BIRMINGHAM CITY UNIVERSITY

SCHOOL OF COMPUTING AND DIGITAL TECHNOLOGY
MSc Advanced Computer Science

Road Traffic Prediction

Coursework Assignment
CMP7161 Advanced Data Science

Module Co-ordinator:
Yevgeniya Kovalchuk

Submitted by:
Elizabeta Budini 19147099

May 11, 2020

Abstract

This study explores different machine learning algorithms to predict the level of traffic on UK roads based on different aspects. The Department for Transportation releases every year a report, with datasets, about road traffic in the UK. The approach is based on the distribution of various features in the dataset. There are linear and non-linear relationships to be analysed and three ML regressions (Random Forest, SVR, Linear) will be applied and compared.

Contents

1 Introduction **1**
1.1 Aim 1
1.2 Background 1
1.3 Dataset description 1
1.4 Problem statement 3

2 Pre-processing and visualization **4**
2.1 Pre-processing 4
2.2 Visualization 8

3 Machine learning models **13**
3.1 Multiple Linear regression model 13
3.1.1 Correlation 13
3.1.2 Test and training split 14
3.1.3 Linear regression modelling 15
3.1.4 Linear regression modelling with RANSAC 16
3.2 Random forest regression 17
3.2.1 Random forest train-test modelling 18
3.2.2 Random forest cross-validation modelling 19
3.2.3 Random forest with synthetic features modelling 19
3.2.4 Model tuning 20
3.3 Support Vector Regression 22
3.3.1 SVR Modelling 23

4 Results **24**

5 Conclusion **26**

Chapter 1

Introduction

1.1 Aim

Predict road traffic level (in billion vehicle miles) based on different features of traffic in the UK.

1.2 Background

In 2018 more than 328.1 billion of vehicles miles were driven in UK roads. From 1993 to 2018 it has increased by 28% due to many factors. The traffic level is strictly correlated with the economy (Gross Domestic Product), population growth, density and habits. In the last few decades many studies have been focused on traffic and flow prediction for several reasons. Traffic has a variety of impacts on the environment (Department for Transport, 2019), one of this is the pollution level: road transport accounted for 25% of the UK's CO2 emissions in 2017. Also, Intelligent Traffic Systems (ITS) are being developed and studied to improve congestion control and safety on roads. Predicting traffic level with high accuracy will improve the solutions for these systems (Do et al., 2019).

1.3 Dataset description

The UK Department for Transport collects traffic data to analyse the level of road traffic in the country. This dataset describes road traffic over UK from 1993 to 2018. Traffic refers to the total distance travelled by all vehicles over a specific period, is it measured in vehicle miles which combines the number of vehicles on the road and the travelled distance. The dataset has 12 features. Follows the description of these feature according to Department for Transport (2019):

- `year`: Counts are shown for each year from 1993 to 2018.
- `region_id`: Government office region id in which the link sits in

ID	Region
1	South West
2	East Midlands
3	Scotland
4	Wales
5	North West
6	London
7	East of England
8	Yorkshire and The Humber
9	South East
10	West Midlands
11	North East

Table 1.1: Government office region IDs

- `name`: Government office region in which the link sits in.
- `road_category_id`: the classification of the road type.

Category	Definition
1	M or Class A Trunk Motorways
2	M or Class A Principal Motorways
3	Class A Trunk roads
4	Class A Principal Roads
5	Class B roads
6	Class C and U roads

Table 1.2: Road categories

- `total_link_length_km`: Total length of the network road link (in kilometres).
- `total_link_length_mi`: Total length of the network road link (in miles).
- `pedal_cycles`: Traffic volume (in thousands of vehicle kilometres) for pedal cycles.
- `two_wheeled_motor_vehicles`: Traffic volume (in thousands of vehicle miles) for two-wheeled motor vehicles.
- `cars_and_taxis`: Traffic volume (in thousands of vehicle miles) for Cars and Taxis.
- `buses_and_coaches`: Traffic volume (in thousands of vehicle miles) for Buses and Coaches
- `lgvs (vans)`: Traffic volume (in thousands of vehicle miles) for Light Goods Vans.
- `all_hgvs (lorries)`: Traffic volume (in thousands of vehicle miles) for all Heavy Goods Vehicles.

- `all_motor_vehicles`: Traffic volume (in thousands of vehicle miles) for all motor vehicles

1.4 Problem statement

The problem this study is going to approach falls into the supervised learning category. In supervised learning problems, the model will predict unseen or future data based on labeled training features (Raschka and Mirjalili, 2017). The target variable, in this case `all_vehicles` will be referred to as y^i, while the input features will be referred as x^i. A model based on these input features (such as `year`, `road_category`, etc) will predict traffic volume (in billions of vehicle miles). X an Y will denote, correspondingly, the space of input and output variables. In this dataset both X and Y belong to the set of positive real numbers $\mathbb{R}^+$. As the target variable is continuous the problem will be a regression problem. The mathematical definition of the problem that this study will approach is to learn a function $h : X \mapsto Y$ so that h has a good score when predicting the corresponding value of y (Ng, 2012). Addressing this problem could give an efficient control over traffic which includes many benefits such as foresee journey time and avoid traffic congestion (Clark, 2003).

Chapter 2

Pre-processing and visualization

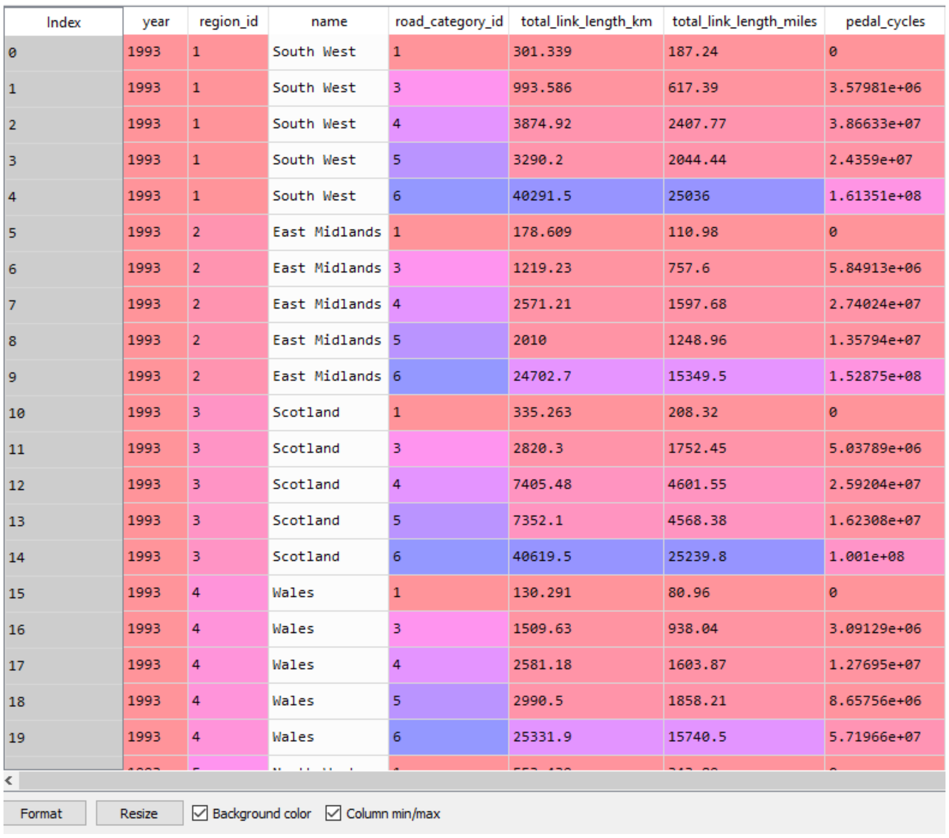
2.1 Pre-processing

This section will describe the steps of cleaning and pre-processing data that are common to all the ML models applied to the dataset. Pre-processing is an important step in which data is transformed and encoded so that the machine can easily understand it. For example, machine larning algorithms will not be able to parse null values of categorical data. Also, some algorithms perform better when data is scaled (Pandey, 2019).

