Car Accident Analysis in the UK (2012 - 2014)

Ariston Lim (000491325)
Hung Nguyen (000493176)
Julio Candela (000489735)
Valdemar Hernandez (000489715)

Table of Contents

I. Introduction

Problem Definition

There were more than 400,000 accidents occuring in the UK from 2012 to 2014. This means that on average, there are around 365 accidents happening *daily.* Thus, there is an urgent need to understand and dig deeper into these accidents: how and why they happened. Fortunately, the UK Police Department stored comprehensive information about the accidents. From this we could perform data analysis and even predict when accidents *might* happen. There are 3 main focus in this report:

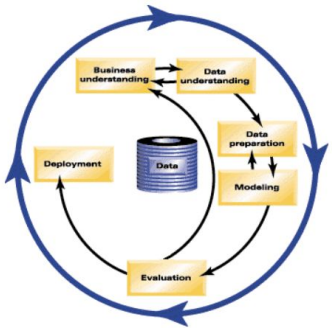
1. Identify accident hotspots.
2. Find the causes of fatal accidents and detect patterns in traffic conditions (if any)
3. Build a model to predict - and hence, prevent - fatal accidents.

Data Mining Framework

CRISP-DM methodology, which stands for “Cross-Industry Process for Data Mining”, is chosen for this project. It is widely used across all industries and provides a structured template in how to solve a data mining problem. Additionally, it provides flexibility and ease of customisation which makes it a really powerful framework. The lifecycle of this model consists of 6 phases:

1. Business understanding: determine objectives and goals
2. Data understanding: describe and explore data, understand what data is available
3. Data preparation: perform data cleansing and selection
4. Modeling: design models, select the best one, select best features, tune parameters
5. Evaluation: assess the result of model, decide to use current model or try other models
6. Deployment: deploy the model, monitor and review the results

The diagram below illustrates the connection among the phases:



The blue outer arrow represents the cyclical nature of CRISP-DM: the result of the deployment can be used to refine business understanding, which may trigger another project. The black inner arrows show the flexibility of this framework: the result of one phase can be used to modify other phases.

This report will be divided based on the phases of CRISP-DM. Below are the summary for each section:

1. *Business understanding:*
   The objective of this report is already described above in the problem definition.

2. *Data understanding:*
   We will take a look at the available data, take note of all the different features (columns), understand all the different features, and how the features relate to our goals.

   In this chapter, we will also answer Question 1, based on graphical analysis and clustering via DBSCAN. We will also perform bi-variate analysis.

3. *Data preparation:*
   We will clean the data - fix missing values, remove rows - and select which features that will be useful to answer the questions, i.e. select feature importance. By doing this, we will be able to answer Question 2.

4. *Modeling:*
   We will design and try multiple models. For each model, we will tune the parameters and try different techniques: undersampling/oversampling, use different features, etc. Next, we will select the best model and further refine it.

5. *Evaluation:*
   We will test the chosen model and - depending on the result - decide to continue with the current result or continue iterating to get a better performance.

6. *Deployment:*
   This step is skipped since there is no deployment for this project

Tools and Environment

We will work with python using multiple libraries such as:

a. Pandas and Numpy
b. DBSCAN
c. Scikitlearn (for preprocessing and algorithms)
d. Matplotlib
e. Lightgbm (algorithm)
f. Xgboost (algorithm)
g. Folium (maps)

II. Business Understanding

As mentioned above, this report focuses on 3 main topics:

1. Identify accident hotspots.
2. Find the causes of fatal accidents and detect patterns in traffic conditions (if any)
3. Build a model to predict - and hence, prevent - fatal accidents.

III. Data Understanding

To begin data exploration, we first load the csv file into pandas using the following command

```
df = pd.read_csv('accidents_clusterized.csv', low_memory=False)
```

This loads the csv file into a DataFrame, which makes data exploration easier.

Next, we check the data types of all the columns present in the data, using the following command

```
df.dtypes
```

This gives us the following result:

```
Accident_Index                                object
Location_Easting_OSGR                          int64
Location_Northing_OSGR                         int64
Longitude                                    float64
Latitude                                     float64
Police_Force                                   int64
Accident_Severity                              int64
Number_of_Vehicles                             int64
Number_of_Casualties                           int64
Date                                          object
Day_of_Week                                    int64
Time                                          object
Local_Authority_(District)                     int64
Local_Authority_(Highway)                     object
1st_Road_Class                                 int64
1st_Road_Number                                int64
Road_Type                                     object
Speed_limit                                    int64
Junction_Detail                              float64
Junction_Control                              object
2nd_Road_Class                                 int64
2nd_Road_Number                                int64
Pedestrian_Crossing-Human_Control             object
Pedestrian_Crossing-Physical_Facilities       object
Light_Conditions                              object
Weather_Conditions                            object
Road_Surface_Conditions                       object
Special_Conditions_at_Site                    object
Carriageway_Hazards                           object
Urban_or_Rural_Area                            int64
Did_Police_Officer_Attend_Scene_of_Accident    object
LSOA_of_Accident_Location                      object
Year                                           int64
```

Next, we want to see the distribution of the values for each column. First, we split the data into two groups: numerical and categorical. As a preliminary step, we only need the standard statistical values (count, unique value, frequency, avg, max, min, std, percentiles, etc).

For categorical data:

```
df.select_dtypes(include='object').describe()
```

The results are as follows:

	Accident_Index	Date	Time	Local_Authority_(Highway)	Road_Type	Junction_Control	Pedestrian_Crossing-Human_Control	Pedestrian_Crossing-Physical_Facilities
count	464697	464697	464684	464697	464697	286087	464697	464697
unique	263811	1096	1439	207	6	4	3	6
top	2.01E+12	25/05/2012	17:00	E10000016	Single carriageway	Giveway or uncontrolled	None within 50 metres	No physical crossing within 50 meters
freq	177752	748	4609	13033	351268	232915	462133	377099

For numerical data:

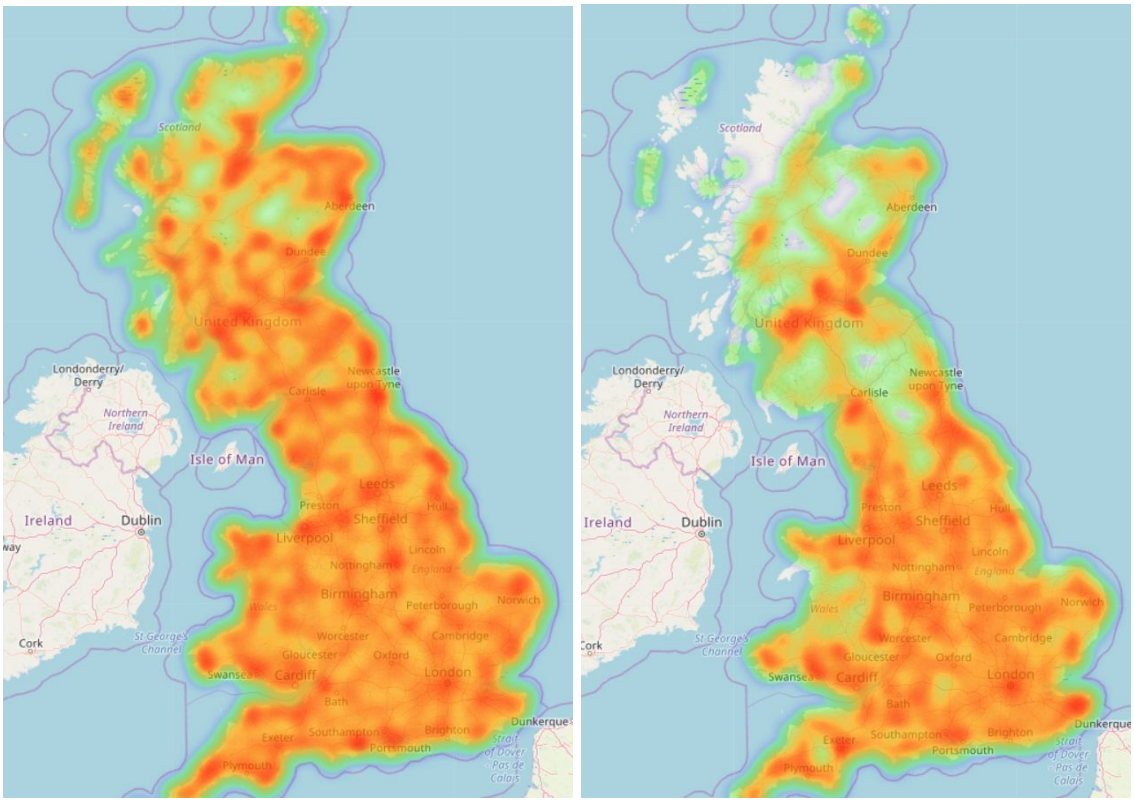
```
df.select_dtypes(exclude='object').describe()
```

The results:

	Unnamed: 0	Location_Easting_OSGR	Location_Northing_OSGR	Longitude	Latitude	Police_Force	Accident_Severity	Number_of_Vehicles
count	464697.000000	464697.000000	4.646970e+05	464697.000000	464697.000000	464697.000000	464697.000000	464697.000000
mean	232348.000000	443834.284222	2.986258e+05	-1.375156	52.575498	28.504051	2.833461	1.828086
std	134146.613358	94098.865933	1.594701e+05	1.382137	1.436370	25.334899	0.402029	0.708703
min	0.000000	65510.000000	1.029000e+04	-7.509162	49.912941	1.000000	1.000000	1.000000
25%	116174.000000	379059.000000	1.777100e+05	-2.315799	51.484841	6.000000	3.000000	1.000000
50%	232348.000000	445539.000000	2.606800e+05	-1.323374	52.232169	22.000000	3.000000	2.000000
75%	348522.000000	525350.000000	3.989590e+05	-0.192935	53.485973	45.000000	3.000000	2.000000
max	464696.000000	655370.000000	1.190858e+06	1.759382	60.597984	98.000000	3.000000	67.000000

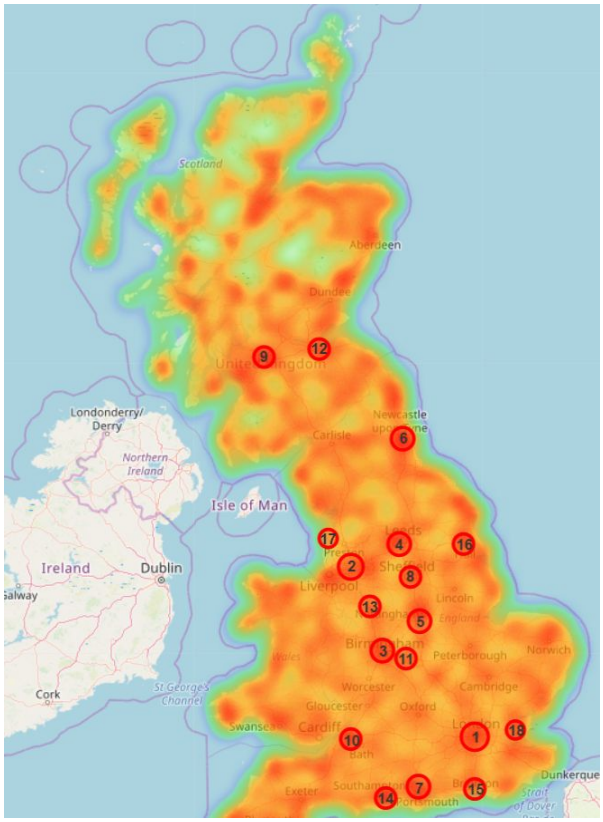
Accident Hotspots

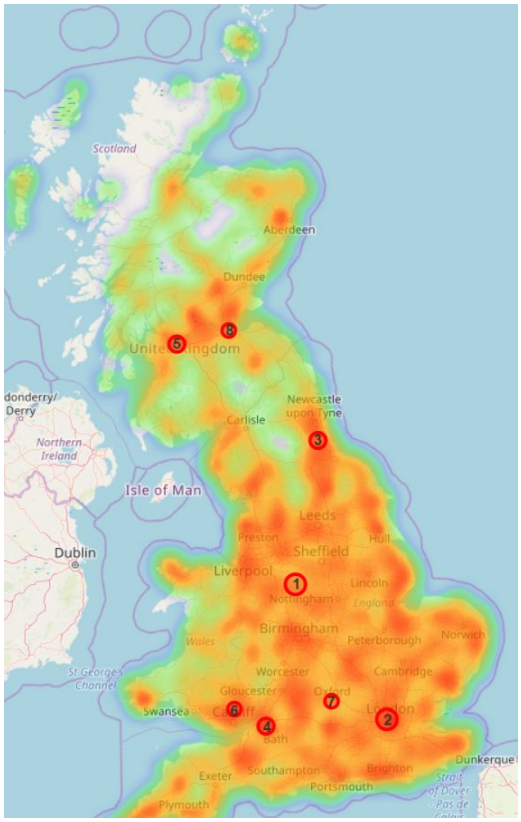
As a preliminary method, we created heatmaps for the total accidents and fatal accidents using the folium library. The results are as follows:



Left: All accidents Right: Fatal accidents

To get a better accuracy on the hot spots for the location, we performed a DBSCAN algorithm to cluster the accidents together. Initially, we use 100 000 data points and then we apply this model to the entire data set. We also calculate the centroids for these hot spots. The results are as follows:





Left: Clusters of all accidents Right: cluster of fatal accidents.

From this data, we can get the area with the most accidents, they are shown in the tables below.
We use eps of 0.12 and minimum points of 600.

Rank	**Area**	**Count**	**Fatal**	**%Fatal**
1	London	121271	831	0.69%
2	Liverpool	46700	451	0.97%
3	Birmingham	18229	209	1.15%
4	Bradford	17474	160	0.92%
5	Nottingham	16268	129	0.79%
6	Gateshead	13673	116	0.85%
7	Fareham	9045	58	0.64%
8	Sheffield	8316	74	0.89%

9	Glasgow	6228	62	1.00%
10	Bristol	5517	69	1.25%
11	Coventry	4518	53	1.17%
12	Stoke-on-Trent	4471	52	1.16%
13	Edinburgh	4401	38	0.86%
14	Bournemouth	3913	29	0.74%
15	Brighton	3847	33	0.86%
16	Blackpool	3557	28	0.79%
17	Hull	3460	26	0.75%
18	Rayleigh	2518	17	0.68%
-1	Unclustered	171291	2868	1.67%

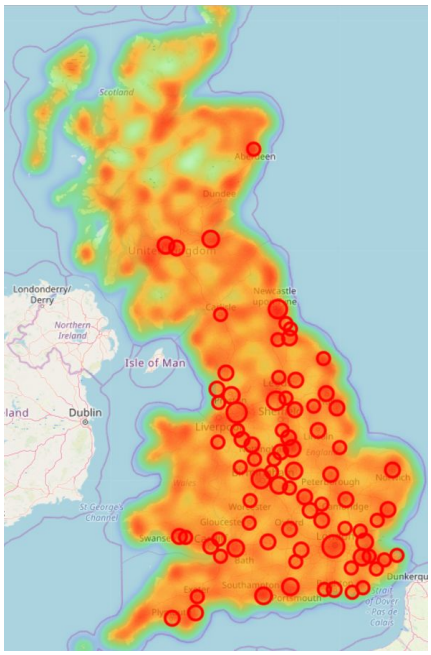
We did the same analysis for fatal accidents. To account for the smaller numbers of fatal accidents, we increase the eps to 0.2 and reduce the minimum points to 40.

