Seattle Car Accident Project (IBM Applied Data Science) - Report

Introduction

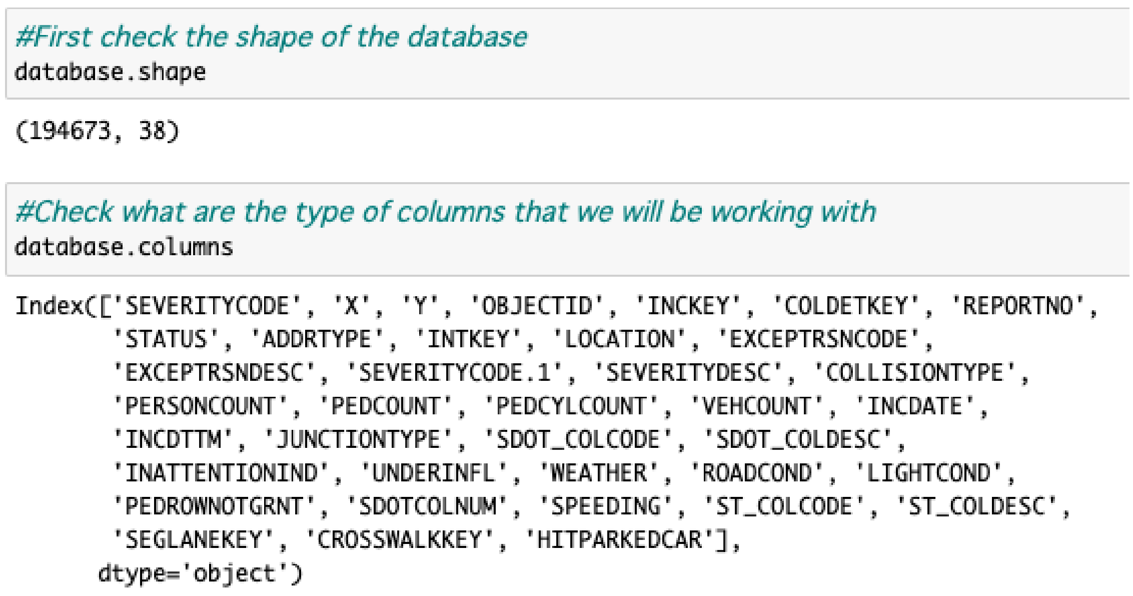
An estimated 1.2 million people are killed in road crashes each year, and as many as 50 million are injured.

Road traffic injuries are predictable and preventable, but the study of data is of utmost importance. Understanding the factors around which these tragedies occurs, will give us the insight needed in order to prevent them.

Data

Using the Database of the *SDOT Traffic Management Division of Seattle*, the main objectives of this project are: to build a Classification model that can predict the probability of one of these type of accidents (property damage or injury) and to get a better understanding of the key factors involved in road crashes.

The Database consist of 194,673 rows and 38 columns/features, one of which will be the dependent variable ("SEVERITYCODE") and thus, the one to predict. Its values are either 1 (property damage) or 2 (injury).



```
#First check the shape of the database
database.shape
```

```
(194673, 38)
```

```
#Check what are the type of columns that we will be working with
database.columns
```