Import dataset

First, to analyse the data and build a model, the dataset needs to be imported. Pandas library is used as follows to achieve this:

```
// Main.py
    import pandas as pd
    #import dataset
    frame = pd.read_csv(file)
```



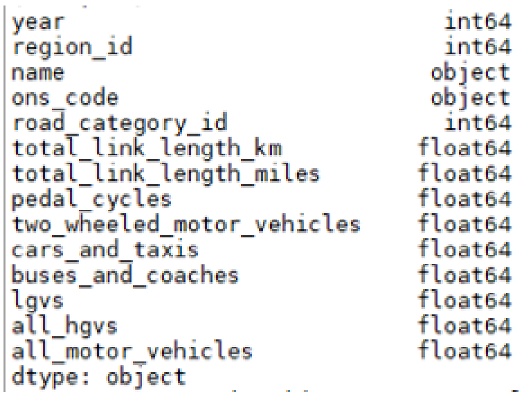
Index	year	region_id	name	road_category_id	total_link_length_km	total_link_length_miles	pedal_cycles
0	1993	1	South West	1	301.339	187.24	0
1	1993	1	South West	3	993.586	617.39	3.57981e+06
2	1993	1	South West	4	3874.92	2407.77	3.86633e+07
3	1993	1	South West	5	3290.2	2044.44	2.4359e+07
4	1993	1	South West	6	40291.5	25036	1.61351e+08
5	1993	2	East Midlands	1	178.609	110.98	0
6	1993	2	East Midlands	3	1219.23	757.6	5.84913e+06
7	1993	2	East Midlands	4	2571.21	1597.68	2.74024e+07
8	1993	2	East Midlands	5	2010	1248.96	1.35794e+07
9	1993	2	East Midlands	6	24702.7	15349.5	1.52875e+08
10	1993	3	Scotland	1	335.263	208.32	0
11	1993	3	Scotland	3	2820.3	1752.45	5.03789e+06
12	1993	3	Scotland	4	7405.48	4601.55	2.59204e+07
13	1993	3	Scotland	5	7352.1	4568.38	1.62308e+07
14	1993	3	Scotland	6	40619.5	25239.8	1.001e+08
15	1993	4	Wales	1	130.291	80.96	0
16	1993	4	Wales	3	1509.63	938.04	3.09129e+06
17	1993	4	Wales	4	2581.18	1603.87	1.27695e+07
18	1993	4	Wales	5	2990.5	1858.21	8.65756e+06
19	1993	4	Wales	6	25331.9	15740.5	5.71966e+07

Figure 2.1: Import dataset into Pandas frame

Data types

Once imported it is important to understand the feature types and data distribution to be able to clean the data and apply standardisation and label encoding algorithms in case is necessary.

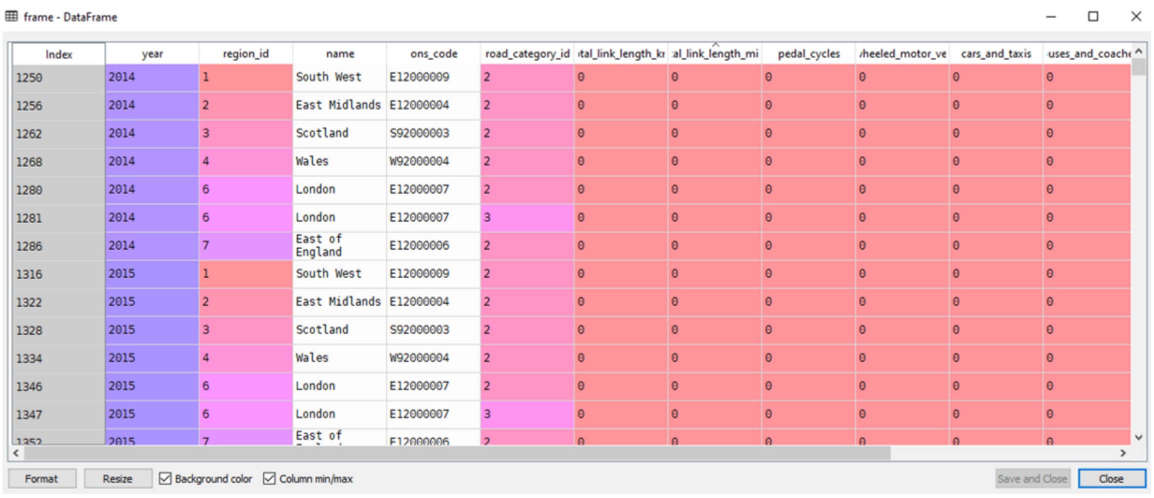
```
// Preprocessing.py
    def data_preprocessing(frame):
        print(frame.shape)
        print(frame.dtypes)
        missing_value=frame.isna()
        print(missing_value)

        #exclude road links with 0 km length
        frame=frame[frame[linkKM] != 0]
        describe=frame.describe()
        [...]
```



```
year                              int64
region_id                         int64
name                             object
ons_code                         object
road_category_id                  int64
total_link_length_km            float64
total_link_length_miles         float64
pedal_cycles                    float64
two_wheeled_motor_vehicles      float64
cars_and_taxis                  float64
buses_and_coaches               float64
lgvs                            float64
all_hgvs                        float64
all_motor_vehicles              float64
dtype: object
```

Figure 2.2: Feature types analysis

Missing and null values

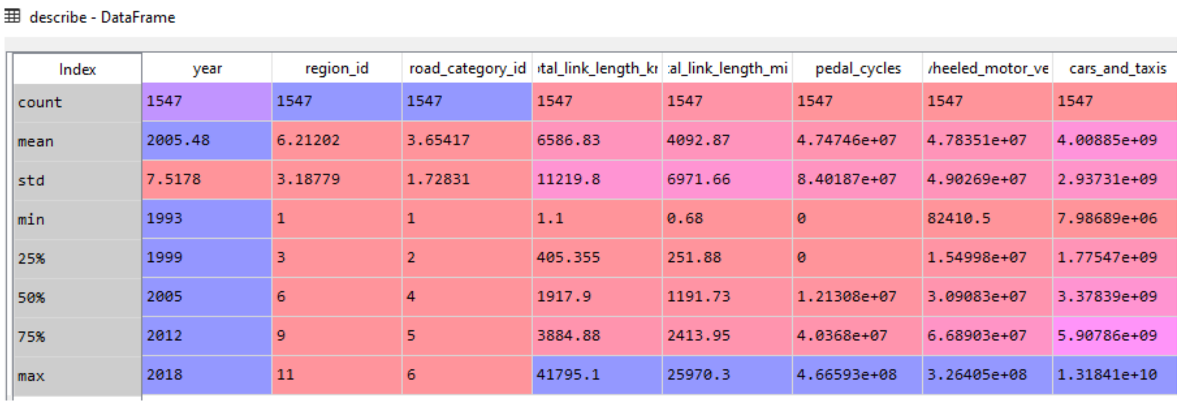


Index	year	region_id	name	ons_code	road_category_id	ıtal_link_length_kı	al_link_length_mi	pedal_cycles	/heeled_motor_ve	cars_and_taxis	ıuses_and_coache
1250	2014	1	South West	E12000009	2	0	0	0	0	0	0
1256	2014	2	East Midlands	E12000004	2	0	0	0	0	0	0
1262	2014	3	Scotland	S92000003	2	0	0	0	0	0	0
1268	2014	4	Wales	W92000004	2	0	0	0	0	0	0
1280	2014	6	London	E12000007	2	0	0	0	0	0	0
1281	2014	6	London	E12000007	3	0	0	0	0	0	0
1286	2014	7	East of England	E12000006	2	0	0	0	0	0	0
1316	2015	1	South West	E12000009	2	0	0	0	0	0	0
1322	2015	2	East Midlands	E12000004	2	0	0	0	0	0	0
1328	2015	3	Scotland	S92000003	2	0	0	0	0	0	0
1334	2015	4	Wales	W92000004	2	0	0	0	0	0	0
1346	2015	6	London	E12000007	2	0	0	0	0	0	0
1347	2015	6	London	E12000007	3	0	0	0	0	0	0
1352	2015	7	East of	E12000006	2	0	0	0	0	0	0

Figure 2.3: Null values analysis

As shown in Figure 2.3, in the dataset there are no explicit missing values, but some road links have 0 km length and no data about traffic. Therefore these records are remove from the dataset.

Descriptive statistics


describe - DataFrame

Index	year	region_id	road_category_id	ıtal_link_length_kı	:al_link_length_mi	pedal_cycles	/heeled_motor_ve	cars_and_taxis
count	1547	1547	1547	1547	1547	1547	1547	1547
mean	2005.48	6.21202	3.65417	6586.83	4092.87	4.74746e+07	4.78351e+07	4.00885e+09
std	7.5178	3.18779	1.72831	11219.8	6971.66	8.40187e+07	4.90269e+07	2.93731e+09
min	1993	1	1	1.1	0.68	0	82410.5	7.98689e+06
25%	1999	3	2	405.355	251.88	0	1.54998e+07	1.77547e+09
50%	2005	6	4	1917.9	1191.73	1.21308e+07	3.09083e+07	3.37839e+09
75%	2012	9	5	3884.88	2413.95	4.0368e+07	6.68903e+07	5.90786e+09
max	2018	11	6	41795.1	25970.3	4.66593e+08	3.26405e+08	1.31841e+10

Figure 2.4: Statistical description of data

As described in Figure 2.4, the dataset has 1547 records distributed between the year 1993 and 2018.

Categorical data

Many machine learning algorithms are not able to process categorical data, thus is necessary to encode the categorical features in the dataset. One way to do this is by using `sklearn.preprocessing.OneHotEncoder`. This technique encodes nominal features giving an optimal outcome, it allows any kind of data to be represented as a binary variable (Massaron and Boschetti, 2016). Using other encoders like `sklearn.preprocessing.LabelEncoder` for some kind of data might lead to the common mistake of giving a particular order to nominal data which will not give the possibility to the algorithm to give an optimal result. The code below gives an example on how to encode categorical data like the region `name`, but as the column `region_id` carries the same information, this operation might not be needed.