Rank	Area	% of Total Fatal Accidents
1	Thorncliffe	29.8%
2	London	21.5%
3	Shincliffe	4.6%
4	Bristol	2.1%
5	Glasgow	1.7%
6	Mynyddislwyn	1.5%
7	Oxford	1.1%
8	Edinburgh	1.0%
-1	Unclustered	36.8%

We can see that 50% of all fatal accidents occur in Thorncliffe and London.

Next, we want to see more detailed clusters for the hotspots. To do this, we reduce the eps and minimum points to 0.05 and 100 points respectively.

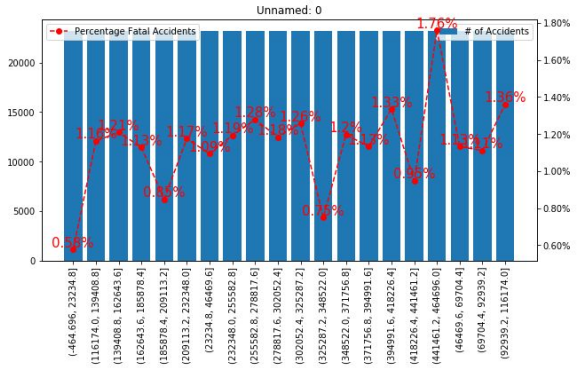
```
cluster_accidents = getHotSpotsAccidents(lat_lon, 'accidents_hs.html',0.05,100,1)
```

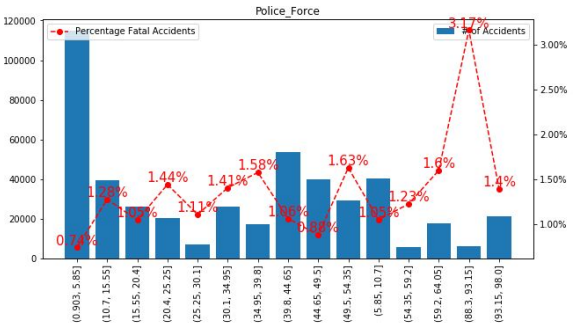


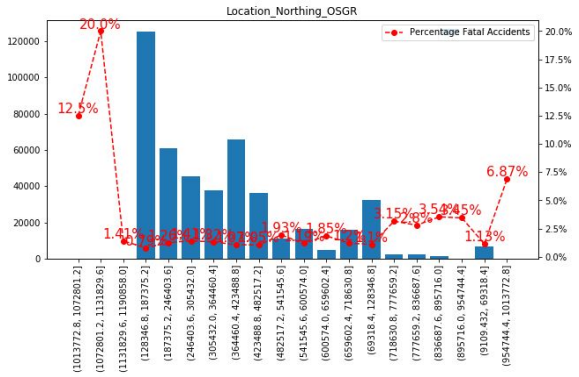
Bivariate Analysis

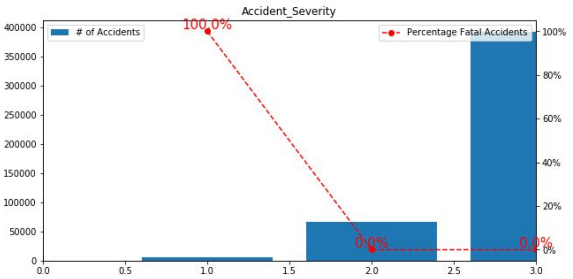
Bivariate analysis is the simultaneous analysis of two variables (attributes). It explores the concept of relationship between two variables, whether there exists an association and the strength of this association, or whether there are differences between two variables and the significance of these differences.

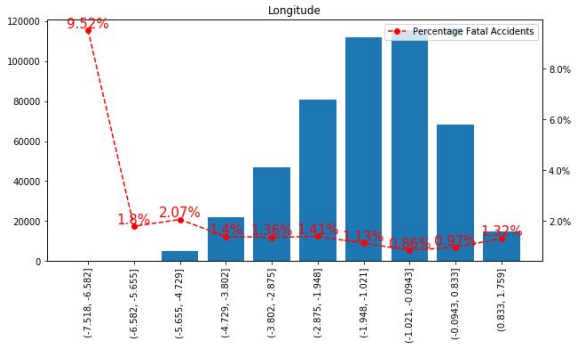
For this project, we perform bivariate analysis on all features against fatal accidents. The results are shown below.

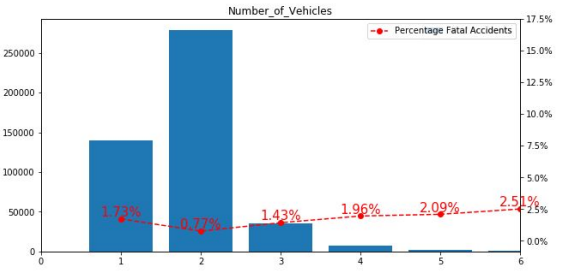

Unnamed: 0
Percentage Fatal Accidents
# of Accidents

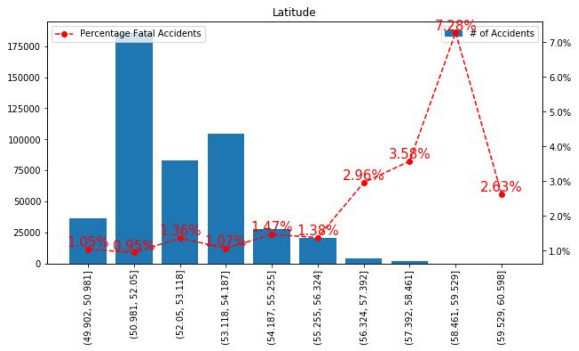

Police_Force
Percentage Fatal Accidents
# of Accidents

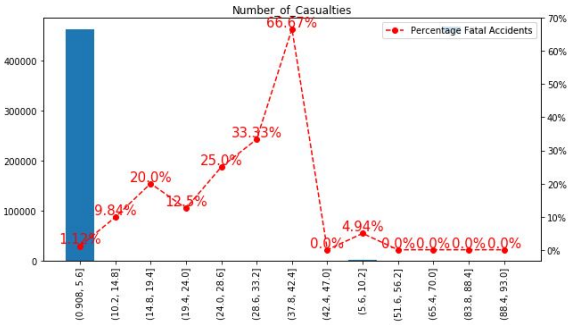

Location_Northing_OSGR
Percentage Fatal Accidents

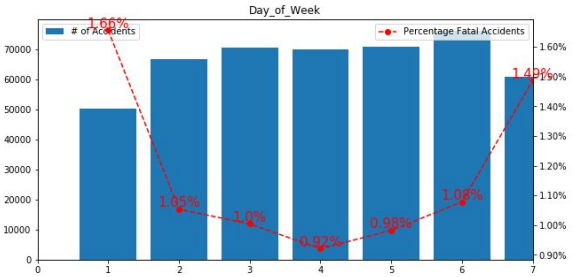

Accident_Severity
# of Accidents
Percentage Fatal Accidents

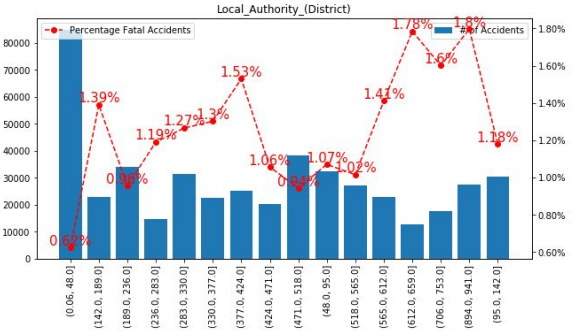

Longitude
Percentage Fatal Accidents

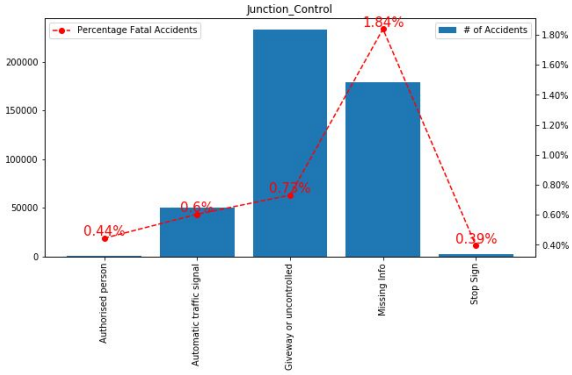

Number_of_Vehicles
Percentage Fatal Accidents

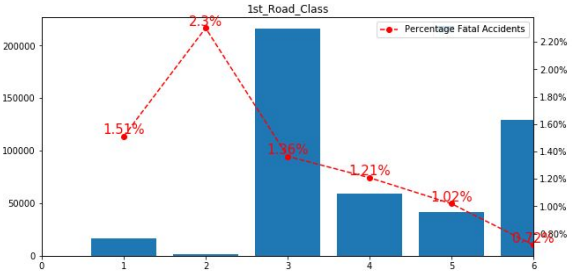

Latitude
Percentage Fatal Accidents
# of Accidents

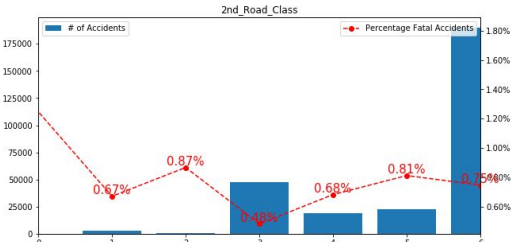

Number_of_Casualties
Percentage Fatal Accidents

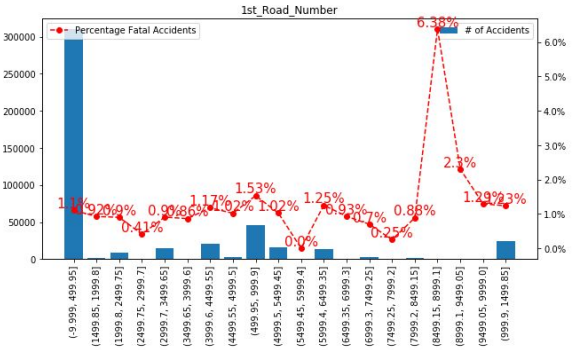

Day_of_Week
# of Accidents
Percentage Fatal Accidents

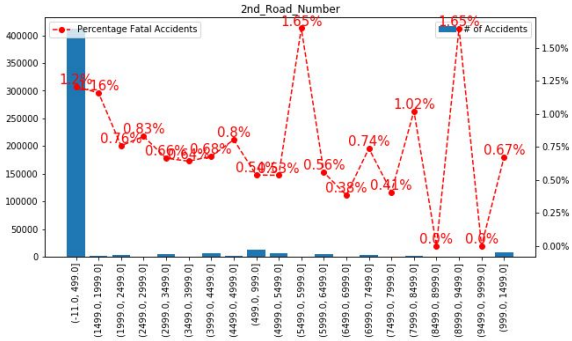

Local_Authority_(District)
Percentage Fatal Accidents
# of Accidents

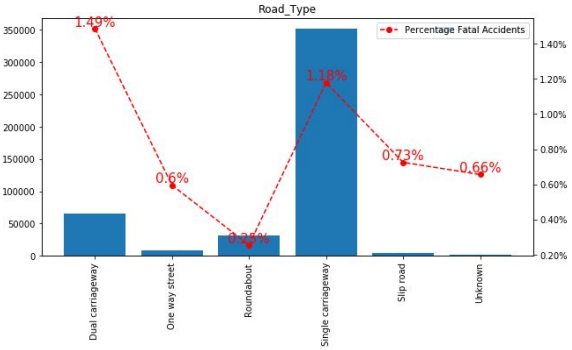

Junction_Control
Percentage Fatal Accidents
# of Accidents
Authorised person
Automatic traffic signal
Giveway or uncontrolled
Missing Info
Stop Sign

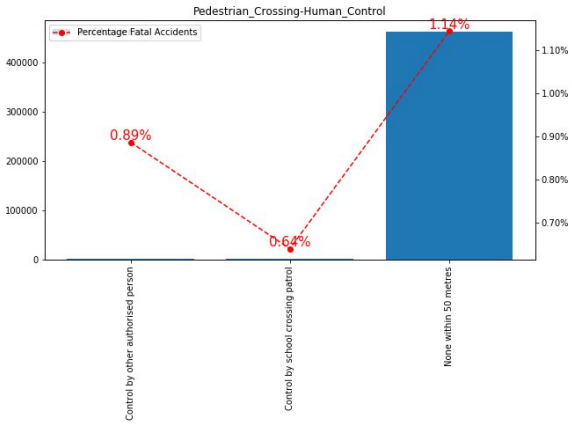

1st_Road_Class
Percentage Fatal Accidents

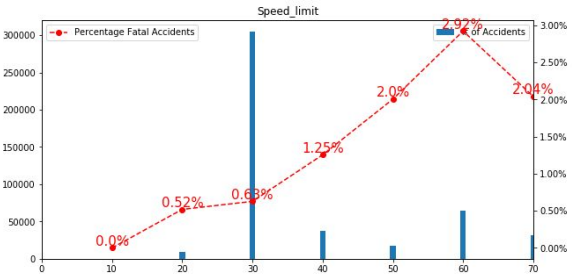

2nd_Road_Class
# of Accidents
Percentage Fatal Accidents

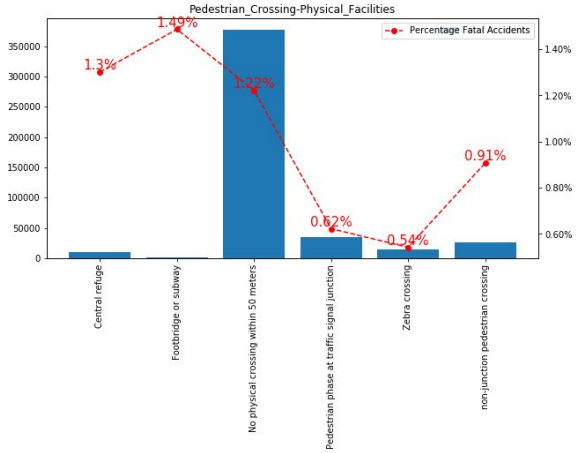

1st_Road_Number
Percentage Fatal Accidents
# of Accidents

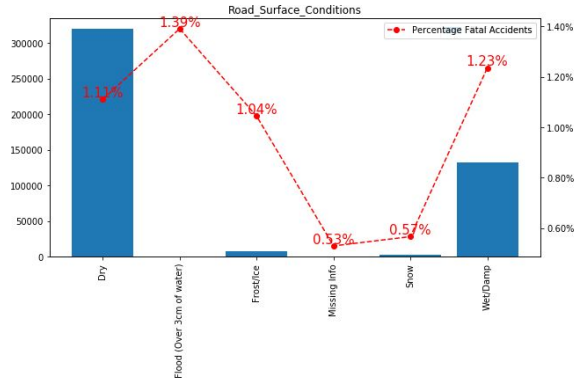

2nd_Road_Number
Percentage Fatal Accidents
# of Accidents

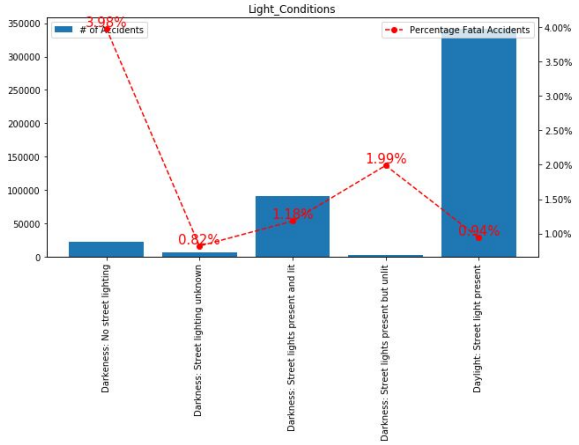

Road_Type
Percentage Fatal Accidents
Dual carriageway
One way street
Roundabout
Single carriageway
Slip road
Unknown