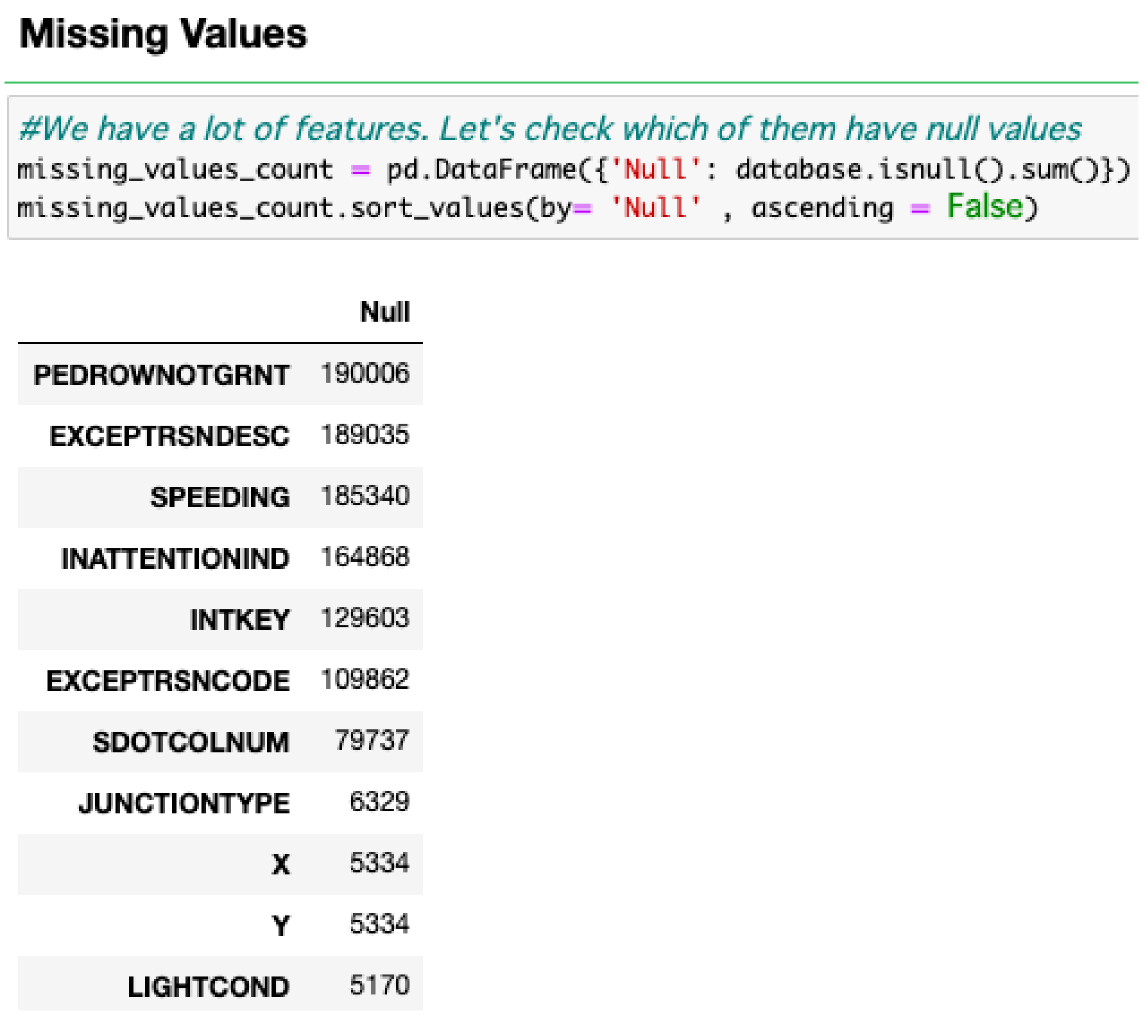
```
Index(['SEVERITYCODE', 'X', 'Y', 'OBJECTID', 'INCKEY', 'COLDETKEY', 'REPORTNO',
       'STATUS', 'ADDRTYPE', 'INTKEY', 'LOCATION', 'EXCEPTRSNCODE',
       'EXCEPTRSNDESC', 'SEVERITYCODE.1', 'SEVERITYDESC', 'COLLISIONTYPE',
       'PERSONCOUNT', 'PEDCOUNT', 'PEDCYLCOUNT', 'VEHCOUNT', 'INCDATE',
       'INCDTTM', 'JUNCTIONTYPE', 'SDOT_COLCODE', 'SDOT_COLDESC',
       'INATTENTIONIND', 'UNDERINFL', 'WEATHER', 'ROADCOND', 'LIGHTCOND',
       'PEDROWNOTGRNT', 'SDOTCOLNUM', 'SPEEDING', 'ST_COLCODE', 'ST_COLDESC',
       'SEGLANEKEY', 'CROSSWALKKEY', 'HITPARKEDCAR'],
      dtype='object')
```

Overview of the Database

Methodology

Missing values

For starters, it is important to check the missing values in the Database. That will be the basis for the next steps.