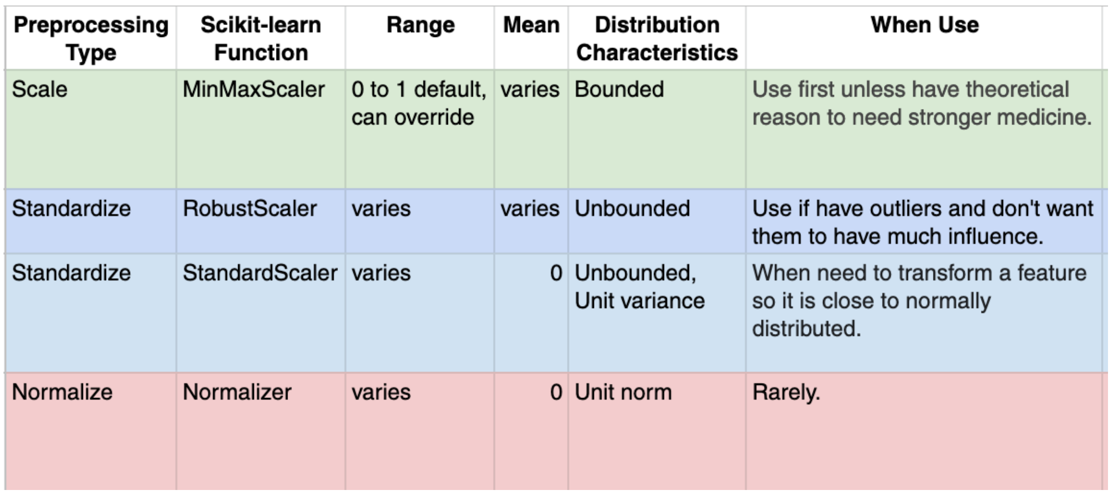
```
// Preprocessing.py
    from sklearn.preprocessing import OneHotEncoder

    # creating instance of one-hot-encoder
    enc = OneHotEncoder(handle_unknown='ignore')

    #encoding feature regionName
    enc_df =
        pd.DataFrame(enc.fit_transform(frame[[regionName]]).toarray())

    # merge with main df frame on key values
    frame = frame.join(enc_df)
```

Feature scaling

As we notice many features have different ranges, for example `region_id` ranges from 1 - 11 while `pedal_cycles` ranges from 0 - 4.6×10^8. While decision trees and random forest are scale independent, many machine learning algorithms are more efficient if features have similar ranges (Raschka and Mirjalili, 2017). When features are on different scales, even if the algorithm will process them separately, features with larger values will influence more the outcome. In case algorithms like *gradient descent* or *multivariate* are applied to the dataset, normalization or standardization will be needed because the accuracy score will vary significantly based on that. Normalization rescales all the values to the range of $[0, 1]$, which is a special case of min-max scaling. The code below gives an example on min-max scaling applied to the traffic dataset using `sklearn.preprocessing.MinMaxScaler`.

```
// Preprocessing.py
    from sklearn.preprocessing import MinMaxScaler

    #scale features using min-max
    frame.iloc[:,:] = MinMaxScaler().fit_transform(frame.iloc[:,:])
```

Standardization instead centres the feature mean at 0 and standard deviation at 1 so that the column has the form of a standard distribution. This can be useful when using ML algorithms that initialise the weights to zero, as the algorithm will learn the weights easier (Massaron and Boschetti, 2016). Table 2.1 suggests which scaler to use based on different characteristics of the problem to solve.



Preprocessing Type	Scikit-learn Function	Range	Mean	Distribution Characteristics	When Use
Scale	MinMaxScaler	0 to 1 default, can override	varies	Bounded	Use first unless have theoretical reason to need stronger medicine.
Standardize	RobustScaler	varies	varies	Unbounded	Use if have outliers and don't want them to have much influence.
Standardize	StandardScaler	varies	0	Unbounded, Unit variance	When need to transform a feature so it is close to normally distributed.
Normalize	Normalizer	varies	0	Unit norm	Rarely.

Table 2.1: Feature scaling approaches comparison (Hale, 2020)

2.2 Visualization

It is important to visualize the data to spot trends over different features and spot outliers. In 2018 the amount of vehicles miles travelled in the UK was 328.1 billion, 0.3% more than 2017. Table 2.2 show the distribution between different vehicle types.

```
+-----------------------------+------------------------+
| Vehicle type                |   Billion vehicle miles |
|-----------------------------+------------------------|
| all_motor_vehicles          |               328.115   |
| cars_and_taxis              |               255.013   |
| vans                        |                50.9832  |
| lorries                     |                17.0826  |
| pedal_cycles                |                 3.32911 |
| two_wheeled_motor_vehicles  |                 2.73903 |
| buses_and_coaches           |                 2.29719 |
+-----------------------------+------------------------+
```

Table 2.2: Summary of road traffic by vehicle type in 2018

The graph in Figure 2.5 describes how the trends across different types of vehicles compare over the years. To do so data has been scaled and modified in order to represent the trend pattern rather than actual values.

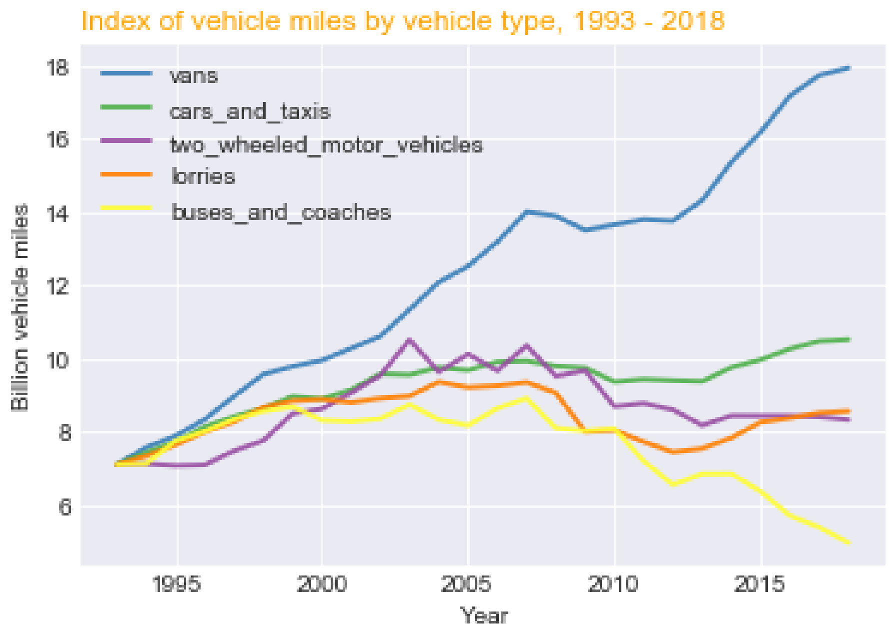


Figure 2.5: Index of vehicle miles by vehicle type, 1993 - 2018

Analysing the main trends from the graph, vans traffic has seen the fastest growth while the largest decrease was seen by buses and coaches traffic. The trend in vans in the last 20 years has clearly followed the economy (gross domestic product), in fact the 2008 crisis corresponds to the decrease that we notice in the graph. Also growth in internet shopping and home deliveries might have boosted the vans traffic on the roads. Instead, the number of buses might have decreased caused by the number of passengers that each bus can carry: today's buses have more passengers travelling for each bus (according to government analysis). Also, the support of local authorities for public buses decreased by 46% since 2008 in areas outside London, with a decrease in the service. Lorry traffic has seen a slightly growth but still remains below the peak seen around 2005. Motorcycles traffic increased and reach a peak in 2003, then went down again during the last decade. As show in the graph, in 2018, cars and taxis along with vans are the dominant part of traffic across Great Britain. The code below shows how data has been manipulated and plotted:

```
\\ visualization.py
    nFrame.iloc[:,-7:] = MinMaxScaler().fit_transform(nFrame.iloc[:,-7:])
    nFrame=nFrame.groupby('year').sum().reset_index()

    #determine which type of vehicle has the min number across all types
        in 1993
    baseCol = df.idxmin(axis=1) #Get Column name
    base = df[baseCol[0]].min() #get first year min value
    for column in df.drop('x', axis=1):
        diff = df[column].iloc[0]-base
        df[column] -= diff

    # create a color palette
    palette = plt.get_cmap('Set1')
    # multiple line plot
    num=0
    for column in df.drop('x', axis=1):
        num+=1
        plt.plot(df['x'], df[column], marker='', color=palette(num),
            linewidth=2, alpha=0.9, label=column)
```

Next graph will show how the traffic composition has changed over 25 years (between 1993 and 2018). While cars continue to be the main contributor to traffic across all vehicle types, it's clear that also vans traffic has increased significantly. In 1993 vans accounted for 10% of total traffic (25 billion out of 250) comparing to 2018 16% (51 billion out of 328).