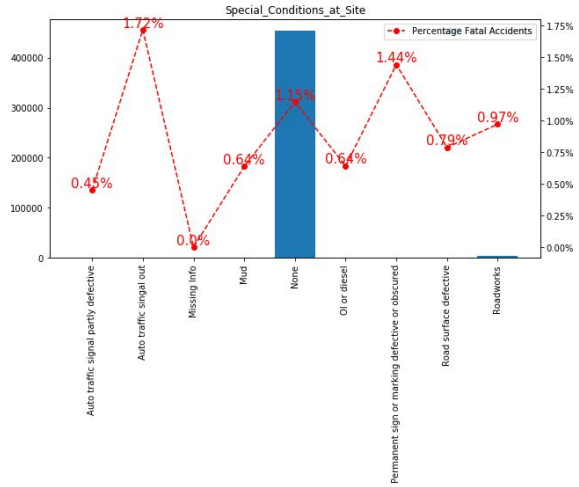

Pedestrian_Crossing-Human_Control
Percentage Fatal Accidents
Control by other authorised person
Control by school crossing patrol
None within 50 metres

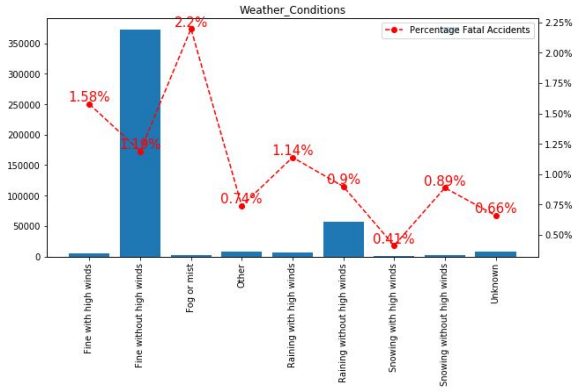

Speed_limit
Percentage Fatal Accidents
of Accidents

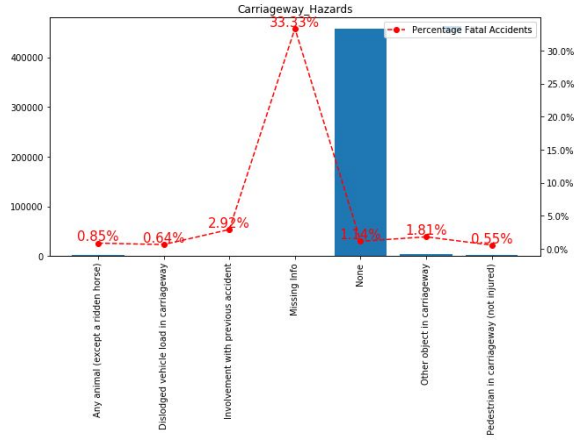

Pedestrian_Crossing-Physical_Facilities
Percentage Fatal Accidents
1.3%
1.49%
1.22%
0.62%
0.54%
0.91%
Central refuge
Footbridge or subway
No physical crossing within 50 meters
Pedestrian phase at traffic signal junction
Zebra crossing
non-junction pedestrian crossing

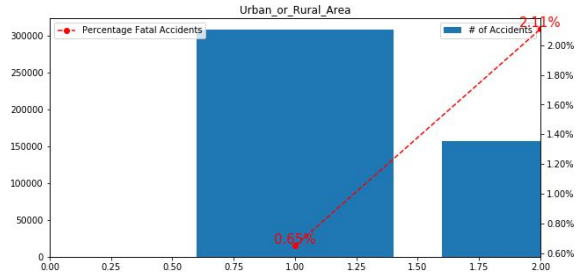

Road_Surface_Conditions
Percentage Fatal Accidents
1.11%
1.39%
1.04%
0.53%
0.57%
1.23%
Dry
Flood (Over 3cm of water)
Frost/Ice
Missing Info
Snow
Wet/Damp

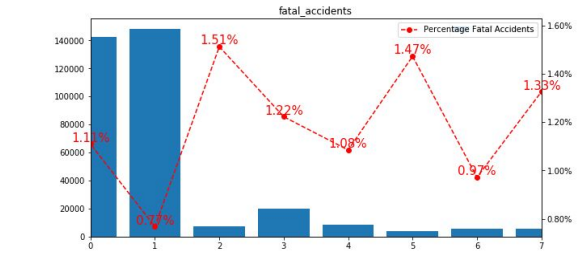

Light_Conditions
# of Accidents
Percentage Fatal Accidents
3.98%
0.82%
1.18%
1.99%
0.94%
Darkness: No street lighting
Darkness: Street lighting unknown
Darkness: Street lights present and lit
Darkness: Street lights present but unlit
Daylight: Street light present

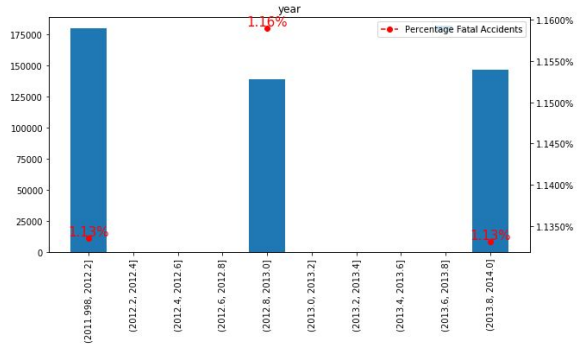

Special_Conditions_at_Site
Percentage Fatal Accidents
0.45%
1.72%
0.0%
0.64%
1.15%
0.64%
1.44%
0.79%
0.97%
Auto traffic signal partly defective
Auto traffic singal out
Missing Info
Mud
None
Oil or diesel
Permanent sign or marking defective or obscured
Road surface defective
Roadworks

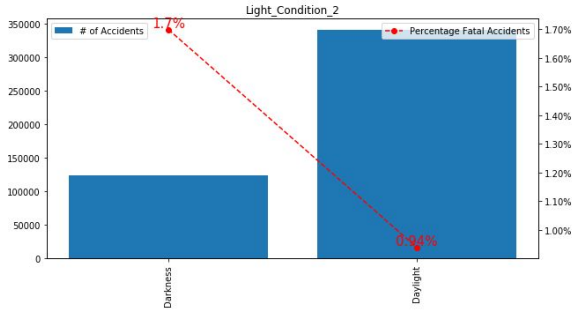

Weather_Conditions
Percentage Fatal Accidents
1.58%
1.74%
2.2%
0.74%
1.14%
0.9%
0.41%
0.89%
0.66%
Fine with high winds
Fine without high winds
Fog or mist
Other
Raining with high winds
Raining without high winds
Snowing with high winds
Snowing without high winds
Unknown

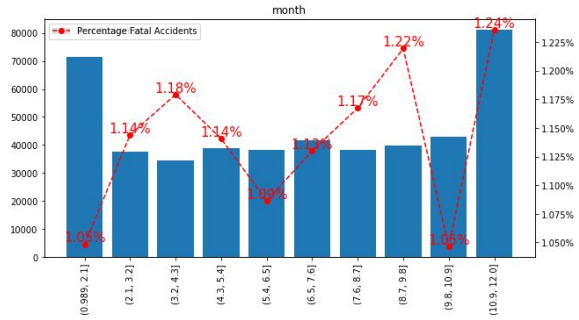

Carriageway_Hazards
Percentage Fatal Accidents
0.85%
0.64%
2.92%
33.33%
1.44%
1.81%
0.55%
Any animal (except a ridden horse)
Dislodged vehicle load in carriageway
Involvement with previous accident
Missing Info
None
Other object in carriageway
Pedestrian in carriageway (not injured)

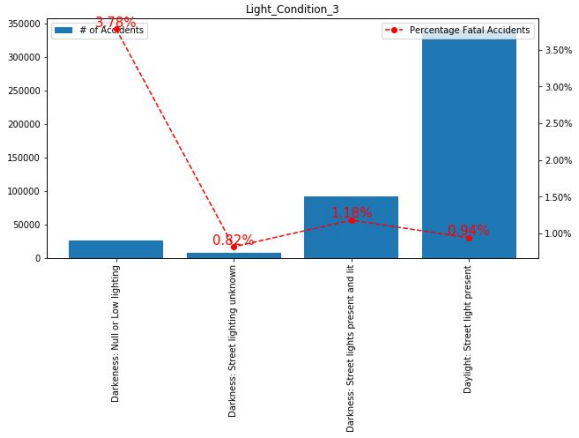

Urban_or_Rural_Area
Percentage Fatal Accidents
# of Accidents
0.65%
2.11%

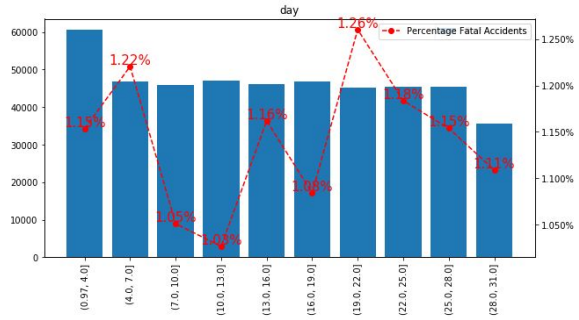


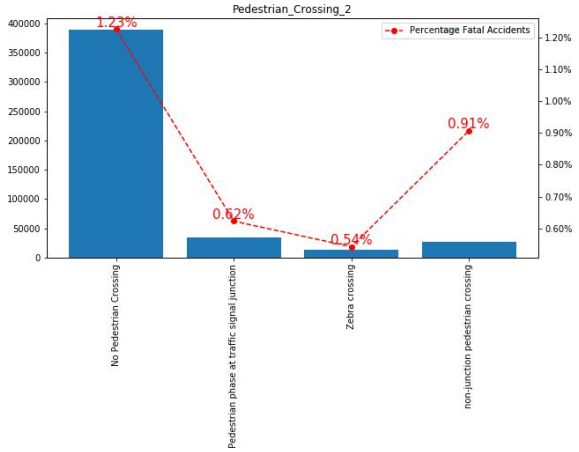


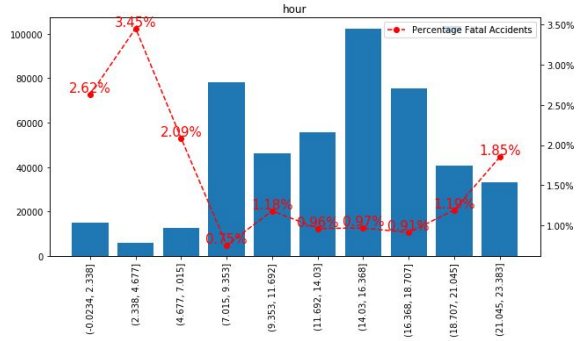












After doing this analysis, we discovered some features that need further pre-processing, such as missing values, splitting into smaller features, changing type from numerical to categorical, etc. Therefore, we will do further pre-processing for these features, which will be described below.

IV. Data preparation

Creating new features

Light_Condition_2 is set to 'Daylight' if there is street light lit, otherwise it is set to 'Darkness'.