Missing Values

```
#We have a lot of features. Let's check which of them have null values
missing_values_count = pd.DataFrame({'Null': database.isnull().sum()})
missing_values_count.sort_values(by= 'Null' , ascending = False)
```

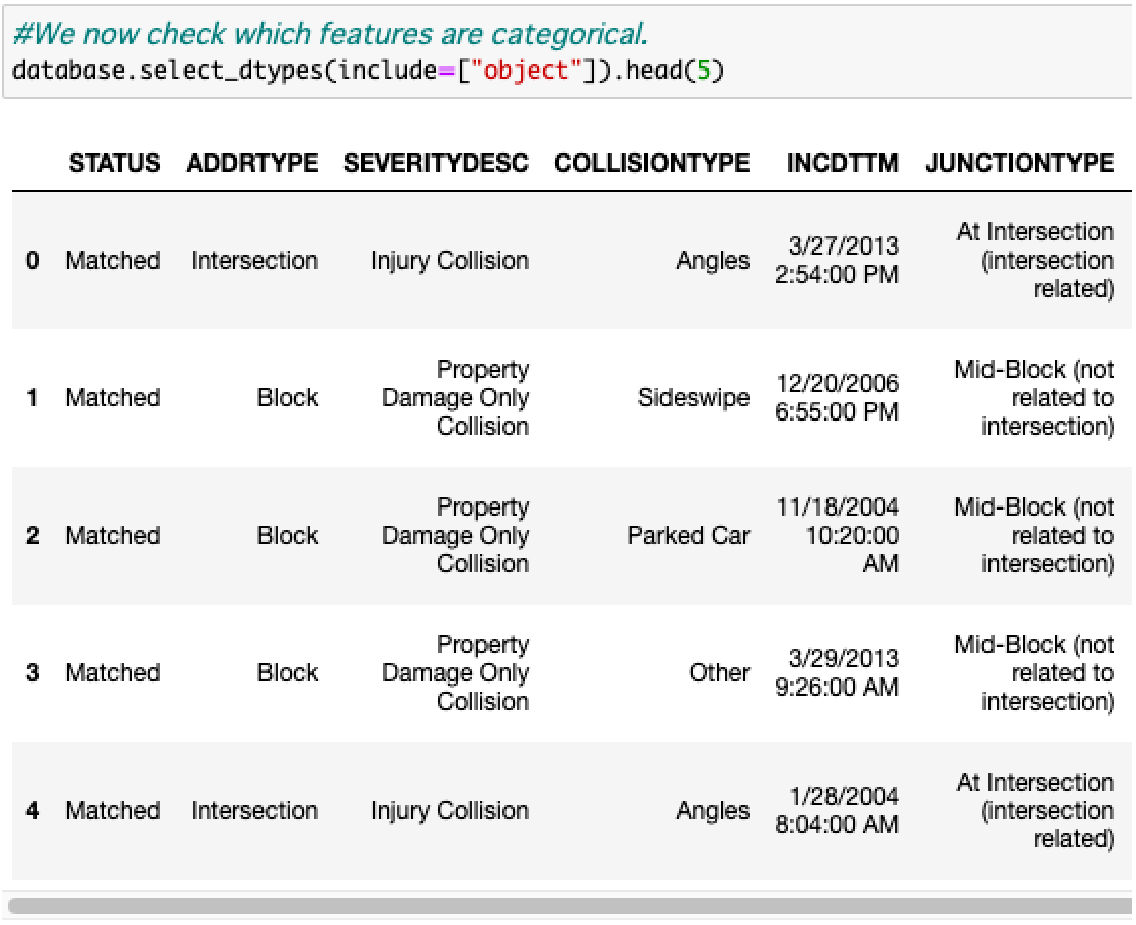
	Null
PEDROWNOTGRNT	190006
EXCEPTRSNDESC	189035
SPEEDING	185340
INATTENTIONIND	164868
INTKEY	129603
EXCEPTRSNCODE	109862
SDOTCOLNUM	79737
JUNCTIONTYPE	6329
X	5334
Y	5334
LIGHTCOND	5170

Missing values in our Database

In a Database of roughly 195,000 rows, having columns with as many as almost 110,000 missing values represents a massive lack of data. For this reason, it is a good idea to drop the 6 first features with the highest null (NaN) values, as shown in the image above.

Together with these, there are also some features that don't bring much value to the deployment of the model, such as the reportN0, and so they will be also dropped. To get the insight needed on what features would not be meaningful to build the model, I used the official *Attribute Information* provided together with the Database.