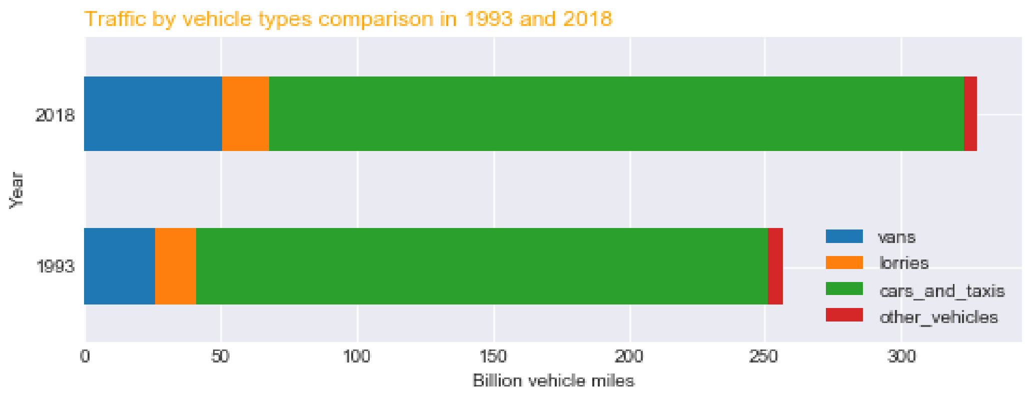


Figure 2.6: Traffic by vehicle type, 1993 vs 2018

The two scatterplot matrix shown in figure 2.7 and 2.8, will help to analyse the distribution of the features (Raschka and Mirjalili, 2017). A linear relationship is expected in the first matrix, where different types of vehicles are compared. In the second matrix other features from the dataset are plotted, the relationship between `total_link_length_km` and `all_vehicles` column is linear.

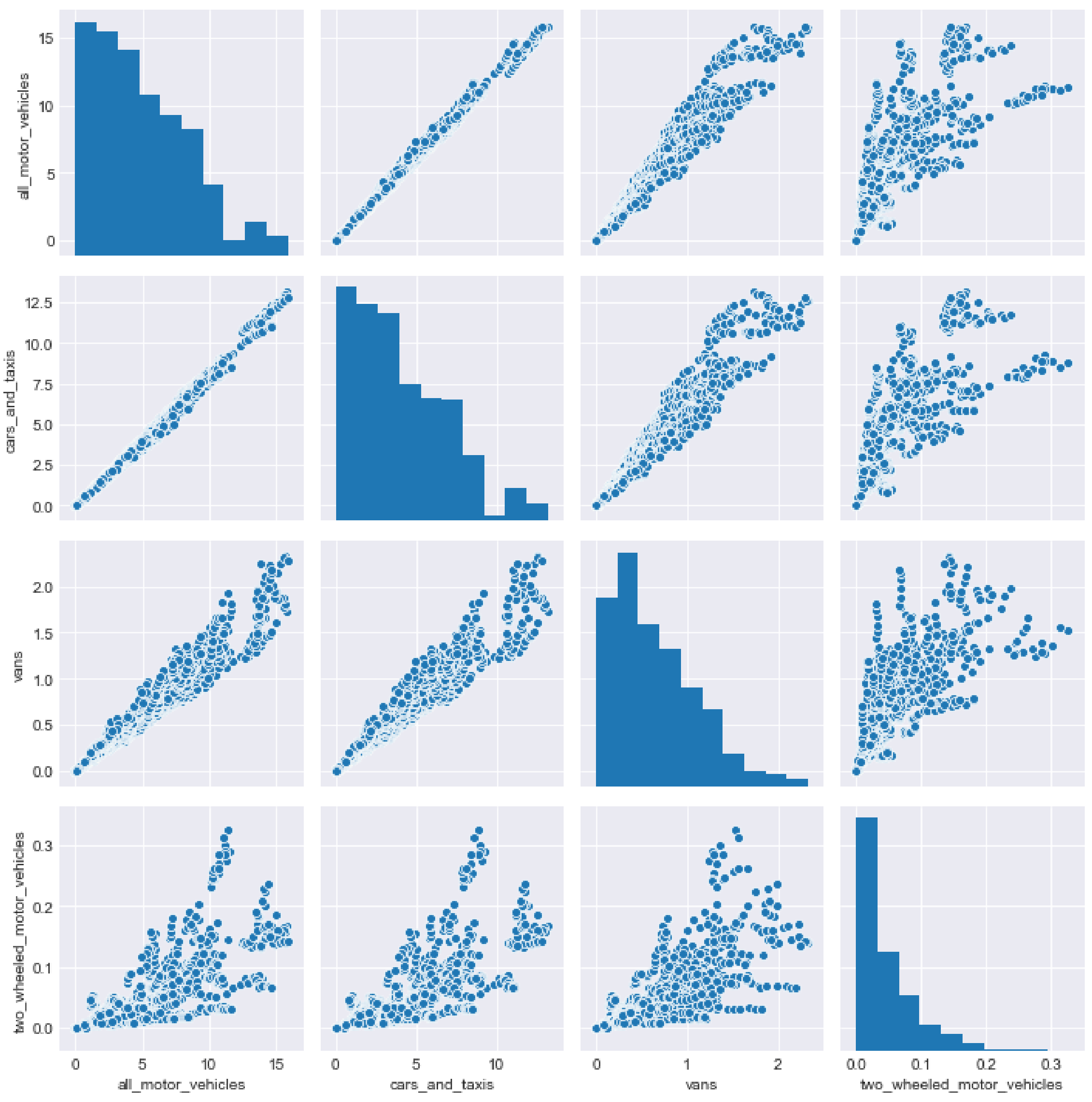


Figure 2.7: Summary of the relationships of the dataset, part 1

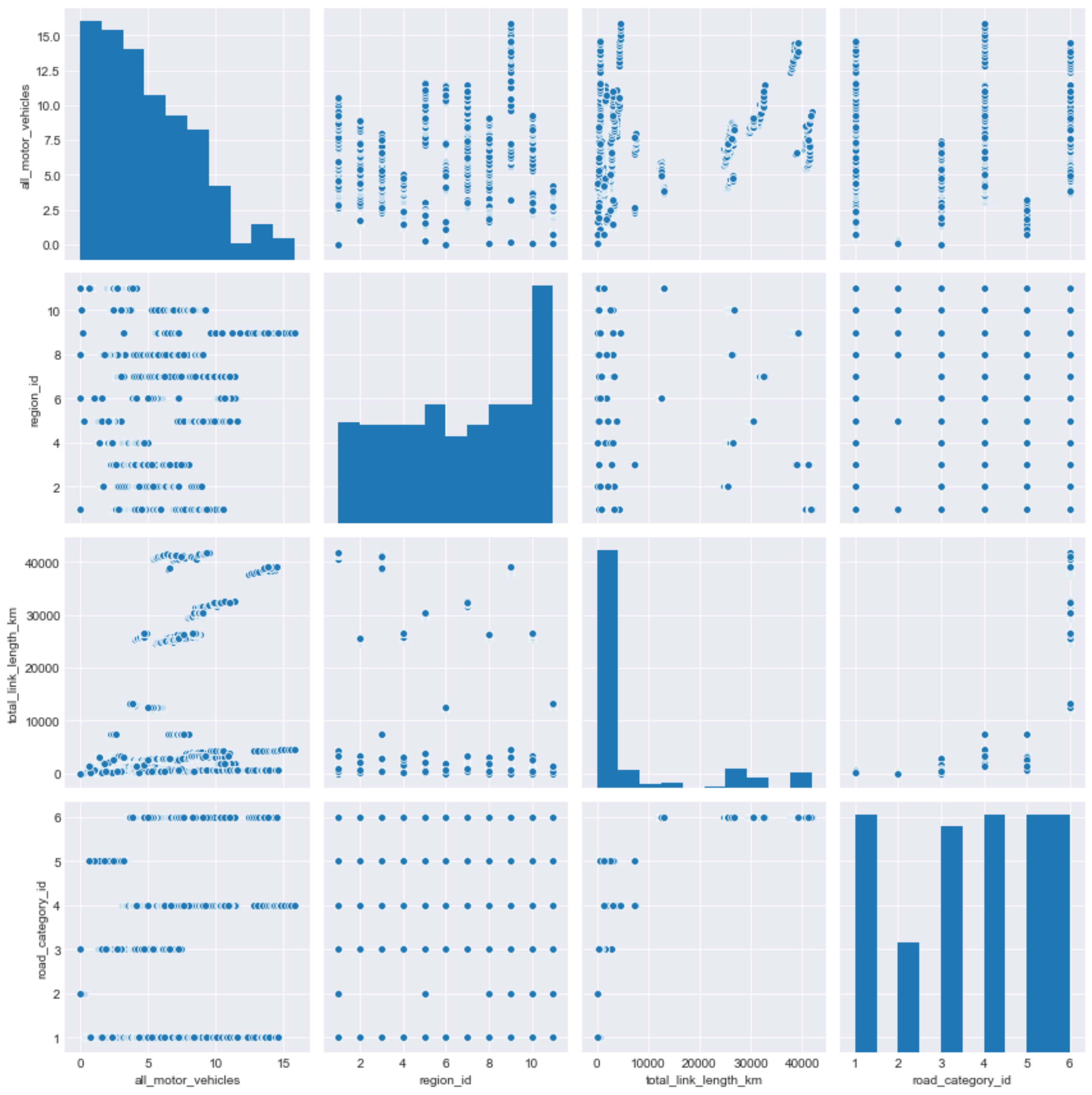


Figure 2.8: Summary of the relationships of the dataset, part 2

Chapter 3

Machine learning models

3.1 Multiple Linear regression model

3.1.1 Correlation

As shown in figure 2.7, there are several features that are linearly correlated with the target. The correlation matrix in figure 3.1 and the correlation table 3.1 quantify this relationships in the dataset. The correlation matrix contains the Pearson product-moment correlation coefficient which gives the value of the linear dependency between features. The correlation table shows the correlation value between the target and each feature. This will help selecting the features that will give the best score when applying a linear model. Selecting features that do not impact on the target variable will lead to a lower score, know as garbage in-garbage out rule (Srinivasa-Desikan, 2018). There are different methods in feature selection, such as Backward Elimination or Recursive Feature Elimination, in this case the filtering method will be applied.