Light_Condition_3 is set to 'Darkness: Null or Low lighting' if when it is dark and there is no street lights or the street lights unlit.

```
accidents_train['Light_Condition_2'] = np.where(accidents_train['Light_Conditions']
                                        == 'Daylight: Street light present' , 'Daylight', 'Darkness')
accidents_train['Light_Condition_3'] = np.where(accidents_train['Light_Conditions']
                                        == 'Darkeness: No street lighting' ,
                'Darkeness: Null or Low lighting', accidents_train['Light_Conditions'])
accidents_train['Light_Condition_3'] = np.where(accidents_train['Light_Condition_3']
                                        == 'Darkness: Street lights present but unlit' ,
                'Darkeness: Null or Low lighting', accidents_train['Light_Condition_3'])
```

Pedestrian_Crossing_2 is set to 'No Pedestrian Crossing' if there is central refuge or footbridge or subway or no physical crossing within 50 meters

```
accidents_train['Pedestrian_Crossing_2'] = np.where(accidents_train['Pedestrian_Crossing-Physical_Facilities']
                                        == 'Central refuge' ,
               'No Pedestrian Crossing', accidents_train['Pedestrian_Crossing-Physical_Facilities'])
accidents_train['Pedestrian_Crossing_2'] = np.where(accidents_train['Pedestrian_Crossing_2']
                                        == 'Footbridge or subway' ,
               'No Pedestrian Crossing', accidents_train['Pedestrian_Crossing_2'])
accidents_train['Pedestrian_Crossing_2'] = np.where(accidents_train['Pedestrian_Crossing_2']
                                        == 'No physical crossing within 50 meters' ,
               'No Pedestrian Crossing', accidents_train['Pedestrian_Crossing_2'])
```

We use a grid of 15x10 cells to represent both latitude and longitude. They are divided into equal sized bins using percentiles based on the distribution of the data. We have 150 cells in this grid, which means 150 new attributes with values equal to 0 or 1.

```
accidents_train['B_Latitude'] = np.array(pd.qcut(accidents_train['Latitude'], q=15, precision=1).astype(str))
accidents_train['B_Longitude'] = np.array(pd.qcut(accidents_train['Longitude'], q=10, precision=1).astype(str))
accidents_train['M_LAT_LON'] = accidents_train[['B_Latitude', 'B_Longitude']].apply(lambda x: '-'.join(x), axis=1)
accidents_train.drop(['B_Latitude', 'B_Longitude'], axis=1, inplace=True)
```

The DateTime data is breakdown into year, month, day, and hour. With these new attributes, we can have more information about when the accidents occur the most. (which hour? Which day of week? Which month? Which year?)

```
accidents_train['DateTime'] = pd.to_datetime(accidents_train['Date']+" "
                              +accidents_train['Time'], format='%d/%m/%Y %H:%M')
accidents_train['year'] = accidents_train['DateTime'].dt.year
accidents_train['month'] = accidents_train['DateTime'].dt.month
accidents_train['day'] = accidents_train['DateTime'].dt.day
accidents_train['hour'] = accidents_train['DateTime'].dt.hour
accidents_train.drop(['Date', 'Time', 'Year', 'DateTime'], axis=1, inplace=True)
accidents_train['hour'] = accidents_train['hour'].astype(np.float32)
```

Dealing With Missing Values

First, we need to calculate the total missing values for each attribute and their percentage. Methods for dealing with missing value is decided based on this result.

```
null_in_column = []
for column in accidents_train.columns:
    null_in_column.append((column, accidents_train[column].isnull().sum(),
                           str(accidents_train[column].isnull().sum()*100/len(accidents_train))+"%"))
pd.DataFrame(null_in_column, columns=['Column Name', 'Total Missing', 'Percentage Missing'])
```

No	Column Name	Total Missing	Percentage Missing
0	Unnamed: 0	0	0.0%
1	Accident_Index	0	0.0%
2	Location_Easting_OSGR	0	0.0%
3	Location_Northing_OSGR	0	0.0%
4	Longitude	0	0.0%
5	Latitude	0	0.0%
6	Police_Force	0	0.0%
7	Accident_Severity	0	0.0%
8	Number_of_Vehicles	0	0.0%
9	Number_of_Casualties	0	0.0%
10	Date	0	0.0%
11	Day_of_Week	0	0.0%
12	Time	13	0.0027975218260500928%

13	Local_Authority_(District)	0	0.0%
14	Local_Authority_(Highway)	0	0.0%
15	1st_Road_Class	0	0.0%
16	1st_Road_Number	0	0.0%
17	Road_Type	0	0.0%
18	Speed_limit	0	0.0%
19	Junction_Detail	464697	100.0%
20	Junction_Control	178610	38.43579795006200%
21	2nd_Road_Class	0	0.0%
22	2nd_Road_Number	0	0.0%
23	Pedestrian_Crossing-Human_Control	0	0.0%
24	Pedestrian_Crossing-Physical_Facilities	0	0.0%
25	Light_Conditions	0	0.0%
26	Weather_Conditions	0	0.0%
27	Road_Surface_Conditions	755	0.16247145989752462%
28	Special_Conditions_at_Site	2	0.0004303879732384758%
29	Carriageway_Hazards	3	0.0006455819598577137%
30	Urban_or_Rural_Area	0	0.0%
31	Did_Police_Officer_Attend_Scene_of_Accident	2	0.0004303879732384758%
32	LSOA_of_Accident_Location	28718	6.179940907731270%
33	Year	0	0.0%
34	cluster_accidents	0	0.0%
35	fatal_accidents	0	0.0%
36	isFatal	0	0.0%

With 38.44% of missing values, we decided to fill missing values of *'Junction_Control'* with 'Missing Info' string

```
## Set Missing Values to category according to bivariate analysis
accidents_train["Junction_Control"].fillna("Missing Info", inplace = True)
```

Feature Elimination

These following columns are dropped due to some reasons explained in the comments. Feature elimination makes the model more accurate and avoids overfitting.

```
# Drop null columns
accidents_train = accidents_train.drop(['Junction_Detail'], axis=1)
# Drop string variables with a lot of different values
accidents_train.drop(['LSOA_of_Accident_Location', 'Local_Authority_(District)',
                      'Local_Authority_(Highway)', '1st_Road_Number',
                      '2nd_Road_Number'], axis=1, inplace=True)
# Drop string with concentration in just one category
accidents_train.drop(['Special_Conditions_at_Site', 'Carriageway_Hazards',
                      'Pedestrian_Crossing-Human_Control'], axis=1, inplace=True)
# Drop null rows
accidents_train.dropna(axis=0, subset=['Road_Surface_Conditions', 'Time'], inplace=True)
# Drop post-accident rows
accidents_train = accidents_train.drop(['Number_of_Casualties', 'Number_of_Vehicles',
                                        'Did_Police_Officer_Attend_Scene_of_Accident'], axis=1)
# Drop other (Index, Target Derived variable and Index of source file)
accidents_train = accidents_train.drop(['Accident_Index'], axis=1)
accidents_train = accidents_train.drop(['isFatal'], axis=1)
accidents_train = accidents_train.drop(['Unnamed: 0'], axis=1)
```

Preparing train/test dataset

Categorical columns need to be converted into string type and encoded using one-hot encodings

```
new_categorial_analysis = ['Day_of_Week','cluster_accidents','fatal_accidents','month',
                           'hour','Road_Type', 'Light_Conditions','Weather_Conditions',
                           'Road_Surface_Conditions','Urban_or_Rural_Area','Light_Condition_2',
                           'Light_Condition_3','Pedestrian_Crossing_2','Junction_Control',
                           'Pedestrian_Crossing-Physical_Facilities','M_LAT_LON']

for col_cat in new_categorial_analysis: accidents_train[col_cat] = accidents_train[col_cat].astype(str)

accidents_train = pd.get_dummies(accidents_train, columns = new_categorial_analysis)
```

The dataset is split into training data and test data with a ratio of 0.8:0.2

```
dataset_y = accidents_train[ 'Accident_Severity' ]
dataset_x = accidents_train.drop([ 'Accident_Severity' ], axis= 1 )
X_train, X_test, y_train, y_test = train_test_split(dataset_x, dataset_y,
                                    test_size= 0.20 , random_state= 42 )
```

Feature Importance

The Random Forest Classifier was used to determine which were the most important features in the model.

The first step was to undersample the data in order to obtain a dataset with equal number of samples for each type of accident.

```
fatal_accidents = accidents_train[accidents_train['Accident_Severity']== 1 ]
serious_accidents = accidents_train[accidents_train['Accident_Severity']== 2 ].sample(len(fatal_accidents), random_state= 42)
slight_accidents = accidents_train[accidents_train['Accident_Severity']== 3 ].sample(len(fatal_accidents), random_state= 42)
ds_undersample = pd.concat([slight_accidents,fatal_accidents,serious_accidents],axis= 0 ).sample(frac=1)
```

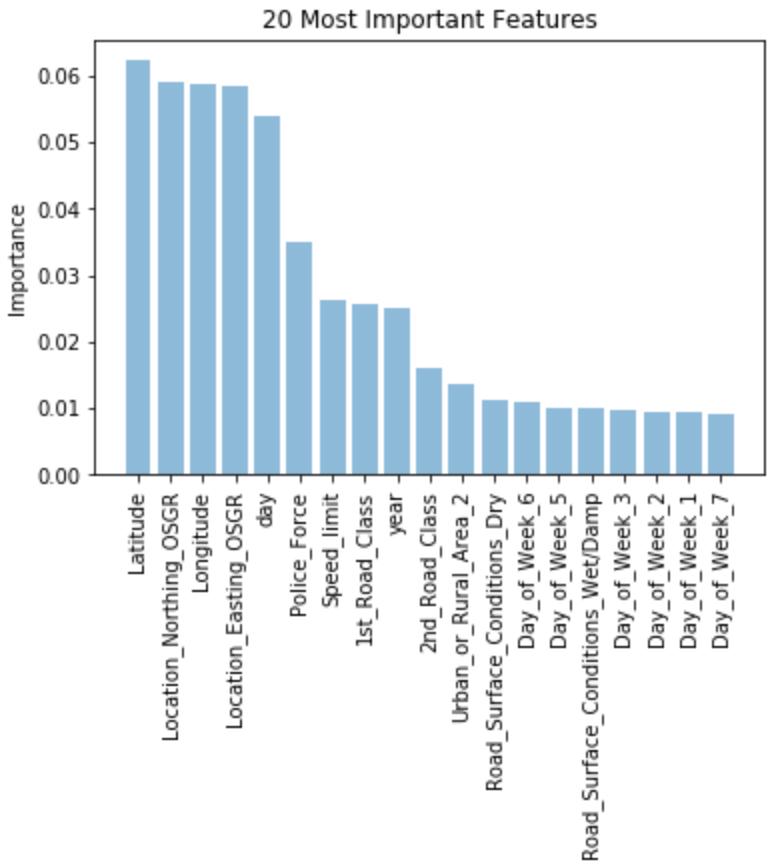
This dataset was then split between a train and a test set.

```
dataset_y = ds_undersample[ 'Accident_Severity' ]
dataset_x = ds_undersample.drop([ 'Accident_Severity' ], axis= 1 )
X_train, X_test, y_train, y_test = train_test_split(dataset_x, dataset_y, test_size= 0.20 , random_state= 42 )
```

After fitting the training dataset to a Random Forest Classifier (using its default parameters), it was possible to obtain the most important features through the *feature_importances_* method.

```
model = RandomForestClassifier(random_state = 42)
model.fit(X_train, y_train)

importances = model.feature_importances_
ds_X = X_train
data = ds_X.columns
t = pd.DataFrame(data, columns = ['Features'])
t['Importances'] = importances
rank_attr = t.sort_values(by=['Importances'], ascending=False)
```

The following chart shows the 20 most important features as determined by the Random Forest Classifier.



It can be observed in the chart that several of the most important features are related to the location of the accident and the day on which it occurred. Other important features also include the speed limit of the road, roads in rural areas and whether there was a dry surface on the area of the accident.

It should also be noted that, overall, the importance values obtained are small, even for the most important features, which suggests that in most cases an accident may be caused by more than one variable.

However, calculating the correlation in the undersampled dataset between all the features and the Accident Severity provides additional information.

```
corr = ds_undersample.corr()
corr["Accident_Severity"].sort_values(ascending=False)
```

The table below shows the 20 features with the highest correlation values. Although the highest values still indicate a weak correlation to the Accident Severity, it can be observed that some of the features identified by the Random Forest as the most important also appear on this table.

Features	Correlation
Speed_limit	-0.2621
Urban_or_Rural_Area_1	0.2423
Urban_or_Rural_Area_2	-0.2423
Junction_Control_Missing Info	-0.2012
2nd_Road_Class	0.1817
Light_Conditions_Darkeness: No street lighting	-0.1714
Light_Condition_3_Darkeness: Null or Low lighting	-0.1694
Junction_Control_Giveway or uncontrolled	0.1540
cluster_accidents_-1	-0.1462
Light_Conditions_Daylight: Street light present	0.1218
Light_Condition_2_Daylight	0.1218
Light_Condition_3_Daylight: Street light present	0.1218
Light_Condition_2_Darkness	-0.1218
Road_Type_Roundabout	0.1165
cluster_accidents_0	0.1147
fatal_accidents_1	0.1042
1st_Road_Class	0.1009
Police_Force	-0.0962
fatal_accidents_-1	-0.0935

For example, speed limit not only was identified as an important feature, but it also has the highest correlation value; since the value is negative, this suggests that roads with higher speed limits increase the severity of the accidents.

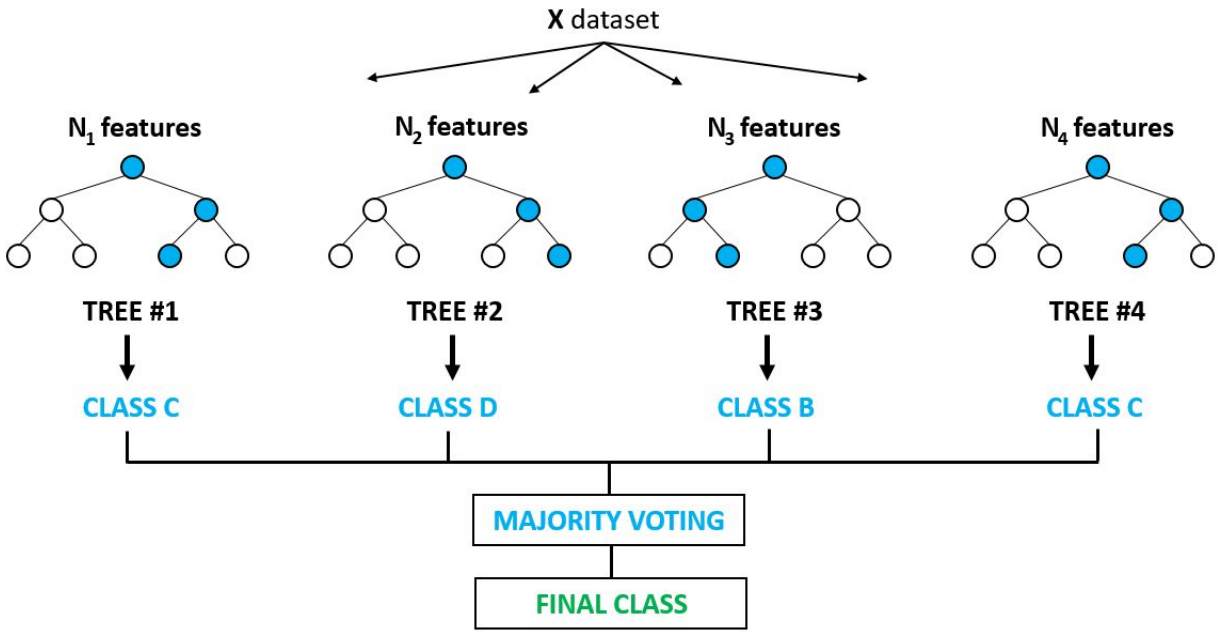
V. Modelling

Random Forest

Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest (Breiman, 2001).

For every tree in the Random Forest, each node is split by using a random selection of features available in the data set. Then, after a large number of trees is generated, each one of them casts its vote indicating which should be the class selected. The class with the highest amount of votes (or most popular class) is selected for the classification.

The image below shows an example of how a Random Forest works. In this case, 4 different trees were generated and the final class selected would be “CLASS C”, since it obtained the highest amount of votes (2) among the 4 trees.