It is now time to take a look into the Categorical Features. Interestingly enough, there is a Date mixed in, which Python has not recognised as a date datatype. Thus, the Feature Engineering begins:



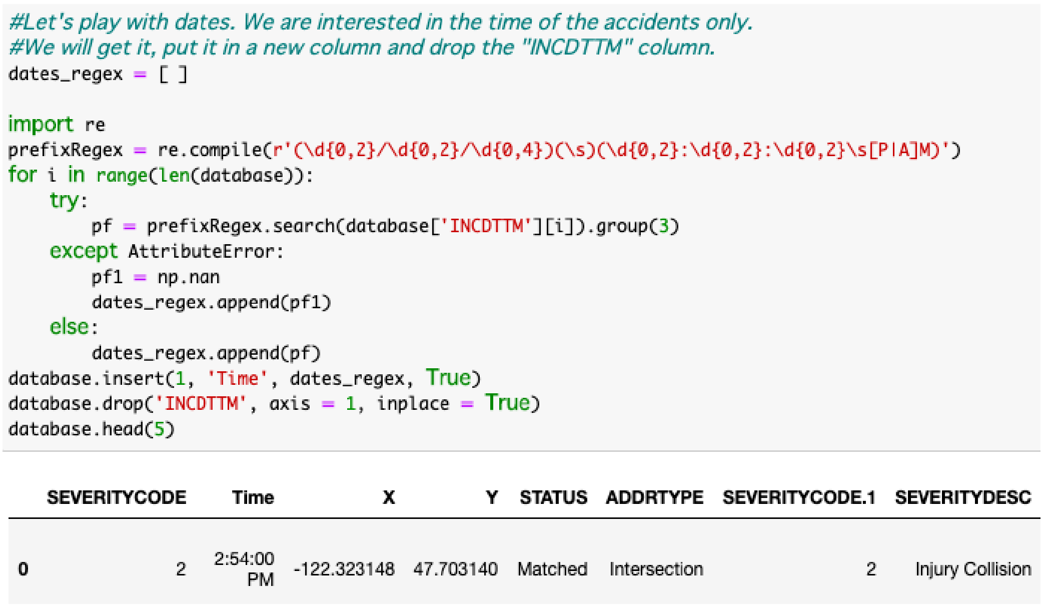
```python
#We now check which features are categorical.
database.select_dtypes(include=["object"]).head(5)
```

	STATUS	ADDRTYPE	SEVERITYDESC	COLLISIONTYPE	INCDTTM	JUNCTIONTYPE
0	Matched	Intersection	Injury Collision	Angles	3/27/2013 2:54:00 PM	At Intersection (intersection related)
1	Matched	Block	Property Damage Only Collision	Sideswipe	12/20/2006 6:55:00 PM	Mid-Block (not related to intersection)
2	Matched	Block	Property Damage Only Collision	Parked Car	11/18/2004 10:20:00 AM	Mid-Block (not related to intersection)
3	Matched	Block	Property Damage Only Collision	Other	3/29/2013 9:26:00 AM	Mid-Block (not related to intersection)
4	Matched	Intersection	Injury Collision	Angles	1/28/2004 8:04:00 AM	At Intersection (intersection related)

Some of the Categorical Features of the Database

Feature engineering

I am specially interested in the time zone where the accidents occurred. Since it is only a part of the whole date, Regular Expressions will be useful to match the wanted pattern and to later store it in a brand new column in our database.