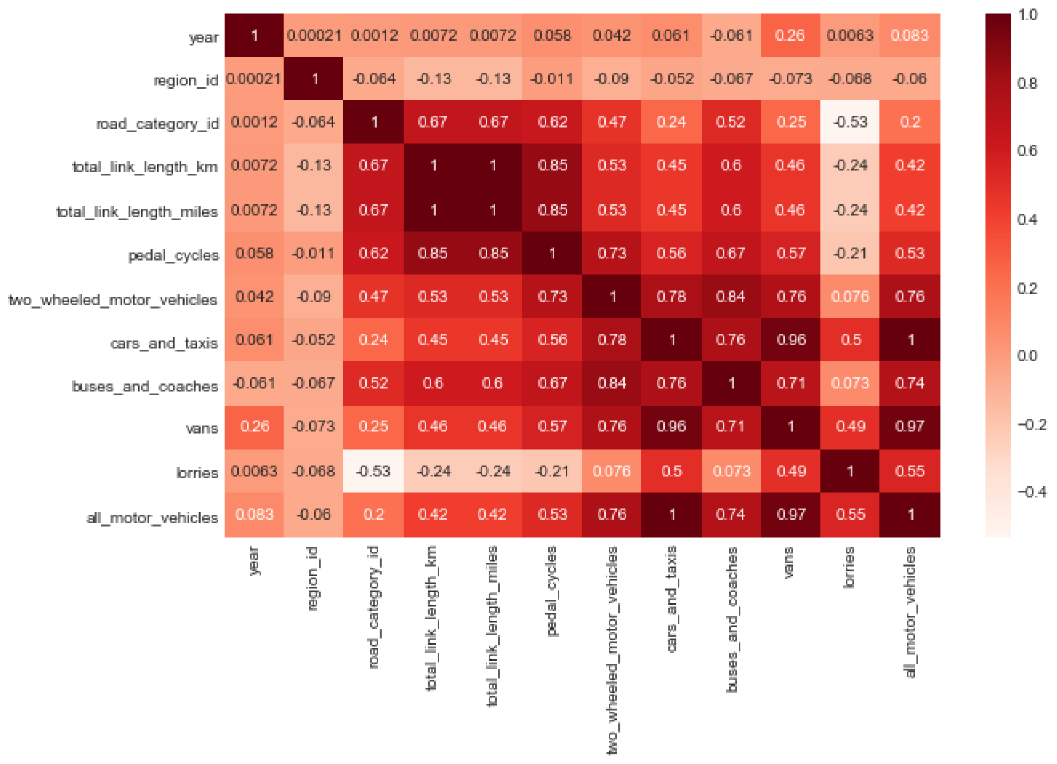


Figure 3.1: Correlation matrix

```
Correlation with target variable
all_motor_vehicles            1.000000
cars_and_taxis                0.997343
vans                          0.967801
two_wheeled_motor_vehicles    0.762308
buses_and_coaches             0.742106
lorries                       0.549614
pedal_cycles                  0.527216
total_link_length_miles       0.418928
total_link_length_km          0.418928
road_category_id              0.199224
year                          0.083343
region_id                    -0.059570
```

Table 3.1: Linear correlation between features and target

3.1.2 Test and training split

To evaluate the performance of the models applied, the data will be split into training set and test set with a proportion of 70% and 30%. The training set will be used to train the model, while the test set will be used to evaluate the predictions on unseen data. `Sklearn.model_selection.train_test_split` is used to randomly partition the dataset as the code below shows:

```
#split train and test set
validation_size = 0.3
X_train, X_test, y_train, y_test = train_test_split(x, y,
    test_size=validation_size)
```

3.1.3 Linear regression modelling

Linear regression's aim is to find the coefficients (θ) of an a-dimensional hyperplane that best fits the target variable. Given the regression function:

$$\hat{y} = w_0 + \sum_{j=1}^{a} w_j f_j(X_j)$$

where $\theta = \{w_0, w_1 ..., w_a\}$; w_0 determines the y-axis intercept of the hyperplane and $\{w_1, ..., w_a\}$ determine the slope. fjs are smooth functions not necessarily linear as the linearity is maintained in the parameters. In case of simple linear regression, when only one input feature is considered, this process can be represented as fitting the regression line through the sample points, minimising the offset between the line and the points. The dimension space $a + 1$ of the problem is given by the number of features, for example the shape of a 2-dimensional hyperplane with two input features and one target would look like this:

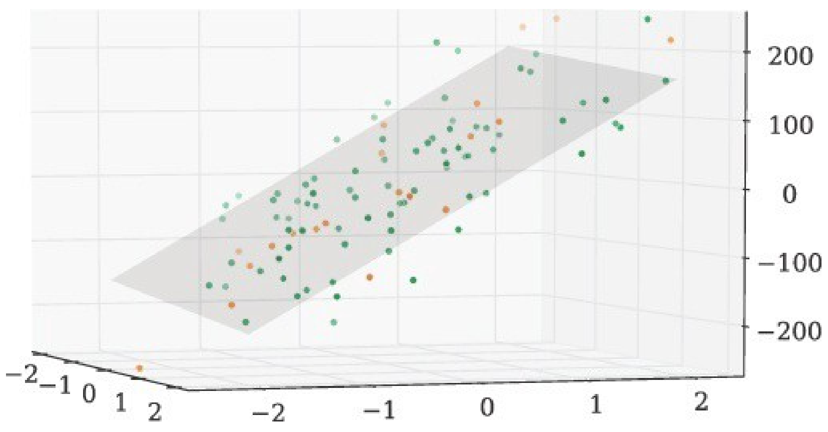


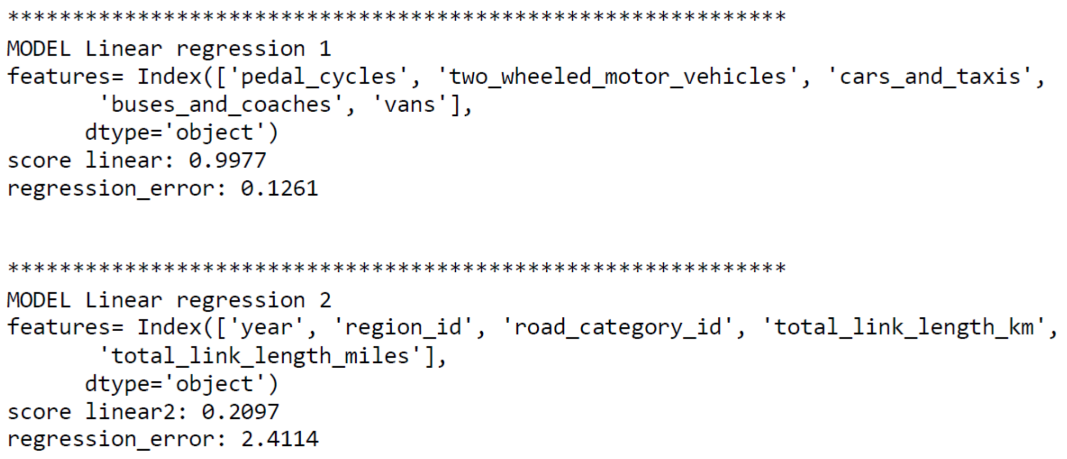
Figure 3.2: 2-dimesional fitted hyperplane (Govindaraj, 2018)

There are different techniques in fitting the model, in this casy the Ordinary Least square approach will be applied. The algorithm `LinearRegression` implemented in sklearn Python library, minimizes the residual sum of squares between the predicted and the actual values. The code below shows how the linear regression is implemented for the traffic dataset.

```
\\ model.py
    #apply regression model
    model_linear= LinearRegression(copy_X=True)
    model_linear.fit(X_train, y_train)
    y_pred = model_linear.predict(X_test)
    l_reg = model_linear.score(X_test, y_test)

    #show results of regression model
    print("score linear: %.4f" % l_reg)
    regression_error = mean_absolute_error(y_test, y_pred)
    print("regression_error: %.4f" % regression_error)
```