Global Software Support (2018).Retrieved from https://www.globalsoftwaresupport.com/random-forest-classifier/

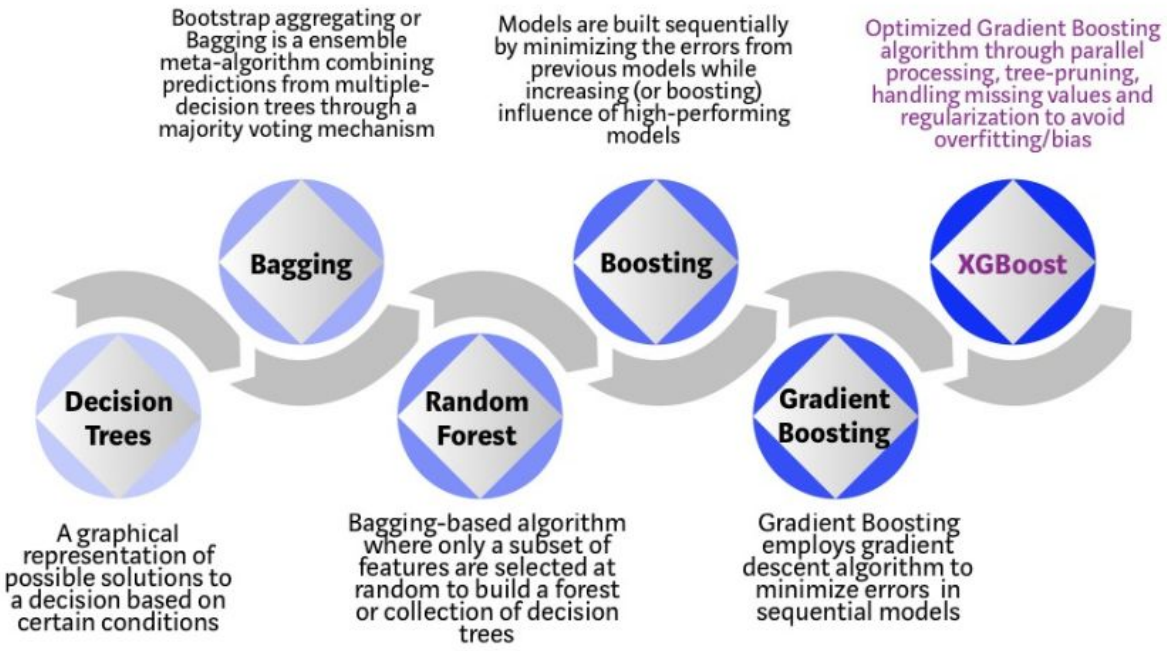
Some of the most important parameters that have to be considered when working with a Random Forest Classifier are the following:

- **Number of estimators:** Number of trees in the forest. Each tree will cast a vote for the most popular class in order to decide the final class assigned to the sample.

- **Maximum Depth:** Maximum depth of the trees generated.
- **Minimum samples to split:** Minimum number of samples required to split an internal node.
- **Maximum number of features:** The number of features to consider when looking for the best split.

XGBoost

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. Below shows the evolution of tree-based algorithms:



Evolution of XGBoost Algorithm from Decision Trees

XGBoost and Gradient Boosting Machines (GBMs) are both ensemble tree methods that apply the principle of boosting weak learners using the gradient descent architecture. However, XGBoost improves upon the base GBM framework through systems optimization and algorithmic enhancements.

System Optimization:

- Parallelization: XGBoost approaches the process of sequential tree building using parallelized implementation. This is possible due to the interchangeable nature of loops used for building base learners; the outer loop that enumerates the leaf nodes of a tree, and the second inner loop that calculates the features. This nesting of loops limits parallelization because without completing the inner loop (more computationally

demanding of the two), the outer loop cannot be started. Therefore, to improve run time, the order of loops is interchanged using initialization through a global scan of all instances and sorting using parallel threads. This switch improves algorithmic performance by offsetting any parallelization overheads in computation.

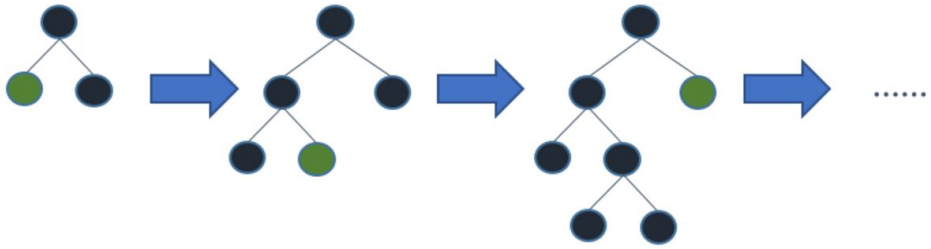
- Tree Pruning: The stopping criterion for tree splitting within GBM framework is greedy in nature and depends on the negative loss criterion at the point of split. XGBoost uses 'max_depth' parameter as specified instead of criterion first, and starts pruning trees backward. This 'depth-first' approach improves computational performance significantly.
- Hardware Optimization: This algorithm has been designed to make efficient use of hardware resources. This is accomplished by cache awareness by allocating internal buffers in each thread to store gradient statistics. Further enhancements such as 'out-of-core' computing optimize available disk space while handling big data-frames that do not fit into memory.

Algorithmic Enhancements:

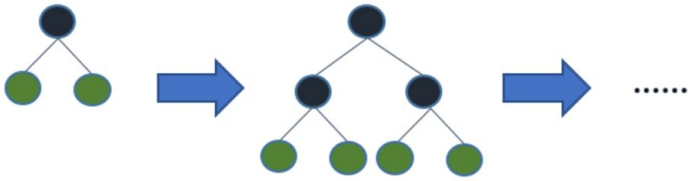
- **Regularization:** It penalizes more complex models through both LASSO (L1) and Ridge (L2) regularization to prevent overfitting.
- **Sparsity Awareness:** XGBoost naturally admits sparse features for inputs by automatically 'learning' best missing value depending on training loss and handles different types of sparsity patterns in the data more efficiently.
- **Weighted Quantile Sketch:** XGBoost employs the distributed weighted Quantile Sketch algorithm to effectively find the optimal split points among weighted datasets.
- **Cross-validation:** The algorithm comes with built-in cross-validation method at each iteration, taking away the need to explicitly program this search and to specify the exact number of boosting iterations required in a single run.

LightGBM

LightGBM is a new gradient boosting framework designed to be distributed, highly efficient and scalable. It supports many different tree based learning algorithms including GBDT, GBRT, GBM, and MART. The difference is Light GBM grows tree vertically (leaf-wise) while other algorithms grow trees horizontally (level-wise). The figures below explain these tree growing concepts. Leaf-wise algorithms can reduce more loss than level-wise algorithms. LightGBM can handle a large amount of data with higher speed and lower memory. It also supports GPU learning, which is one of the reasons why it is becoming widely used for data science application development.



Leaf-wise tree growth



Level-wise tree growth

The complicated part in implementing LightGBM is parameter tuning. There are more than 100 parameters, and the following ones are important:

- **boosting**: the type of base algorithm (default=gdbt)
    - gbdt: traditional Gradient Boosting Decision Tree
    - rf: random forest
    - dart: Dropouts meet Multiple Additive Regression Trees
    - goss: Gradient-based One-Side Sampling
- **max_depth**: describes the maximum depth of trees, used to handle model overfitting.
- **num_leaves:** number of leaves in full tree, should be less than or equal to 2^(max_depth) to avoid overfitting (default=31))
- **learning_rate**: Boosting learning rate controls the magnitude of changes in the estimates after each round (default=0.1)
- **n_estimators**: Number of boosting iterations (default=100)
- **early_stopping_rounds:** stop boosting if one metric of one validation data doesn't improve in last early_stopping_round rounds.

Multi Layer Perceptron (MLP)

MLP is one of the non-linear algorithms that belongs to the family of neural networks. It is used for both classification and regression complex problems. It consists of three types of layers and an activation function which is generally a logistic function.

- Visible Layer: Contains the input vectors and an intercept

- Hidden Layers: Contains weights that will transform the input vectors and an intercept
- Output Layers: Transforms the hidden layers according to the activation function to get the probabilities or prediction for each target variable.

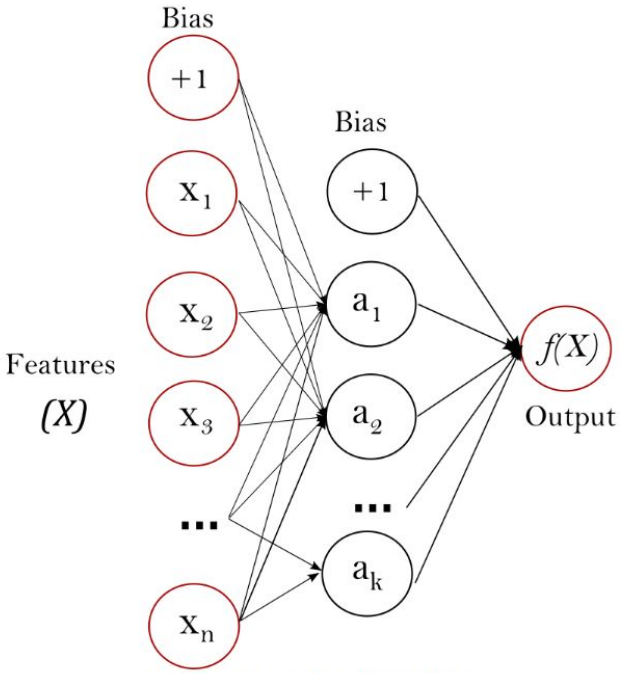


Figure 1 : One hidden layer MLP.

Scikitlearn (2019).Retrieved from https://scikit-learn.org/stable/modules/neural_networks_supervised.html

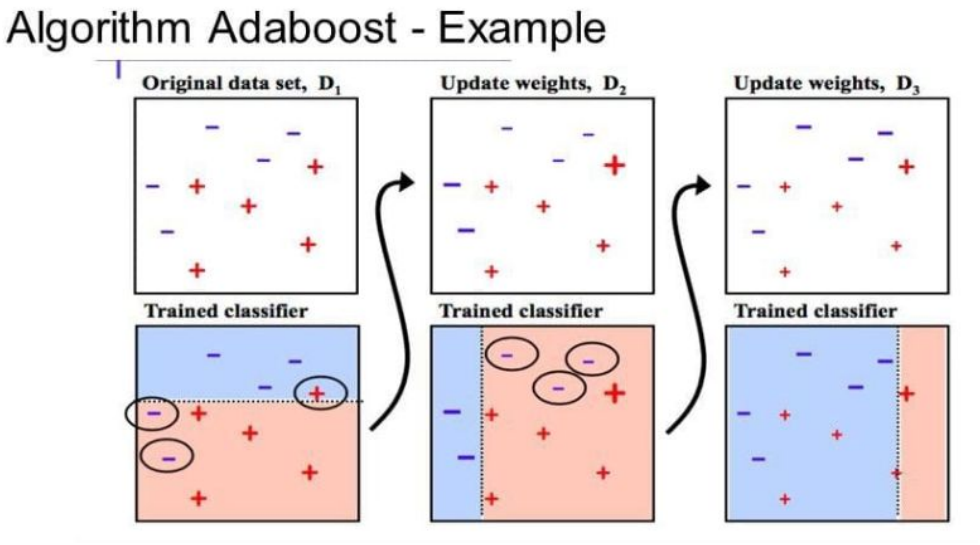
Its complexity is determined by all the parameters needed to tune. The following is a list of the most important:

- **Hidden Layers (*hidden_layer_sizes)*:** This will define the amount of information the neural network is going to store. It can vary depending on the distribution and complexity of the dataset: n hidden layers with different neurons per layer. Since the data is more simple, only one hidden layer with different number of neurons will be tested.
- **Activation function (activation):** Represents the function to activate each layer from its previous weights. Logistic or Tan are often used.
- **Learning rate (learning_rate):** It will influence the time and precision for the stochastic gradient descent.
- **Regularization parameter (alpha):** Also called L2, it allows to control the weights of the input vectors in order to avoid overfitting
- **momentum(momentum):** It allows us to increase the step for learning so that the learning process doesn't get stuck in a local solution.

Ada Boost

AdaBoost belongs to the family of boosting algorithms and it was actually one of the first ones. Its power stems from its capability to create strong classifiers ensembling a set of weak classifiers. The process to train the model is the following:

- First the algorithm choose one weak classifier (generally decision trees) and train in a sample (with replacement) of the dataset in which the weights are the same for each row.
- Then, the misclassification error is calculated in the test set which belongs from the sample
- The weights of every row are updated (the misclassified rows increase their weights; otherwise, their weights decrease.)
- The next classifiers will repeat the process with the new weights.



easyAI(2019).Retrieved from https://easyai.tech/en/ai-definition/adaboost/

The following parameters are important for further tuning:

- **Classifier (base_estimator):** It can be any weak classifier that will be improved by the algorithm technique. By default, it's decision trees with maxDepth = 1.
- **Algorithm:** It represents the boosting algorithm either SAMME (discrete) or SAMME.R (real, it also gets faster results)
- **Number of estimators (n_estimators)**: Number of classifiers to train until you get the ideal model.
- **Learning Rate (learning_rate):** It will influence the weights of weak classifiers.

Experiments

After preparing the data, 5 experiments were carried out in order to test each one of the models described in the section below. Every new experiment had the intention to improve the result obtained in the previous one. Testing the models using the same 5 experiments for each one helps to determine which one makes the best prediction for the accident severity without having bias from different data processing or the handling of the models' parameters.

The output monitored for each experiment was the resulting ROC curve, which is a performance measurement for classification that indicates how much model is capable of distinguishing between classes (Narkhede, 2018).The higher this value, the better the model is at predicting the type of accident according to the features used.

The 5 experiments are described below. Although the experiments were carried out for all the models, only the code for the tests performed on the Random Forest Classifier model are shown to provide an example of the implementation. The results for each experiment on every model can be observed in the following section of this document.