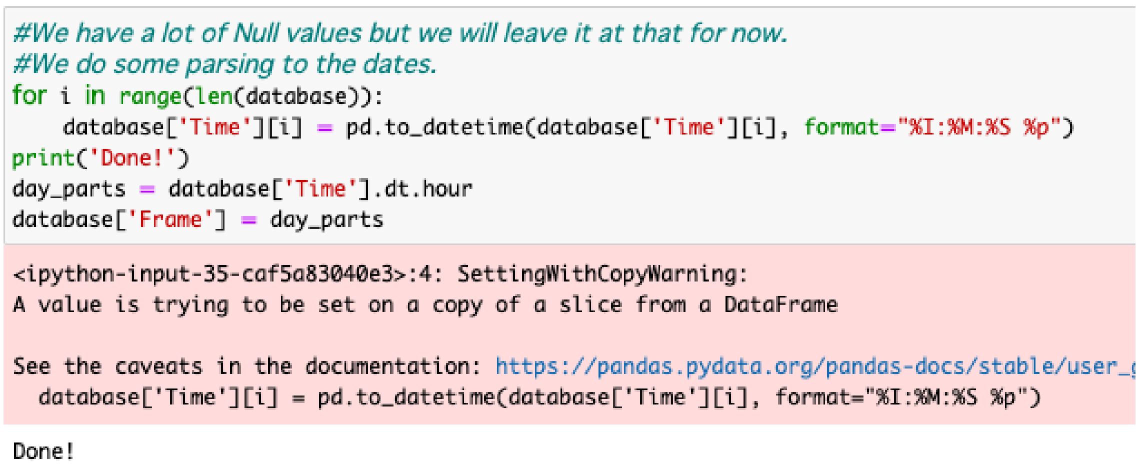


```python
#Let's play with dates. We are interested in the time of the accidents only.
#We will get it, put it in a new column and drop the "INCDTTM" column.
dates_regex = [ ]

import re
prefixRegex = re.compile(r'(\d{0,2}/\d{0,2}/\d{0,4})(\s)(\d{0,2}:\d{0,2}:\d{0,2}\s[P|A]M)')
for i in range(len(database)):
    try:
        pf = prefixRegex.search(database['INCDTTM'][i]).group(3)
    except AttributeError:
        pf1 = np.nan
        dates_regex.append(pf1)
    else:
        dates_regex.append(pf)
database.insert(1, 'Time', dates_regex, True)
database.drop('INCDTTM', axis = 1, inplace = True)
database.head(5)
```

	SEVERITYCODE	Time	X	Y	STATUS	ADDRTYPE	SEVERITYCODE.1	SEVERITYDESC
0	2	2:54:00 PM	-122.323148	47.703140	Matched	Intersection	2	Injury Collision

Pattern Matching with Regular Expressions

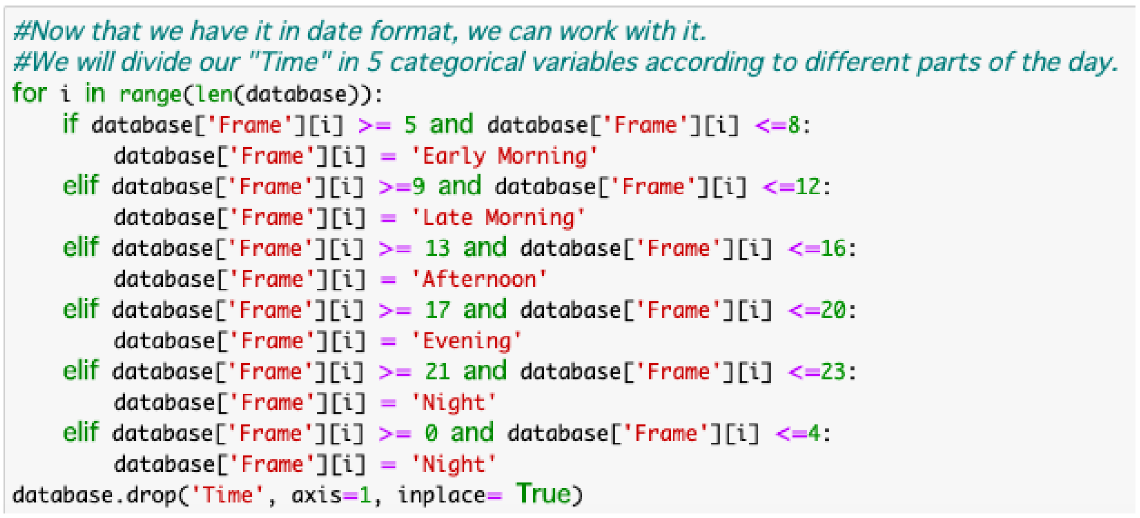
Now, for the main issue. Even though there are missing values, for Python to recognise the value as a proper date datatype, parsing is necessary. The goal in this whole procedure is to extract the hour (as an *int*) and to create time frames (parts of the day), which is done in the following steps:



```
#We have a lot of Null values but we will leave it at that for now.
#We do some parsing to the dates.
for i in range(len(database)):
    database['Time'][i] = pd.to_datetime(database['Time'][i], format="%I:%M:%S %p")
print('Done!')
day_parts = database['Time'].dt.hour
database['Frame'] = day_parts
```

```
<ipython-input-35-caf5a83040e3>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
  database['Time'][i] = pd.to_datetime(database['Time'][i], format="%I:%M:%S %p")
```

```
Done!
```



```
#Now that we have it in date format, we can work with it.
#We will divide our "Time" in 5 categorical variables according to different parts of the day.
for i in range(len(database)):
    if database['Frame'][i] >= 5 and database['Frame'][i] <=8:
        database['Frame'][i] = 'Early Morning'
    elif database['Frame'][i] >=9 and database['Frame'][i] <=12:
        database['Frame'][i] = 'Late Morning'
    elif database['Frame'][i] >= 13 and database['Frame'][i] <=16:
        database['Frame'][i] = 'Afternoon'
    elif database['Frame'][i] >= 17 and database['Frame'][i] <=20:
        database['Frame'][i] = 'Evening'
    elif database['Frame'][i] >= 21 and database['Frame'][i] <=23:
        database['Frame'][i] = 'Night'
    elif database['Frame'][i] >= 0 and database['Frame'][i] <=4:
        database['Frame'][i] = 'Night'
database.drop('Time', axis=1, inplace= True)
```