Evaluation

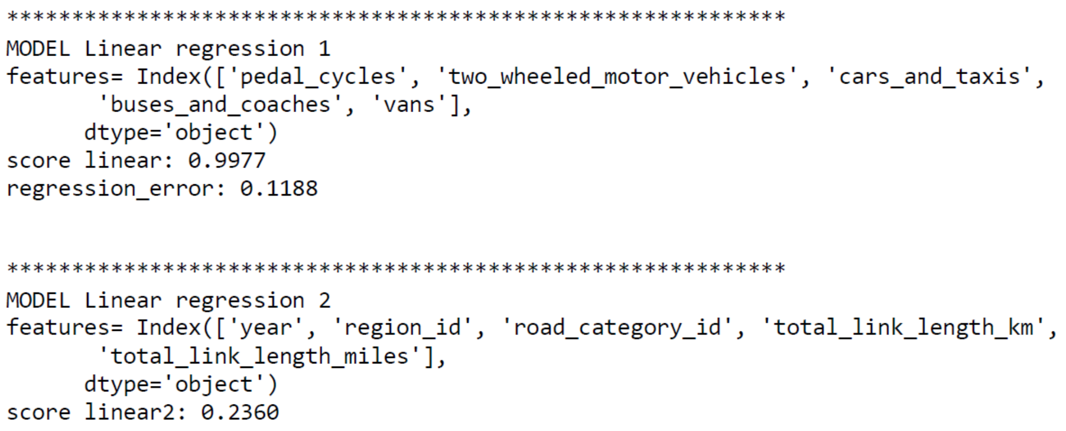
Figure 3.3 shows that the score when considering as input features some linearly correlated variables (different vehicle types) is higher (99%) than the same approach with `year`, `road_category`, `total_link_length_km` and `region_id` as input features (20%). When predicting a target that is linearly related with the input features this model performs good, but when considering non-linear correlated features it gives a low score.



```
************************************************************
MODEL Linear regression 1
features= Index(['pedal_cycles', 'two_wheeled_motor_vehicles', 'cars_and_taxis',
       'buses_and_coaches', 'vans'],
      dtype='object')
score linear: 0.9977
regression_error: 0.1261


************************************************************
MODEL Linear regression 2
features= Index(['year', 'region_id', 'road_category_id', 'total_link_length_km',
       'total_link_length_miles'],
      dtype='object')
score linear2: 0.2097
regression_error: 2.4114
```

Figure 3.3: Evaluation of linear regression

Also `sklearn.linear_model.LinearRegression` uses libraries and functionalities that work better with unstandardized data (Raschka and Mirjalili, 2017), in fact the results are slightly higher (99% and 23%):



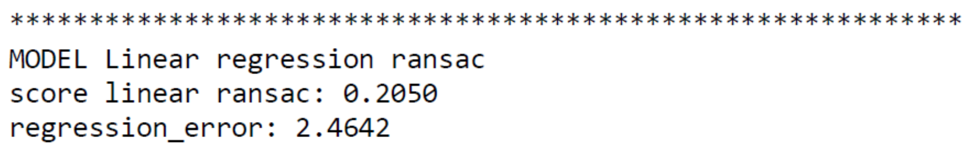
```
************************************************************
MODEL Linear regression 1
features= Index(['pedal_cycles', 'two_wheeled_motor_vehicles', 'cars_and_taxis',
       'buses_and_coaches', 'vans'],
      dtype='object')
score linear: 0.9977
regression_error: 0.1188


************************************************************
MODEL Linear regression 2
features= Index(['year', 'region_id', 'road_category_id', 'total_link_length_km',
       'total_link_length_miles'],
      dtype='object')
score linear2: 0.2360
```

Figure 3.4: Evaluation of linear regression with unstandardized data

3.1.4 Linear regression modelling with RANSAC

Linear regression algorithms performance is highly influenced by outliers. Instead of removing the outliers from the dataset, the RANdom SAmple Consensus (RANSAC) algorithm will apply the model to the inliers. Figure 3.5 shows that, even if the outliers effect has been reduced, the accuracy score didn't improve. The code below show the implementation on the traffic dataset.

```
\\ model.py
   ransac = RANSACRegressor(LinearRegression(),
                            max_trials=100,
                            min_samples=50,
                            loss='absolute_loss',
                            residual_threshold=5.0,
                            random_state=0)
   ransac.fit(X_train, y_train)
   y_pred = ransac.predict(X_test)
   l_reg = ransac.score(X_test, y_test)
```



```
*************************************************************
MODEL Linear regression ransac
score linear ransac: 0.2050
regression_error: 2.4642
```

Figure 3.5: Evaluation of linear regression with RANSAC

3.2 Random forest regression

As noted in section 3.1.1, some features have non-linear relationships between them. To find a more suitable model for this problem, the random forest regressor will be applied to the dataset. The algorithm generates multiple decision trees (ensemble-based approach) and uses Bootstrap sampling to select the training set for each tree (Pal, 2016). Random forest regressors are expected to have higher accuracy than decision trees due to their randomness which lowers the prediction variance (Breiman, 2017). Figure 3.6 summarises the structure of a random forest regressor.

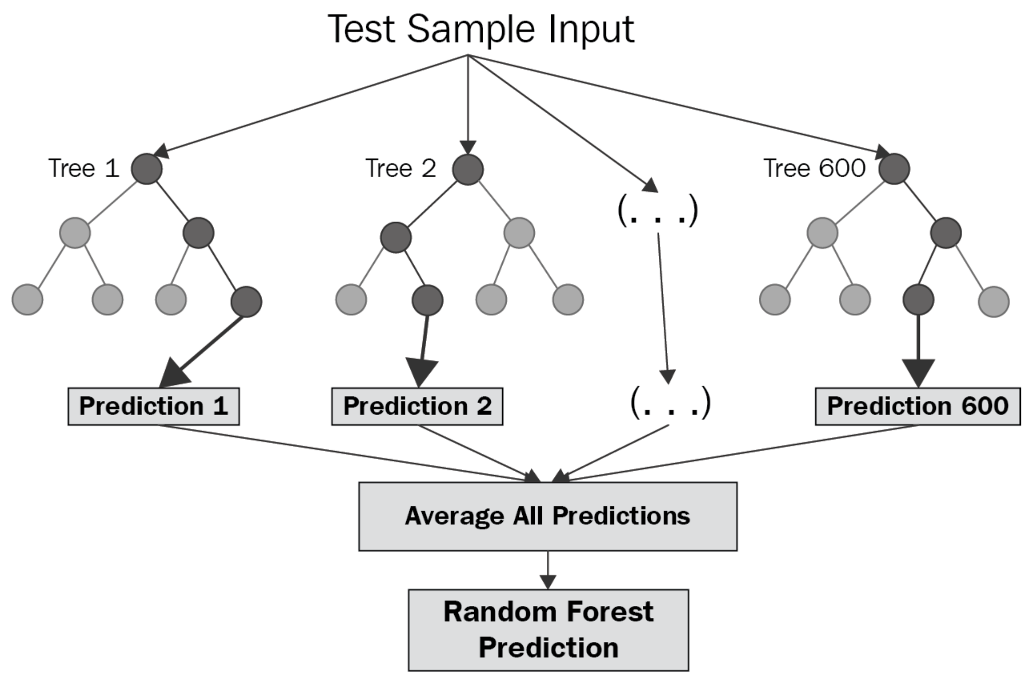


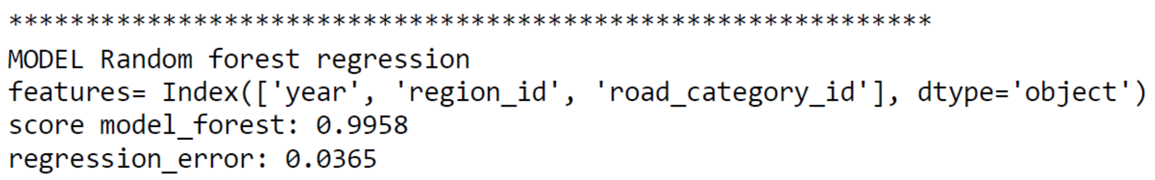
Figure 3.6: Random forest regression structure (Chakure, 2020)

3.2.1 Random forest train-test modelling

The code below show how the random forest is implemented for this dataset.

```
\\ model.py
    #apply regression model
    model_forest= RandomForestRegressor()
    model_forest.fit(X_train, y_train)
    y_pred = model_forest.predict(X_test)
    l_reg = model_forest.score(X_test, y_test)

    #show results of regression model
    print("score model_forest: %.4f" % l_reg)
    regression_error = mean_absolute_error(y_test, y_pred)
    print("regression_error: %.4f" % regression_error)
```

Figure 3.7 shows the accuracy score for this approach (99.58%)



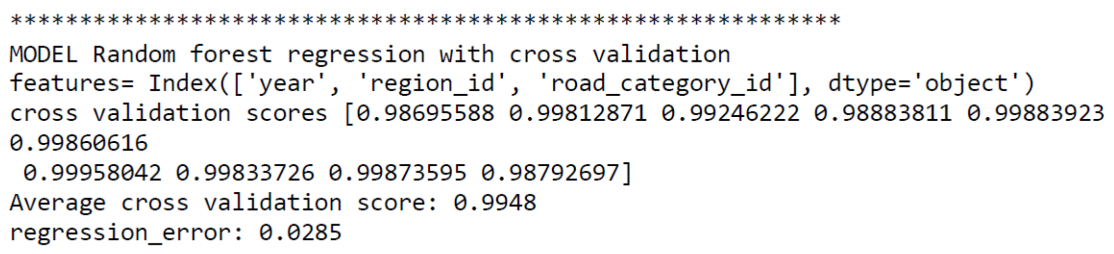
```
************************************************************
MODEL Random forest regression
features= Index(['year', 'region_id', 'road_category_id'], dtype='object')
score model_forest: 0.9958
regression_error: 0.0365
```

