n_outputs)
        nn.init.xavier_uniform_(self.layer3.weight)
        # activation
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.layer3(x)
        # x = self.softmax(x)
        # not needed because we are using CrossEntropyLoss (softmax is embedded
           inside it)
        return x
```

26.8 Activation Functions

26.8.1 Rectified Linear Unit

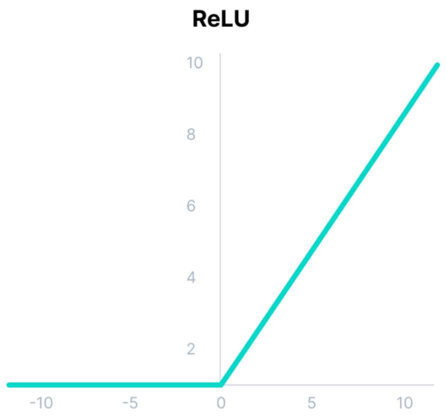


Figure 36: ReLU function.

$$ReLU(x) = max(0, x)$$

26.8.2 Softmax

Softmax is a mathematical function that converts a tensor of numbers into a tensor of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the tensor.

With z being the tensor of raw outputs from the NN, the i^{th} entry in the softmax output vector softmax(z) can be thought of as the predicted probability of the test input belonging to class i.

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

26.9 Pytorch neural network's training loop code

```python
def fit(model, train_loader, optimizer, criterion):
    model.train()
    train_running_loss = 0.0
    train_running_correct = 0
    counter = 0
    total = 0

    for i, (inputs, labels) in enumerate(train_loader):
        counter += 1
        data, target = inputs.to(device), labels.to(device)
        total += target.size(0)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        train_running_loss += loss.item()
        _, preds = torch.max(outputs.data, 1)
        train_running_correct += (preds == target).sum().item()
        loss.backward()
        optimizer.step()

    train_loss = train_running_loss / counter
    train_accuracy = 100. * train_running_correct / total

    return train_loss, train_accuracy


def validate(model, test_dataloader, criterion):
    model.eval()
    val_running_loss = 0.0
    val_running_correct = 0
    counter = 0
    total = 0

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(test_dataloader):
            counter += 1
            data, target = inputs.to(device), labels.to(device)
            total += target.size(0)
            outputs = model(data)
            loss = criterion(outputs, target)

            val_running_loss += loss.item()
            _, preds = torch.max(outputs.data, 1)
            val_running_correct += (preds == target).sum().item()

        val_loss = val_running_loss / counter
        val_accuracy = 100. * val_running_correct / total
        return val_loss, val_accuracy


def training_loop(model, dataset, optimizer, criterion, nb_epochs, batch_size,
    stopping_patience, validset_percentage=0.1):
    train_loss, train_accuracy = [], []
    val_loss, val_accuracy = [], []
    early_stopping = EarlyStopping(patience=stopping_patience)
    train_d, valid_d = random_split(dataset,
                                    [len(dataset) -
                                      int(validset_percentage*len(dataset)),
                                    int(validset_percentage*len(dataset))])
    train_dataloader = DataLoader(train_d, batch_size, shuffle=True)
    val_dataloader = DataLoader(valid_d, batch_size=batch_size)

    start = time.time()
    for epoch in range(nb_epochs):
        train_epoch_loss, train_epoch_accuracy = fit(model,
                                                     train_dataloader,
                                                     optimizer,
                                                     criterion
                                                     )

        val_epoch_loss, val_epoch_accuracy = validate(model,
                                                     val_dataloader,
                                                     criterion
                                                     )
        train_loss.append(train_epoch_loss)
        train_accuracy.append(train_epoch_accuracy)
        val_loss.append(val_epoch_loss)
        val_accuracy.append(val_epoch_accuracy)

        print(f"Epoch [{epoch+1}/{nb_epochs}] -> Train Loss:
            {train_epoch_loss:.4f}, Train Acc: {train_epoch_accuracy:.2f} // Val
            Loss: {val_epoch_loss:.4f}, Val Acc: {val_epoch_accuracy:.2f}")
        # if the nb of epoch since last improve in val_loss is bigger that
            patience -> break
        early_stopping(val_epoch_loss)
        if early_stopping.best_sofar:
            print('INFO: saving...')
            torch.save(model.state_dict(), 'Saved/NNmodel_acc_'+str(batch_size))