Parsing and Time Frames

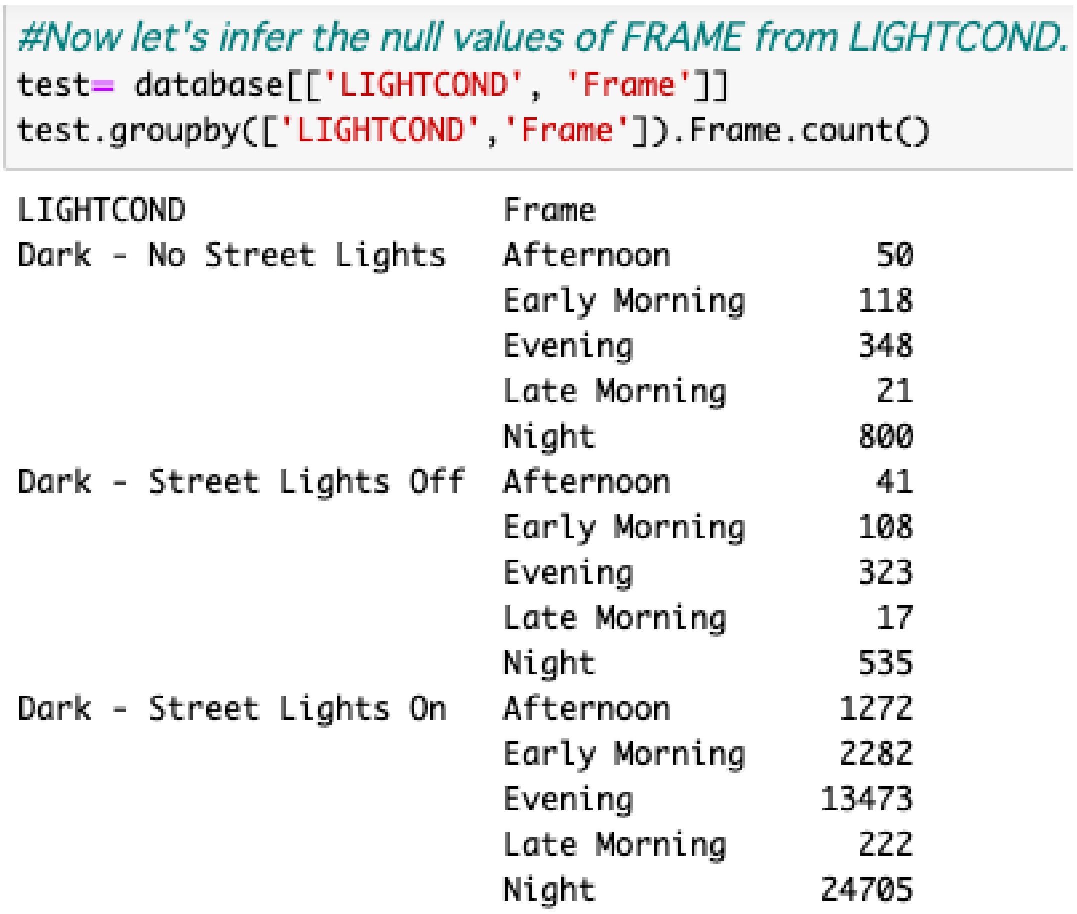
But, what about the NaN values? Since this database has multiple columns with interdependent information, it is possible to use another column to deduce the rest of values.