1. Unbalanced dataset

   The original dataset contains unbalanced data, which can be observed by counting the number of samples for each type of Accident Severity.

```
accidents_train['Accident_Severity'].value_counts()
```

```
3    391942
2     66689
1      5298
Name: Accident_Severity, dtype: int64
```

   So, for the first experiment, the models (with their default parameters), were used on this data set.

```
##Test 1: Normal

dataset_y = accidents_train[ 'Accident_Severity' ]
dataset_x = accidents_train.drop([ 'Accident_Severity' ], axis= 1 )
X_train, X_test, y_train, y_test = train_test_split(dataset_x, dataset_y, test_size= 0.20 , random_state= 42 )

print('Random Forest - Normal')

model = RandomForestClassifier(random_state = 42)
model.fit(X_train, y_train)

y_probas=model.predict_proba(X_test)
roc = roc_auc_score(np.where(y_test==1 , 1 ,0), y_probas[:,0])
skplt.metrics.plot_roc_curve(y_test, y_probas)
plt.show()
```

2. Undersampling

For the second experiment, an undersampled dataset was generated.

```
##Test 2: UnderSampling

fatal_accidents = accidents_train[accidents_train['Accident_Severity']== 1 ]
serious_accidents = accidents_train[accidents_train['Accident_Severity']== 2 ]
                                    .sample(len(fatal_accidents), random_state= 42)
slight_accidents = accidents_train[accidents_train['Accident_Severity']== 3 ]
                                    .sample(len(fatal_accidents), random_state= 42)
ds_undersample = pd.concat([slight_accidents,fatal_accidents,serious_accidents],axis= 0 )
                                    .sample(frac=1, random_state= 42)
```

The number of samples with serious or slight accidents was reduced to the same amount of existing samples for fatal accidents. Therefore, the new dataset was balanced, and the total number of samples was reduced to 15,894.

```
ds_undersample['Accident_Severity'].value_counts()
```

```
3    5298
2    5298
1    5298
Name: Accident_Severity, dtype: int64
```

Once the undersample was completed, the models were fit to this dataset to evaluate the results.

```
dataset_y = ds_undersample[ 'Accident_Severity' ]
dataset_x = ds_undersample.drop([ 'Accident_Severity' ], axis= 1 )
X_train, X_test, y_train, y_test = train_test_split(dataset_x, dataset_y, test_size= 0.20 , random_state= 42 )

model = RandomForestClassifier(random_state = 42)
model.fit(X_train, y_train)

y_probas=model.predict_proba(X_test)
roc = roc_auc_score(np.where(y_test==1 , 1 ,0), y_probas[:,0])
skplt.metrics.plot_roc_curve(y_test, y_probas)
plt.show()
```

3. Normalization

For the third experiment, the undersampled data was normalized. Data normalization is used to scale the data of an attribute to make it fit in a different range (normally between 0 and 1).

$$Normalize(x_i) = \frac{x_i - min(X)}{max(X) - min(X)}$$

To normalize the data, the MinMaxScaler was used with its default feature range (0, 1).

```
##Test 3: Normalization

scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_normalized , X_test_normalized = scaler.transform(X_train), scaler.transform(X_test)

model.fit(X_train_normalized, y_train)

y_probas=model.predict_proba(X_test_normalized)
roc = roc_auc_score(np.where(y_test==1 , 1 ,0), y_probas[:,0])
skplt.metrics.plot_roc_curve(y_test, y_probas)
plt.show()
```

4. Hyperparameter Tuning

The next experiment involved the modification of each model's hyperparameters to identify which values could make a better job predicting the current dataset. This tuning of hyperparameters was performed using a grid search.

Grid search is an approach to hyperparameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid (Paul, 2018). This means that, for a user-defined set of values for the hyperparameters, the grid search will run the model using all possible combinations in order to identify which one provides the best estimator.

The Scikit-Learn library for Python contains a method that performs this grid search, called *GridSearchCV()*. The following image shows an example of its implementation to tune some of the hyperparameters for the Random Forest Classifier.

```
##Test 4: Hyperparameters tunning

param_grid = {
  'n_estimators' : [500,700,1000,1500],
  'max_depth' : [24,30,35,40],
  'min_samples_split': [2,5]
}

model = RandomForestClassifier(random_state = 42)
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = 3, n_jobs = -1, verbose = 2)

grid_search.fit(X_train, y_train)
best_grid = grid_search.best_estimator_
grid_search.best_estimator_
print(best_grid)
```

The output of the GridSearchCV provides the best values for the model's hyperparameters. An example of its output can be observed in the image below.

```
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed: 10.7min
[Parallel(n_jobs=-1)]: Done  96 out of  96 | elapsed: 28.9min finished
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=30, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

These values are then introduced to the model to evaluate the improvement on the estimations.

```
model = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                        max_depth=30, max_features='auto', max_leaf_nodes=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=1, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, n_estimators=500,
                        n_jobs=None, oob_score=False, random_state=42, verbose=0,
                        warm_start=False)

model.fit(X_train, y_train)
y_probas=model.predict_proba(X_test)
roc = roc_auc_score(np.where(y_test==1 , 1 ,0), y_probas[:,0])
skplt.metrics.plot_roc_curve(y_test, y_probas)
plt.show()
```

5. Feature Selection

For the fifth experiment, the number of features considered for the modeled was modified (reduced) to different values to analyze if better results could be obtained by cancelling the noise that some of the less important features could provide.

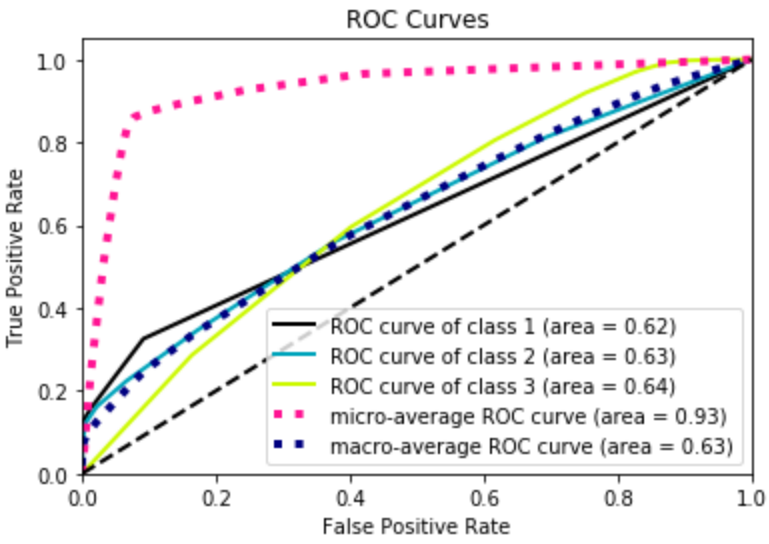
A customized function was generated in order to fit the models into datasets with different numbers of features. The ROC resulting value was then compared with the maximum ROC value obtained so far; if the new value was better, the number of features was stored in a variable called *best_dim*. This value was then used to shorten the features of the dataset and keep only the [best_dim] most important features. The most important features had already been identified during the data preparation, so this did not require additional processing.

```
##Test 5: Feature Importance

def getBestNumberOfFeatures(X_train, y_train, X_test, y_test, model, rank_attr):
    start = [30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,180,190,200,210,220,250,267]
    max_roc = 0
    max_start = 0
    save_Best = pd.DataFrame([[1.000000]*2]*(len(start)),columns = ['dim','roc'])
    for i in range(len(start)):
        model.fit(X_train[rank_attr[0:start[i]]['Features']], y_train)
        y_probas=model.predict_proba(X_test[rank_attr[0:start[i]]['Features']])
        roc = roc_auc_score(np.where(y_test==1 , 1 ,0), y_probas[:,0])
        save_Best['dim'][i] = start[i]
        save_Best['roc'][i] = roc
        if roc>max_roc:
            max_roc = roc
            max_start = start[i]
    plt.plot(save_Best['dim'].astype(int), save_Best['roc'])
    return  max_start

model = model

best_dim = getBestNumberOfFeatures(X_train, y_train, X_test, y_test, model, rank_attr)
print("Best number of dimensions: " + str(best_dim))

new_dims = rank_attr[0:best_dim]['Features']
model.fit(X_train[new_dims], y_train)
```

Experiment Results

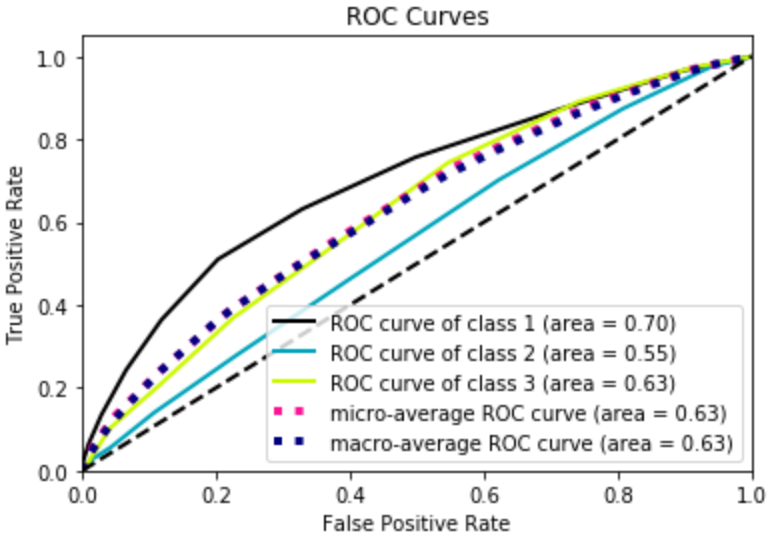
Random Forest

- Unbalanced dataset: With an unbalanced dataset, a value of 0.62 is obtained for the ROC curve of class 1..



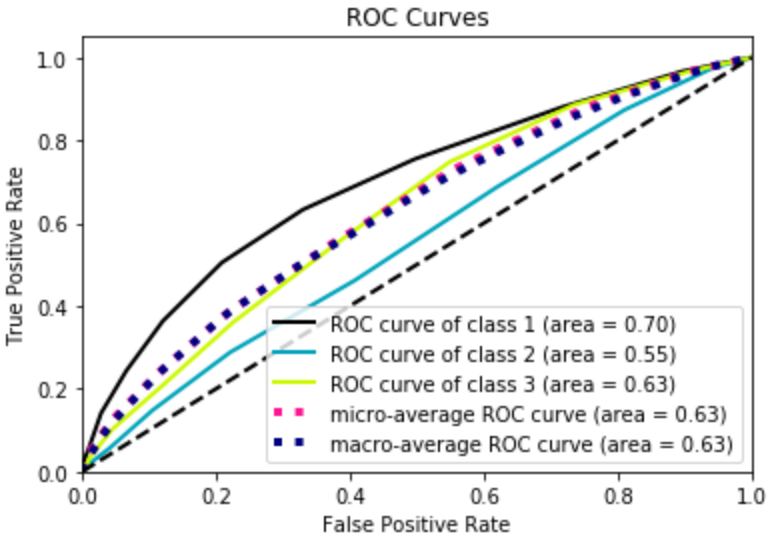
ROC curve after unbalanced data - Random Forest

- Undersampling: After undersampling the data, a significant upgrade can be observed in the ROC value, improving by 0.08.



ROC curve after undersampling - Random Forest

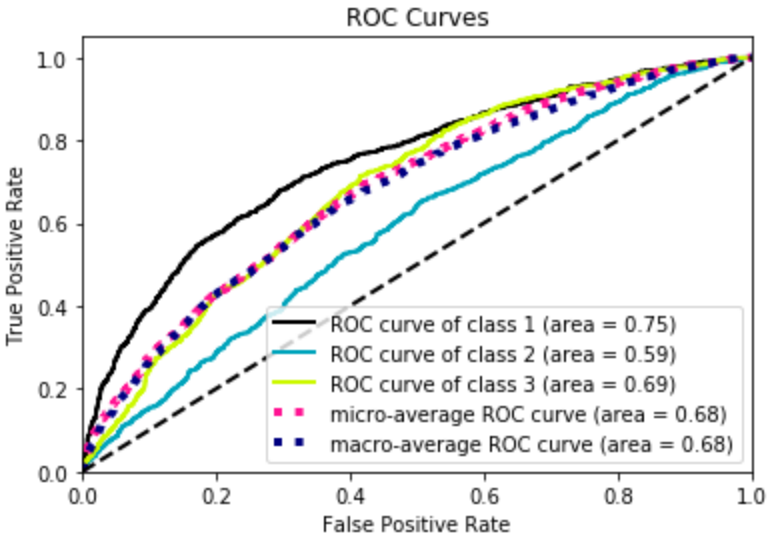
- Normalization: Normalizing the data does not represent an improvement over the previous experiment.



ROC curve after normalization - Random Forest

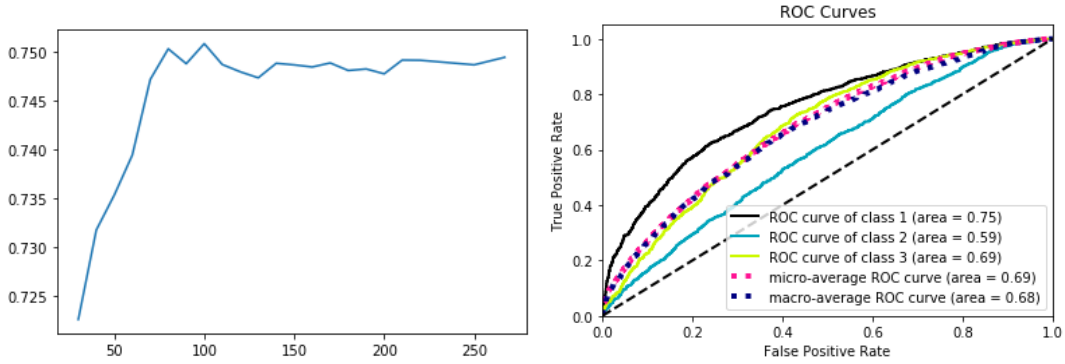
- Hyperparameter tuning: Setting up the correct parameter values for the Random Forest Classifier improves the ROC value, which is now of 0.75.