        if early_stopping.early_stop:
            break


    end = time.time()
    train_time = end - start
    print(f"Training time: {train_time:.3f} seconds")

    return train_loss, train_accuracy, val_loss, val_accuracy, train_time
```

27 References

[1] W. H. Organization *et al.*, “Road safety,” 2020.

[2] S. Yang, W. Wang, F. Zhang, Y. Hu, and J. Xi, “Driving-style-oriented adaptive equivalent consumption minimization strategies for hevs,” *IEEE Transactions on Vehicular Technology*, vol. 67, no. 10, pp. 9249–9261, 2018.

[3] V. C. Magaña and M. Muñoz-Organero, “Artemisa: A personal driving assistant for fuel saving,” *IEEE Transactions on Mobile Computing*, vol. 15, no. 10, pp. 2437–2451, 2015.

[4] M. Martinez, J. Echanobe, and I. del Campo, “Driver identification and impostor detection based on driving behavior signals,” in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2016, pp. 372–378.

[5] Y. L. Murphey, R. Milton, and L. Kiliaris, “Driver’s style classification using jerk analysis,” *2009 IEEE Workshop on Computational Intelligence in Vehicles and Vehicular Systems*, pp. 23–28, 2009.

[6] R. Stoichkov, “Android smartphone application for driving style recognition,” *Dept. Elect. Eng. Inf. Technol., Technische Univ. Munchen*, 2013.

[7] F. U. Syed, D. Filev, and H. Ying, “Fuzzy rule-based driver advisory system for fuel economy improvement in a hybrid electric vehicle,” in *NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society*, 2007, pp. 178–183. DOI: 10.1109/NAFIPS.2007.383833.

[8] G. Li, S. E. Li, B. Cheng, and P. Green, “Estimation of driving style in naturalistic highway traffic using maneuver transition probabilities,” *Transportation Research Part C: Emerging Technologies*, vol. 74, pp. 113–125, 2017.

[9] A. Das, M. N. Khan, and M. M. Ahmed, “Detecting lane change maneuvers using shrp2 naturalistic driving data: A comparative study machine learning techniques,” *Accident Analysis & Prevention*, vol. 142, p. 105 578, 2020.

[10] A. Chowdhury, T. Chakravarty, A. Ghose, T. Banerjee, and P. Balamuralidhar, “Investigations on driver unique identification from smartphone’s gps data alone,” *Journal of Advanced Transportation*, vol. 2018, pp. 1–11, 2018.

[11] W. Wang, J. Xi, A. Chong, and L. Li, “Driving style classification using a semisupervised support vector machine,” *IEEE Transactions on Human-Machine Systems*, vol. 47, no. 5, pp. 650–660, 2017.

[12] L. Moreira-Matias and H. Farah, “On developing a driver identification methodology using in-vehicle data recorders,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 9, pp. 2387–2396, 2017.

[13] M. M. Bejani and M. Ghatee, “Convolutional neural network with adaptive regularization to classify driving styles on smartphones,” *IEEE transactions on intelligent transportation systems*, vol. 21, no. 2, pp. 543–552, 2019.

[14] D. Jeong, M. Kim, K. Kim, *et al.*, “Real-time driver identification using vehicular big data and deep learning,” in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2018, pp. 123–130.

[15] I. Gace, D. Pevec, H. Vdovic, J. Babic, and V. Podobnik, “Driving style categorisation based on unsupervised learning: A step towards sustainable transportation,” in *2021 6th International Conference on Smart and Sustainable Technologies (SpliTech)*, 2021, pp. 1–6. DOI: 10.23919/SpliTech52315.2021.9566371.

[16] Z. Constantinescu, C. Marinoiu, and M. Vladoiu, “Driving style analysis using data mining techniques,” *International Journal of Computers Communications & Control*, vol. 5, no. 5, pp. 654–663, 2010.

[17] Y. Feng, S. Pickering, E. Chappell, P. Iravani, and C. Brace, “Driving style analysis by classifying real-world data with support vector clustering,” *Powertrain and Vehicle Research Center, University of Bath*, 2018.