Figure 3.7: Evaluation of Random Forest Regression with train-test split

3.2.2 Random forest cross-validation modelling

Cross validation gives a slightly lower accuracy score (99.48%) than the previous train/test split approach.

```
\\ model.py
   #apply regression model
   model_forest= RandomForestRegressor()
   model_forest.fit(X, y)
   y_pred=model_forest.predict(X)
   scores= cross_val_score(model_forest, X, y, cv=10)

   #show results of regression model
   print("cross validation scores {}".format(scores))
   print("Average cross validation score: {:.4f}".format(scores.mean()))
   regression_error = mean_absolute_error(y, y_pred)
   print("regression_error: %.4f" % regression_error)
```



```
************************************************************
MODEL Random forest regression with cross validation
features= Index(['year', 'region_id', 'road_category_id'], dtype='object')
cross validation scores [0.98695588 0.99812871 0.99246222 0.98883811 0.99883923
0.99860616
 0.99958042 0.99833726 0.99873595 0.98792697]
Average cross validation score: 0.9948
regression_error: 0.0285
```

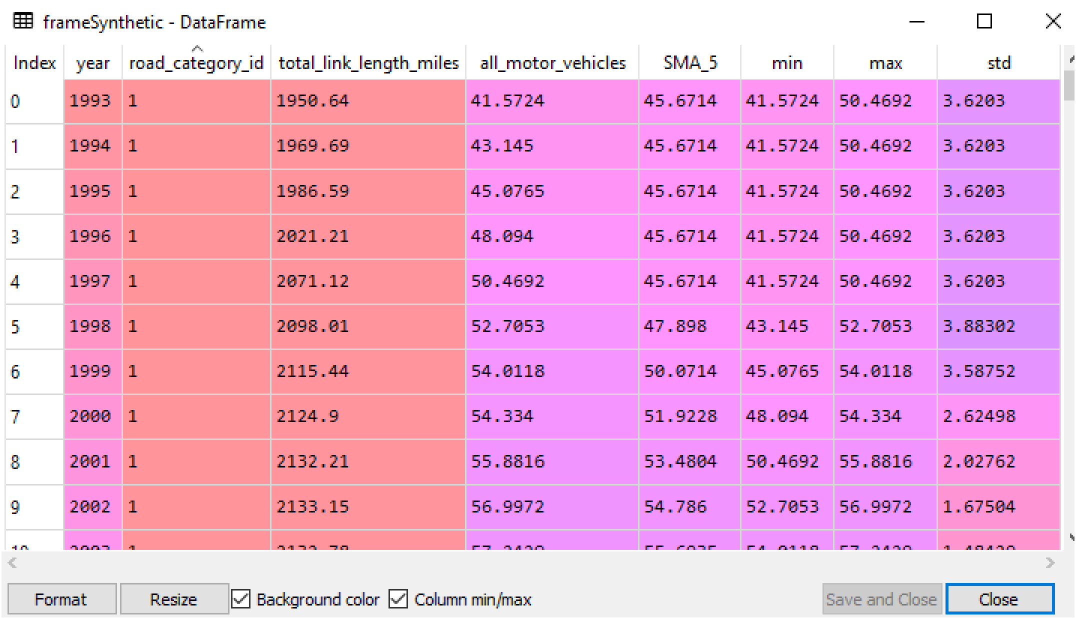
Figure 3.8: Evaluation of Random Forest Regression with cross-validation

3.2.3 Random forest with synthetic features modelling

The dataset has a limited number of features when it comes to predicting the total volume of vehicles on the roads. Knowing the number of pedal cycles, lorries, motorcycles etc does not give extra information about the dataset. As a solution to this it might be useful generate artificial features related to `all_vehicles`, for example mean, st. deviation, min and max values, per each sliding window. These are statistics over the same road category across all regions for each year. This is how it is implemented and how the result DataFrame looks like:

```
\\ util.py
   def add_synthetic_features(frame):
       nFrame=frame.groupby([year,
           roadCat]).sum().sort_values([roadCat,year],
           ascending=True).reset_index()
       categories = nFrame[roadCat].unique()

       newFrame=pd.DataFrame(columns=nFrame.columns)
       frames=[]

       for cat in categories:
           categorySlice=nFrame[nFrame[roadCat] == cat]
           window=categorySlice.iloc[:,11].rolling(window=5)
```

```
            categorySlice['SMA_5'] = window.mean()
            categorySlice['min'] = window.min()
            categorySlice['max'] = window.max()
            categorySlice['std'] = window.std()

            frames.append(categorySlice)

        newFrame = pd.concat(frames)
        newFrame.fillna( method ='bfill', inplace = True)
```


frameSynthetic - DataFrame

Index	year	road_category_id	total_link_length_miles	all_motor_vehicles	SMA_5	min	max	std
0	1993	1	1950.64	41.5724	45.6714	41.5724	50.4692	3.6203
1	1994	1	1969.69	43.145	45.6714	41.5724	50.4692	3.6203
2	1995	1	1986.59	45.0765	45.6714	41.5724	50.4692	3.6203
3	1996	1	2021.21	48.094	45.6714	41.5724	50.4692	3.6203
4	1997	1	2071.12	50.4692	45.6714	41.5724	50.4692	3.6203
5	1998	1	2098.01	52.7053	47.898	43.145	52.7053	3.88302
6	1999	1	2115.44	54.0118	50.0714	45.0765	54.0118	3.58752
7	2000	1	2124.9	54.334	51.9228	48.094	54.334	2.62498
8	2001	1	2132.21	55.8816	53.4804	50.4692	55.8816	2.02762
9	2002	1	2133.15	56.9972	54.786	52.7053	56.9972	1.67504

Figure 3.9: New DataFrame with synthetic features

The random forest regressor is applied to this new dataset to check if there is improvement in the accuracy score.

```
**************************************************************
MODEL Random forest regression with synthetic features
features= Index(['min', 'max', 'std', 'SMA_5', 'year', 'total_link_length_miles',
       'road_category_id'],
      dtype='object')
score model_forest: 0.9972
regression_error: 1.2390
```

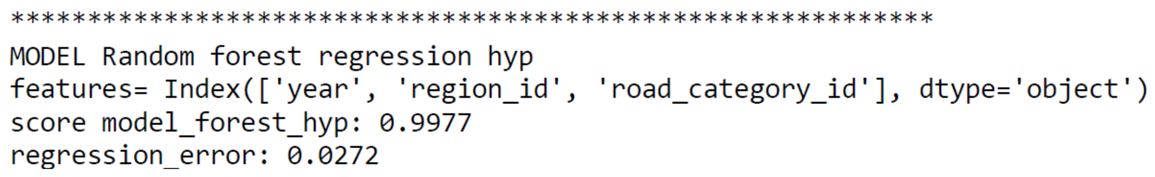
Figure 3.10: Accuracy score for random forest with synthetic features

The accuracy score improved from 99.48% to 99.72%.

3.2.4 Model tuning

To achieve a better score, the configuration of the model (hyperparameters) will be automatically tuned using `sklearn.model_selection.GridSearchCV`. This function will find the optimal hyper-parameters of the function and set them to get a better score. The code below shows how it's implemented.

```
// model.py
    n_test=[100]
    params_dict={'n_estimators':n_test, 'n_jobs':[-1], 'max_features':
        ["auto", "sqrt", "log2"]}

    model_forest_hyp=GridSearchCV(estimator=RandomForestRegressor(),
        param_grid=params_dict, scoring='r2')

    #apply regression model
    model_forest_hyp.fit(X_train, y_train)
    y_pred = model_forest_hyp.predict(X_test)
    l_reg = model_forest_hyp.score(X_test, y_test)
```



```
************************************************************
MODEL Random forest regression hyp
features= Index(['year', 'region_id', 'road_category_id'], dtype='object')
score model_forest_hyp: 0.9977
regression_error: 0.0272
```

Figure 3.11: Evaluation of Random Forest Regression with GridSearchCV tuning

This approach slightly improved the accuracy score as shown in figure 3.11. The execution time of this model increased significantly when using `GridSearchCV`. There are different ways to tune the hyperparameters such as `RandomizedSearchCV` or Bayesian Optimization. While `GridSearchCV` explores every combination of the hyperparameters to find the optimal one, `RandomizedSearchCV` selects random combinations of hyperparameters using a ”fit” and ”score” method, which makes this approach less accurate but more efficient than the first one. A smarter way to explore hyperparameters combinations is given by the Bayesian optimization. This approach takes into account the past iterations to learn from them. Bayesian optimization builds a probabilistic model of the objective function, called surrogate function, which is then searched efficiently.

`RandomizedSearchCV` implementation is similar to `GridSearchCV`, while Bayesian Optimisation need further description. The code below uses bayesian-optimization library from https://github.com/fmfn/BayesianOptimization, to implement the tuning for the traffic dataset.

```
def optimize_rfr(data, targets):
    #using https://github.com/fmfn/BayesianOptimization
    """Apply Bayesian Optimization to Random Forest parameters."""
    def rfr_crossval(n_estimators, max_features):
        return rfr_cv(
            n_estimators=int(n_estimators),
            max_features=max_features,
            data=data,
            targets=targets,
        )
    optimizer = BayesianOptimization(
        f=rfr_crossval,
        pbounds={
            "n_estimators": (10, 100),
            "max_features": (0.1, 0.999),
        },
        random_state=1234,
    )
    optimizer.maximize(n_iter=1)

    print("Final result:", optimizer.max)
```

Table 3.2 shows the performance of the three optimization techniques described above and applied to the traffic dataset.