```
model = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                               max_depth=30, max_features='auto', max_leaf_nodes=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, n_estimators=500,
                               n_jobs=None, oob_score=False, random_state=42, verbose=0,
                               warm_start=False)
```



ROC curve after hyper tuning - Random Forest

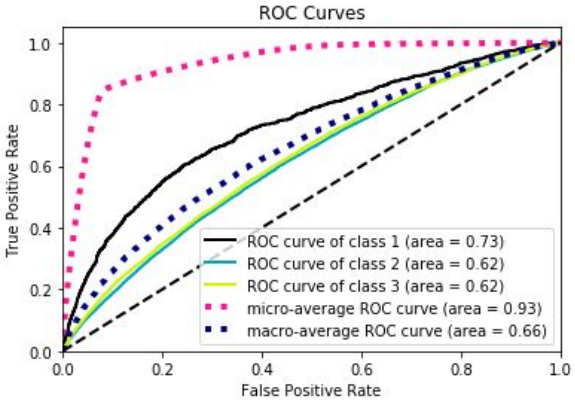
- Feature Selection: Modeling with 90 features improves the ROC value slightly. Taking a closer look at the ROC values outside of the plots, it could be observed that the ROC value goes from 0.7482 in the previous experiment up to 0.7508 with the feature selection.



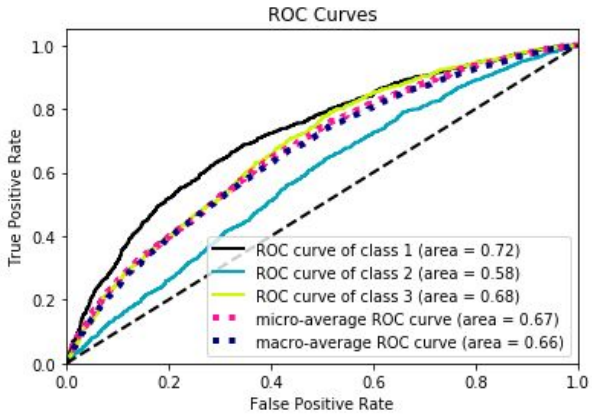
ROC by number of features used (left) and ROC curve after feature selection (right) - Random Forest

XGBoost

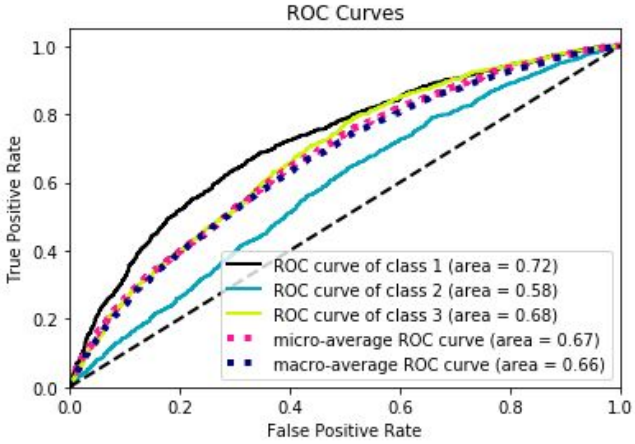
- Unbalanced dataset: We obtained a base ROC value of 0.73.



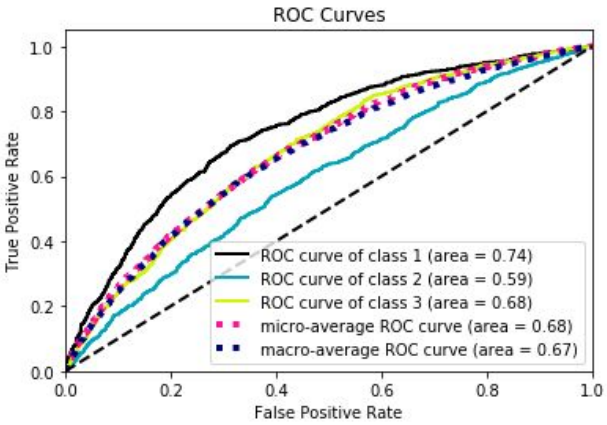
- Undersampling. ROC value actually dropped to 0.72 with undersampling.



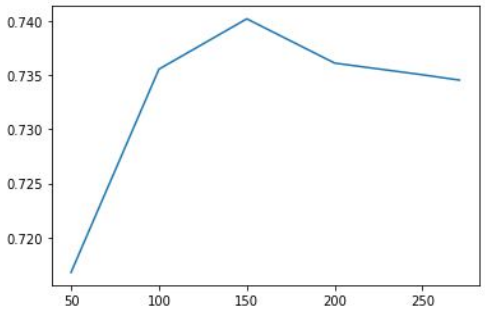
- Normalization

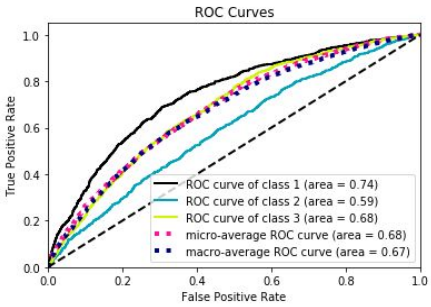


- Hyperparameter Tuning. With hyperparameter tuning, we see an increase from 0.73 to 0.74. However it does not improve much.



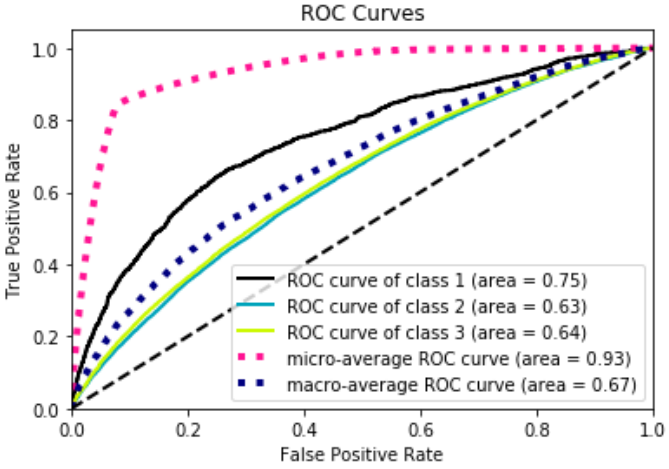
- Feature Selection. We can see that using more features increases the ROC up to around 150 features. The ROC value reached 0.740 in its peak.



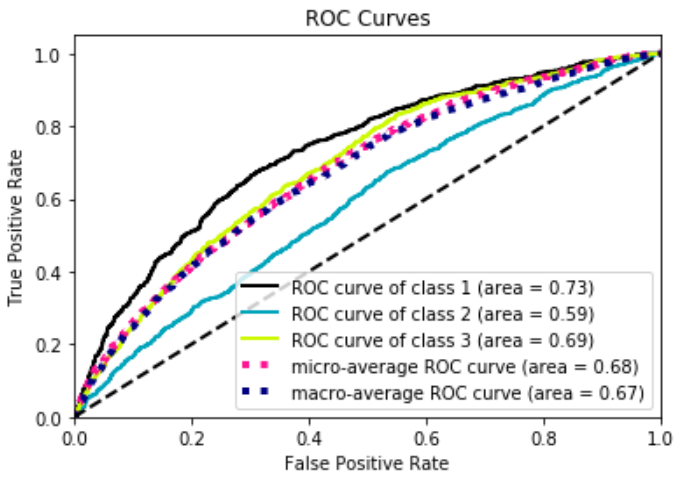


LightGBM

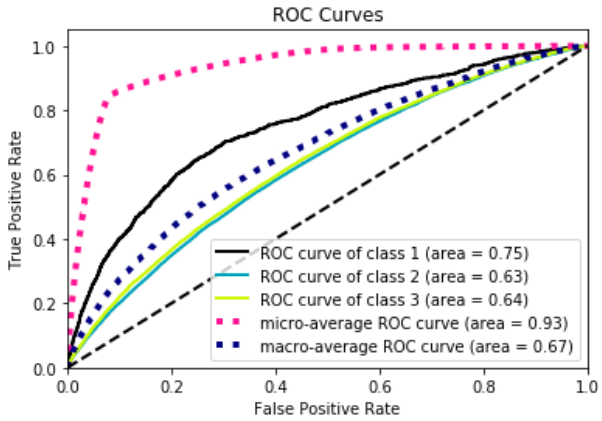
- **Unbalanced dataset:** We first start with unbalanced data and LightGBM with all parameters are set to default. The ROC value for class 1 is 0.75.



- **Undersampling:** Next, we try to undersample the dataset before training the model. The ROC values for all classes drop, for example the ROC value for class 1 drops to 0.73 (it was 0.75 in the previous part). Hence, undersampling in this example decreases the model performance



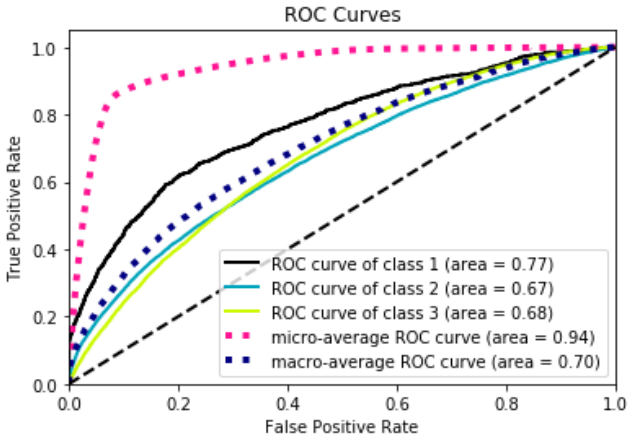
- **Normalization:** With normalization, all the ROC values remain the same. Thus, normalization does not add any value in this case



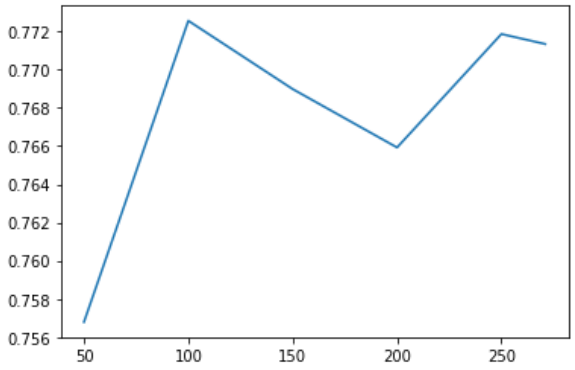
- **Hyperparameter tuning:** Hyperparameter tuning gives us the best hyperparameters to train the model with the highest performance. The hyperparameter values are set as follow:

```
model = lgb.LGBMClassifier(boosting_type='gbdt',
                           colsample_bytree=0.6,
                           importance_type='split', learning_rate=0.01,
                           max_depth=40,
                           min_child_samples=20, min_child_weight=0.001,
                           n_estimators=2000000,
                           n_jobs=8,
                           num_leaves=1000,
                           reg_alpha=0.6,
                           reg_lambda=0.6,
                           subsample=0.8, subsample_freq=10
)