[18] M. Van Ly, S. Martin, and M. M. Trivedi, “Driver classification and driving style recognition using inertial sensors,” in *2013 IEEE Intelligent Vehicles Symposium (IV)*, 2013, pp. 1040–1045. DOI: 10.1109/IVS.2013.6629603.

[19] E. Keogh, S. Chu, D. Hart, and M. Pazzani, “Segmenting time series: A survey and novel approach,” in *Data mining in time series databases*, World Scientific, 2004, pp. 1–21.

[20] I. A. Gheyas and L. S. Smith, “Feature subset selection in large dimensionality domains,” *Pattern recognition*, vol. 43, no. 1, pp. 5–13, 2010.

[21] D. Patel, R. Modi, and K. Sarvakar, “A comparative study of clustering data mining: Techniques and research challenges,” *International Journal of Latest Technology in Engineering, Management & Applied Science*, vol. 3, no. 9, pp. 67–70, 2014.

[22] S. Na, L. Xumin, and G. Yong, “Research on k-means clustering algorithm: An improved k-means clustering algorithm,” in *2010 Third International Symposium on intelligent information technology and security informatics*, Ieee, 2010, pp. 63–67.

[23] S. Aranganayagi and K. Thangavel, “Clustering categorical data using silhouette coefficient as a relocating measure,” in *International conference on computational intelligence and multimedia applications (ICCIMA 2007)*, IEEE, vol. 2, 2007, pp. 13–17.

[24] “Ocslab driving dataset.” (), [Online]. Available: http://ocslab.hksecurity.net/Datasets/driving-dataset. (accessed: 20.01.2023).

[25] B. I. Kwak, J. Woo, and H. K. Kim, “Know your master: Driver profiling-based anti-theft method,” 2016.

[26] T. G. Dietterich, “Machine learning for sequential data: A review,” in *Structural, Syntactic, and Statistical Pattern Recognition*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 15–30.

[27] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O’Reilly Media, 2017, ISBN: 978-1491962299.

[28] “Adaboost algorithm – a complete guide for beginners.” (), [Online]. Available: https://www.analyticsvidhya.com/blog/2021/09/adaboost-algorithm-a-complete-guide-for-beginners/.

[29] F. Martinelli, F. Mercaldo, A. Orlando, V. Nardone, A. Santone, and A. K. Sangaiah, “Human behavior characterization for driving style recognition in vehicle system,” *Computers & Electrical Engineering*, vol. 83, p. 102 504, 2020, ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2017.12.050. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790617329531.

[30] S. Jafarnejad, “Machine learning-based methods for driver identification and behavior assessment: Applications for can and floating car data,” *University of Luxembourg, Esch-sur-Alzette, Luxembourg*, 2020.

[31] J. Zhang, Z. Wu, F. Li, *et al.*, “A deep learning framework for driving behavior identification on in-vehicle can-bus sensor data,” *Sensors*, vol. 19, no. 6, 2019. DOI: 10.3390/s19061356.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] M. N. Azadani and A. Boukerche, “Driving behavior analysis guidelines for intelligent transportation systems,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 6027–6045, 2021.

[34] S. Birrell, J. Taylor, A. McGordon, J. Son, and P. Jennings, “Analysis of three independent real-world driving studies: A data driven and expert analysis approach to determining parameters affecting fuel economy,” *Transportation research part D: transport and environment*, vol. 33, pp. 74–86, 2014.

[35] W. Dib, A. Chasse, P. Moulin, A. Sciarretta, and G. Corde, “Optimal energy management for an electric vehicle in eco-driving applications,” *Control Engineering Practice*, vol. 29, pp. 299–307, 2014.

[36] O. Taubman-Ben-Ari, M. Mikulincer, and O. Gillath, “The multidimensional driving style inventory-scale construct and validation,” *Accident Analysis & Prevention*, vol. 36, no. 3, pp. 323–332, 2004.