Model tuning algorithm	Execution time	Accuracy Score
GridSearchCV	13.61s	99%
RandomizedSearchCV	8.38s	99%
Bayesian optimization	9.22s	97%

Table 3.2: Model tuning comparison

3.3 Support Vector Regression

In SV regression a tube with radius ϵ is fitted to the data as shown in figure 3.12. This kernel-based learning method is build upon a support-vector machine algorithm, which maps the training set into a higher-dimensional feature space and builds the optimal hyperplane (Schlkopf and Smola, 2001). There are different types of kernel, such as linear, polynomial or radial basis function (RBF).

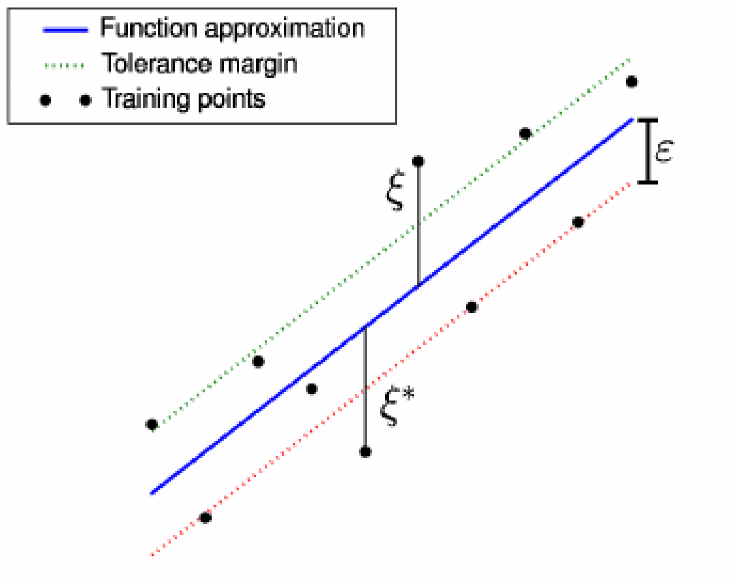


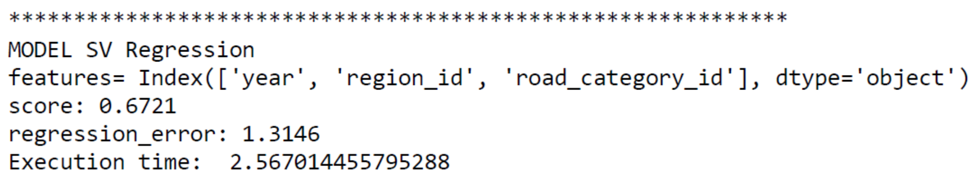
Figure 3.12: Support Vector Regression (Wise and Venter, 2009)

3.3.1 SVR Modelling

The `sklearn.svm.SVR` implements a Epsilon-Support Vector Regression where the default kernel is RBF. Below there is an implementation of the SV regression for the traffic dataset.

```
// model.py
    model_SVR = GridSearchCV(SVR(gamma=0.1),
                   param_grid={"gamma": np.logspace(-2, 2, 5)})

    #apply regression model
    model_SVR.fit(X_train, y_train)
    y_pred = model_SVR.predict(X_test)
    l_reg = model_SVR.score(X_test, y_test)
```

The accuracy score is 67% with hyperparameters already tuned. To keep the execution time low, just one hyperparameter is tuned.



```
*******************************************************
MODEL SV Regression
features= Index(['year', 'region_id', 'road_category_id'], dtype='object')
score: 0.6721
regression_error: 1.3146
Execution time:  2.567014455795288
```

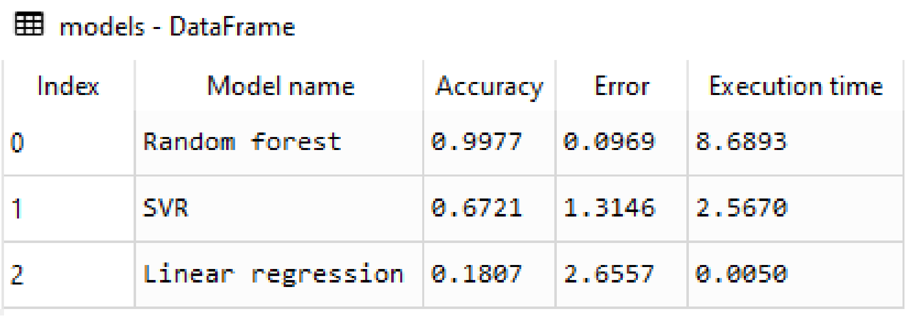
Figure 3.13: Accuracy score Support Vector Regression

Chapter 4

Results

Across all different settings tested for each model approach, table 4.1 shows the accuracy score, mean square error and execution time (in seconds) for each selected model. The objective was to predict `all_vehicles` based on `year`, `region_id`, `road_category` and other additional features such as `link_length` or synthetic features. For Random Forest Regressor and SVR the best (execution time/score) settings configuration was determined with `GridSearchCV`.

```
\\main.py
    # the model functions return (accuracy, error, exec time)
    # which are assigned to 3 variables a, b, c

    a=model_forest_hyp(frame)
    b=model_SVR(frame)
    c=model_linear2(frame)

    models=pd.DataFrame({
        "Model name":["Random forest", "SVR", "Linear regression"],
        "Accuracy": [a[0], b[0], c[0]],
        "Error":[a[1], b[1], c[1]],
        "Execution time":[a[2], b[2], c[2]]
        })
```


models - DataFrame

Index	Model name	Accuracy	Error	Execution time
0	Random forest	0.9977	0.0969	8.6893
1	SVR	0.6721	1.3146	2.5670
2	Linear regression	0.1807	2.6557	0.0050

Table 4.1: Models comparison

While Linear regression is very efficient in terms of time, it was not the best algorithm to model non-linear features. SVR to achieve a high score (>90%) needed more hyperparameter tuning which required a higher execution time

(>70s). That's why the final approach we chose was with less model tuning and lower score (67%) to achieve a good execution time. Random forest with model tuning achieves an excellent score in modelling non-linear features with a good execution time.

Chapter 5

Conclusion

The approaches proposed in this study to predict the level of traffic on UK roads were based on the distribution of different features, the only linearly correlated with the target where useful when predicting with Linear Regression, but this regression approach was not able to predict with high accuracy based on the remaining features. By comparing all the approaches analysed in this study, Random forest regression is recommended for modelling traffic data. Future work could include a Bayesian optimization for random forest instead of `GridSearchCV`, to better the performance in terms of execution time and control the model complexity.

Bibliography

Breiman, L. (2017), *Classification and Regression Trees*, hardcover edn, CRC Press.

Chakure, A. (2020), 'Random Forest Regression', *Medium* .
URL: *https://towardsdatascience.com/random-forest-and-its-implementation-71824ced454f*

Clark, S. (2003), 'Traffic Prediction Using Multivariate Nonparametric Regression', *Journal of Transportation Engineering* **129**(2), 161–168.

Department for Transport (2019), 'Road traffic estimates in Great Britain: 2018', *GOV.UK* .
URL: *https://www.gov.uk/government/statistics/road-traffic-estimates-in-great-britain-2018*

Do, L. N. N., Vu, H. L., Vo, B. Q., Liu, Z. and Phung, D. (2019), 'An effective spatial-temporal attention based neural network for traffic flow prediction', *Transportation Research Part C: Emerging Technologies* **108**, 12–28.

Govindaraj, P. (2018), 'Linear regression a glimpse in Data science / Machine learning', *Medium* .
URL: *https://medium.com/@praveengovi.analytics/linear-regression-a-glimpse-in-data-science-machine-learning-79e958633d16*

Hale, J. (2020), 'Scale, Standardize, or Normalize with Scikit-Learn - Towards Data Science', *Medium* .
URL: *https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02*

Massaron, L. and Boschetti, A. (2016), *Regression Analysis with Python*, Packt Publishing.

Ng, A. (2012), 'Lecture notes on cs229', *Stanford University Machine Learning module* .

Pal, R. (2016), *Predictive Modeling of Drug Sensitivity*, 1st edition edn, Academic Press.

Pandey, P. (2019), 'Data Preprocessing : Concepts - Towards Data Science', *Medium* .
URL: *https://towardsdatascience.com/data-preprocessing-concepts-fa946d11c825*

Raschka, S. and Mirjalili, V. (2017), *Python Machine Learning - Second Edition*, Packt Publishing, Limited, Birmingham, UNITED KINGDOM.
URL: *http://ebookcentral.proquest.com/lib/bcu/detail.action?docID=5050960*

Schlkopf, B. and Smola, A. J. (2001), *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, Adaptive Computation and Machine Learning, 1st edn, The MIT Press.

Srinivasa-Desikan, B. (2018), *Natural Language Processing and Computational Linguistics*, Packt Publishing.

Wise, J. N. and Venter, G. (2009), 'Optimization of a Wind Turbine Blade using Support Vector Regression', *ResearchGate* .