model.fit(X_train, y_train, eval_metric='multi_logloss',
          eval_set=[(X_test, y_test)],
          early_stopping_rounds=50)
```

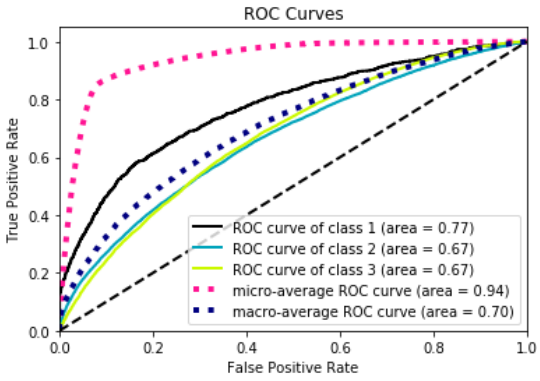
With tuned hyperparameters, the model performance increases with the ROC value for class 1 goes up by 0.02.



- **Feature Selection:** The optimal number of features is 100 and the ROC value for class 1 reaches 0.772 in this case



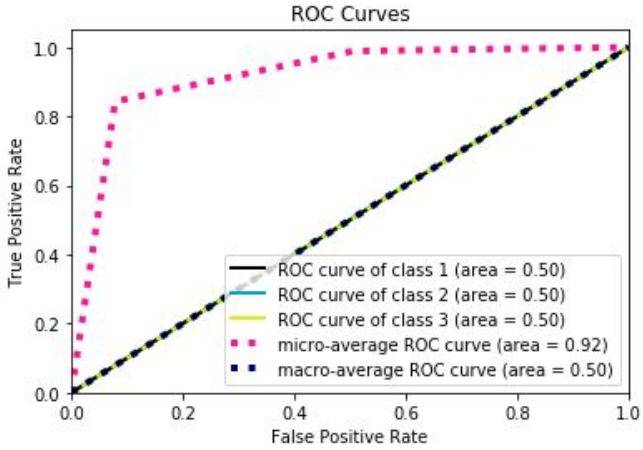
ROC by number of features used



ROC curve after feature selection - LGBM

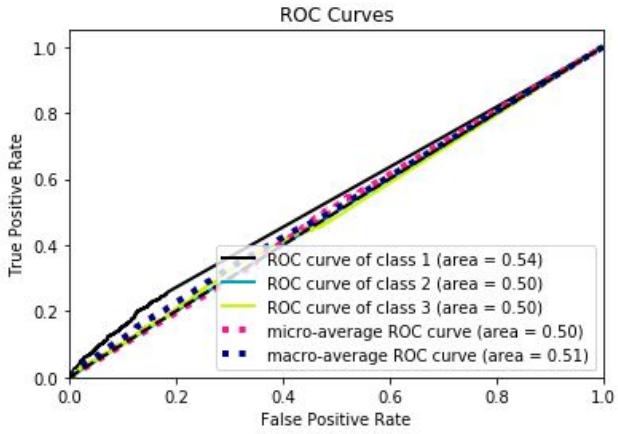
MLP

- Unbalanced dataset: The test with the unbalanced dataset really gives bad results (ROC of 0.50 means that it's almost a random model or it's classifying everything to one class)



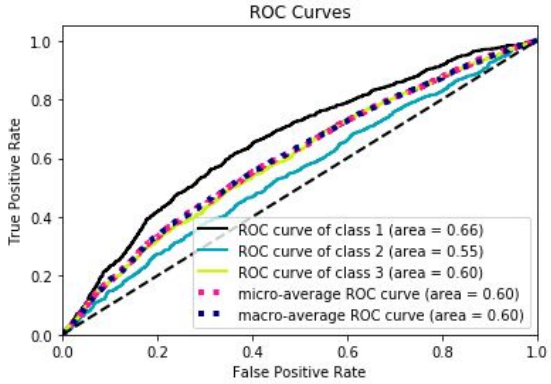
ROC curve after unbalanced data -MLP

- Undersampling: The test with balanced data (33% for each class) yields to better results, yet it's still worse than the other models.



ROC curve after undersampling - MLP

- Normalization: The normalization process rapidly increases the performance of the model in almost 0.12 of ROC. According to the documentation, it is a basic step for this algorithm.

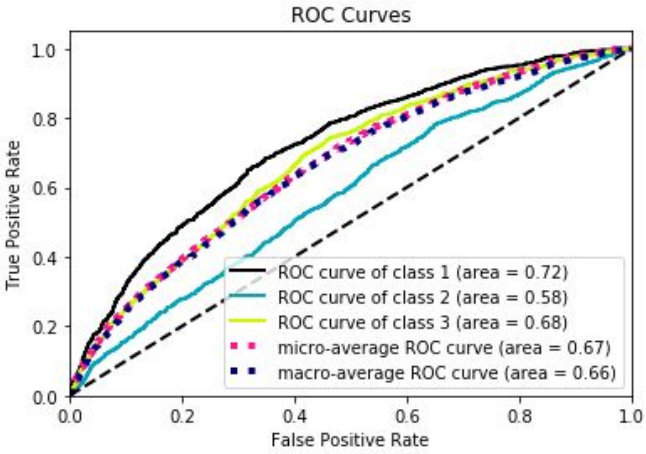


ROC curve after normalization - MLP

- Hyperparameter tuning: The right choice of parameters is also important for neural networks increasing in 0.6 points the roc.

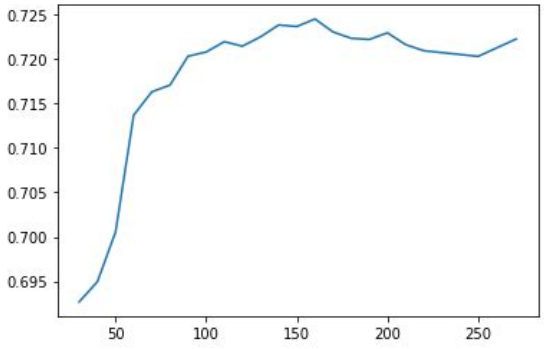
```
model = MLPClassifier(activation='logistic', alpha=0.05, batch_size='auto', beta_1=0.9,
                      beta_2=0.999, early_stopping=False, epsilon=1e-08,
                      hidden_layer_sizes=(30, 30), learning_rate='adaptive',
                      learning_rate_init=0.001, max_fun=15000, max_iter=200,
                      momentum=0.7, n_iter_no_change=10, nesterovs_momentum=True,
                      power_t=0.5, random_state=42, shuffle=True, solver='adam',
                      tol=0.0001, validation_fraction=0.1, verbose=False,
                      warm_start=False)
```

Best combination of parameters for MLP

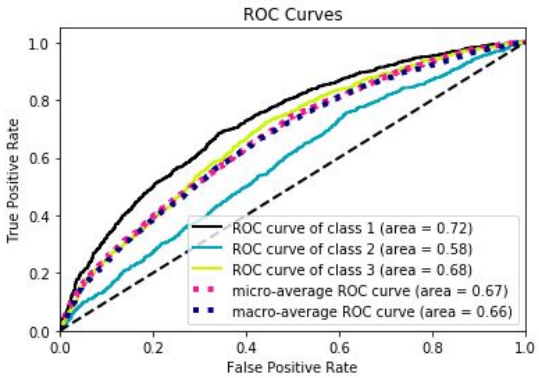


ROC curve after hyper tuning - MLP

- Feature Selection: The less number of dimensions improves slightly the performance (optimum of 160 features from 271). ROC of 0.72



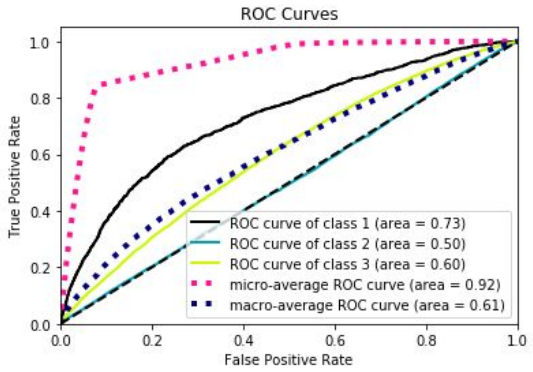
ROC by number of features used



ROC curve after feature selection - MLP

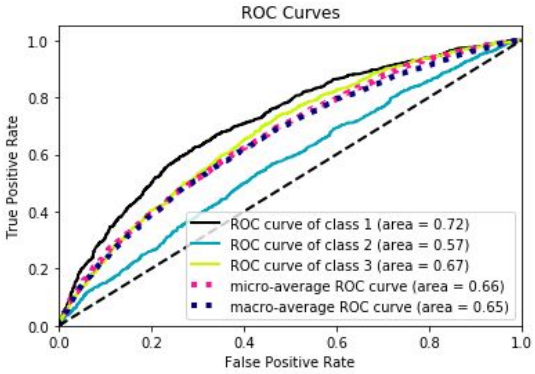
AdaBoost

- Unbalanced dataset: The test with the unbalanced dataset provides good results from the beginning



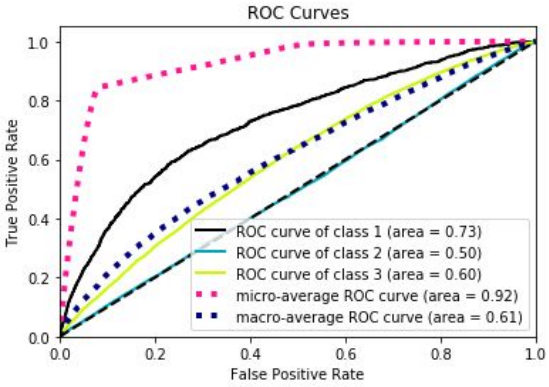
ROC curve after unbalanced data - AdaBoost

- Undersampling: The test with balanced data (33% for each class) slightly decreases the performance.



ROC curve after undersampling - AdaBoost

- Normalization: The normalization doesn't affect the result of the execution

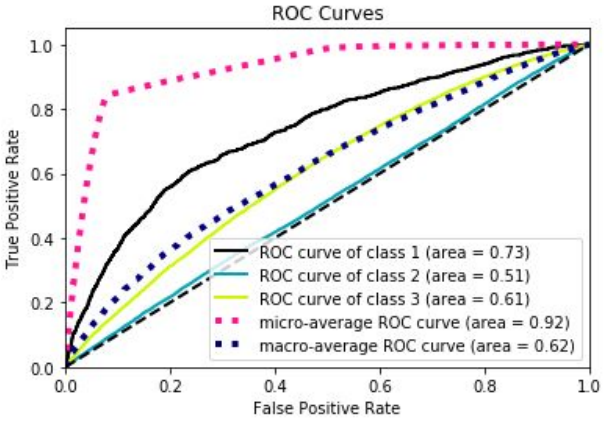


ROC curve after normalization - AdaBoost

- Hyperparameter tuning: The hyper tuning doesn't improve significantly the performance of the model

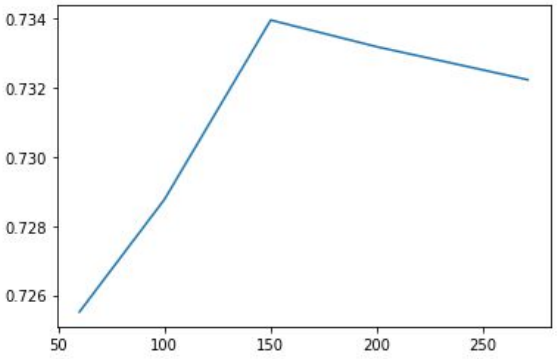
```
model = AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=0.5,
                           n_estimators=600, random_state=42)
```

Best combination of parameters for AdaBoost

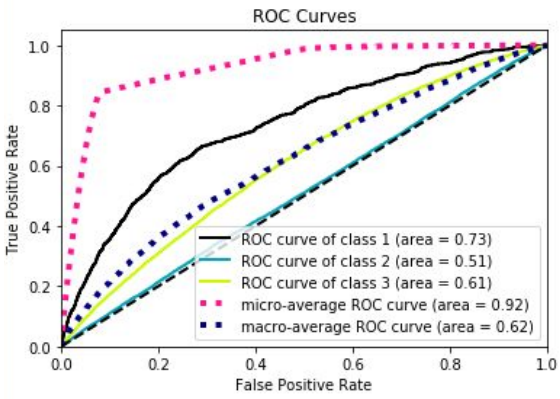


ROC curve after hyper tuning - Adaboost

- Feature Selection: The less number of dimensions increases minimally the result. 150 features results to be the optimal.



ROC by number of features used



ROC curve after feature selection - Adaboost

Evaluation

The previous experiments depicted an extensive iterative process to get the best model which can fit the data and let us predict several accidents from the data source. these experiments, we have obtained two important metrics in order to assess which model should be taken into account: Accuracy and Roc

Accuracy

The accuracy is important to realize the percentage of cases the model is predicting good or not. In this case, the concern is if it is going to be a fatal accident or not, so the formula is the following:

$$Accuracy = \frac{(TP_{(fatal-accident=1)} + TN_{(fatal-accident=0)})}{Total\ Samples}$$

The results according to accuracy shows a great variation depending on the model and the strategy. Although some of the results show a big accuracy, it can lead us to biased results. If we deep into the results, the majority of those classifications are in the bigger category (more than 50% of the samples). The next table shows the results of accuracy:

	RandomForest	XGBoost	LightGBM	MLP	Ada Boost
It1: Unbalanced Data	99%	99%	99%	99%	99%
It2: Undersampling	67%	69%	69%	67%	69%
It3: Normalization	55%	69%	99%	65%	99%
It4: Hyper tuning	71%	70%	99%	68%	99%
It5: Feature Selection	71%	70%	99%	68%	99%

ROC curve

As a consequence, we have supported our choice in a more robust metric: ROC curves. The curve roc represents the area under the curve that is generated by plotting the true positive rate on axis-y (TPR) against the false positive rate on axis-x (FPR)

- TPR = Percentage of correct fatal accidents predictions/total of correct predictions

$$TPR = \frac{TP_{(fatal-accident=1)}}{TP_{(fatal-accident=1)} + FN_{(fatal-accident=0)}}$$

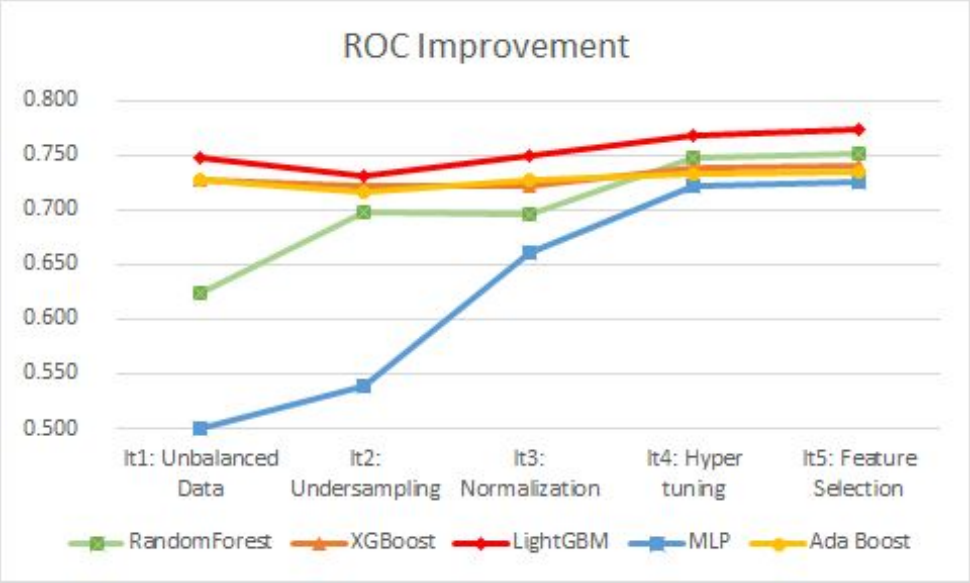
- FPR = Percentage of incorrect fatal accidents predictions/total of incorrect predictions

$$FPR = \frac{FP_{(fatal-accident=1)}}{FP_{(fatal-accident=1)} + TN_{(fatal-accident=0)}}$$

The ROC will let us know the degree of separability between the predictions of the two categories. While the ROC is closer to 1, the model is better. Looking at the results of the next table we can identify how the ROC measures have improved in every iteration which was the purpose of these experiments:

	RandomForest	XGBoost	LightGBM	MLP	Ada Boost
It1: Unbalanced Data	0.623	0.727	0.747	0.500	0.727
It2: Undersampling	0.697	0.721	0.732	0.539	0.716
It3: Normalization	0.696	0.721	0.749	0.660	0.727
It4: Hyper tuning	0.748	0.738	0.768	0.721	0.732
It5: Feature Selection	0.751	0.740	**0.773**	0.724	0.734

In each iteration, some of the algorithms show either sligh or incredible improvement in the performance. These results let us know the importance of each of these iterations, not only to solve the problem, but also to understand the best scenarios for each algorithm.



Final Decision

According to the ROC curve, **LightGBM is the champion with a ROC of 0.77**. Thus, it is the algorithm we are going to choose as the best model. The best performance is in the fifth iteration (after feature selection with 100 attributes). In order to deep more into the results, the confusion matrix is presented:

		pred		
		1	2	3
True	1	112	7	907
	2	0	159	13409
	3	3	66	78127

Precision	97.39%
Recall	10.92%

Conclusion

1. Crisp DM provides a complete framework to organize and develop Data Mining projects. Its use is highly recommendable.

2. Graphical methods as heat maps are important to determine locations where traffic accidents are more frequent; however, machine learning algorithms can help us to be more precise especially when data is bigger. Density based algorithms, as DBSCAN, seems to be the perfect choice to determine these hotspots of accidents since we can identify the concentration of them in a particular radius.
    a. Birmingham, Glasgow, Bristol, Coventry, Stoke-on-Trent have a high ratio of fatal accidents.
    b. Locations in the map that remained unclustered also have a high ratio of fatal accidents, which means that fatal accidents also tend to happen in roads far away from densely populated places. This is backed up by the feature importance and regression analysis performed in this project, where it is shown that driving in rural areas represent an important factor in accidents.
    c. Thorncliffe and London account for 50% of the total amount of fatal accidents.

Moreover, this clusterization is pivotal not only to know the main points, but also to assign a cluster to every row which can be an input in the following models. In addition, the clusterization of common accidents and fatal ones let us identify some critical points which are more prone to these events in the UK and they can be prioritized by government measures.

3. Data is the most valuable asset only if we can understand its influence in the problem. For this reason, variable statistics, bivariate graphics (against the target variable) combined with the domain expertise is a rule of thumb when planning a data mining project.

4. Based on the information provided by the UK government, it was possible to determine some of the most important factors involved in fatal accidents. However, this required several steps to be completed so a conclusion could be reached. After understanding data that was available, it was necessary to prepare it in order to not only identify which information could be useful, but also to make the necessary adjustments on it so it could be used with different modeling techniques. Using the processed data, it was possible to observe that **a fatal accident is more likely to happen while driving at high speed limits, poor lighting conditions on the road and also driving in rural areas.** Another factor that is also important is **the current condition of the road surface** (whether it is dry or wet). We would recommend actions to be taken on these variables to potentially reduce the amount of fatal accidents.

5. In order to predict fatal accidents, several models were used to identify one that could provide the best predictor and help to prevent accidents. After running the data through 5 experiments implemented for 5 different models, the results show that the LightGBM algorithm identifies with greater accuracy the potential accident severity based on the features previously identified. This would be the recommended model to use in order to better identify when, according to the current conditions, an accident could be fatal and take the precautionary actions.

References

1. Smart Vision Europe 2018 (2018). What is the CRISP-DM methodology?. Retrieved from https://www.sv-europe.com/crisp-dm-methodology/
2. IBM Corporation (2012). CRISP-DM Help Overview. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SS3RA7_15.0.0/com.ibm.spss.crispdm.help/crisp_overview.htm
3. Breiman, L. (2001). Random Forests. Retrieved from https://link.springer.com/content/pdf/10.1023%2FA%3A1010933404324.pdf
4. Paul, S. (August 15, 2018). Hyperparameter Optimization in Machine Learning Models. Retrieved from https://www.datacamp.com/community/tutorials/parameter-optimization-machine-learning-models
5. Narkhede, S. (June 26, 2018). Understanding AUC - ROC Curve. Retrieved from https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5
6. Morde, V. (April 8, 2019) XGBoost Algorithm: Long May She Reign! Retrieved from https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d