In this case, the feature “LIGHTCOND” will be the one used as a reference column to fill the missing values of the “Frame” feature we just created. Thus, the goal will be to deduce what time frame of day did the accident happened in by making use of the information on the light conditions when each accident occurred.

Since it is difficult to infer anything if both columns have missing values in the same row, in case that situation arose, the row will be dropped.

That aside, in order to be as exact as possible, “GroupBY” will be the tool used in this part of the kernel. This way, it is possible to know which values are more connected to one each other depending on the frequency they appear on the same row.

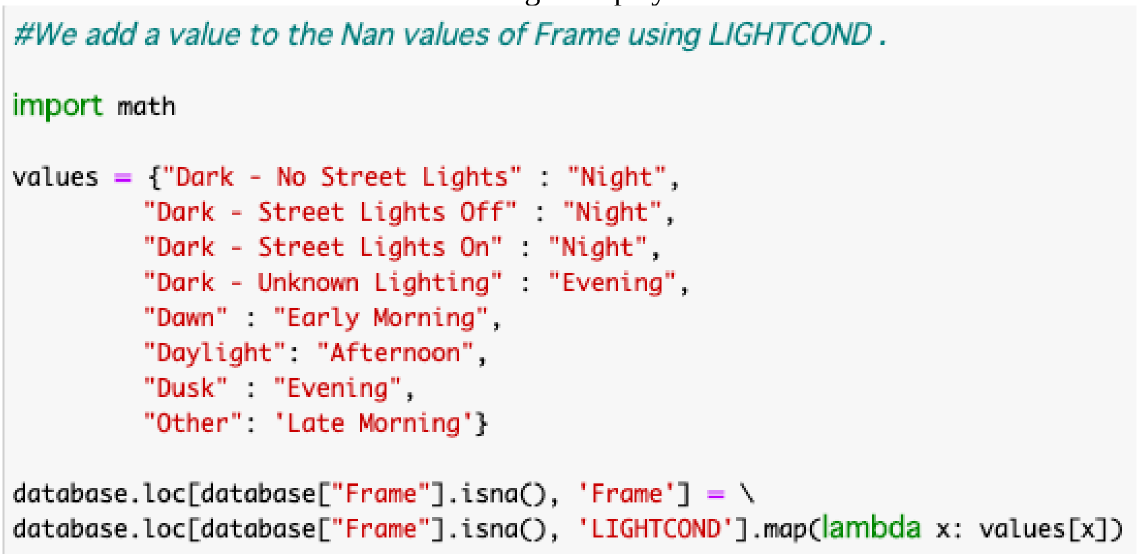
For example, it is clear from the below image that when the reference column "LIGHTCOND" has a value of "Dark—No Street Lights", it is relatively safe to conclude that the missing value of the column "Frame" should be "Night".



```
#Now let's infer the null values of FRAME from LIGHTCOND.
test= database[['LIGHTCOND', 'Frame']]
test.groupby(['LIGHTCOND','Frame']).Frame.count()
```

```
LIGHTCOND                 Frame
Dark - No Street Lights   Afternoon          50
                          Early Morning     118
                          Evening           348
                          Late Morning       21
                          Night             800
Dark - Street Lights Off  Afternoon          41
                          Early Morning     108
                          Evening           323
                          Late Morning       17
                          Night             535
Dark - Street Lights On   Afternoon        1272
                          Early Morning    2282
                          Evening         13473
                          Late Morning      222
                          Night           24705
```

Understanding the relationship between columns using GroupBy



```
#We add a value to the Nan values of Frame using LIGHTCOND .

import math

values = {"Dark - No Street Lights" : "Night",
         "Dark - Street Lights Off" : "Night",
         "Dark - Street Lights On" : "Night",
         "Dark - Unknown Lighting" : "Evening",
         "Dawn" : "Early Morning",
         "Daylight": "Afternoon",
         "Dusk" : "Evening",
         "Other": 'Late Morning'}

database.loc[database["Frame"].isna(), 'Frame'] = \
database.loc[database["Frame"].isna(), 'LIGHTCOND'].map(lambda x: values[x])
```

Assigning new values making use of interconnected columns

Just like the values of the column "Frame" were deduced thanks to the values of the reference column "LIGHTCOND", it will be also done the other way around. This way, we can fill both columns' missing values.

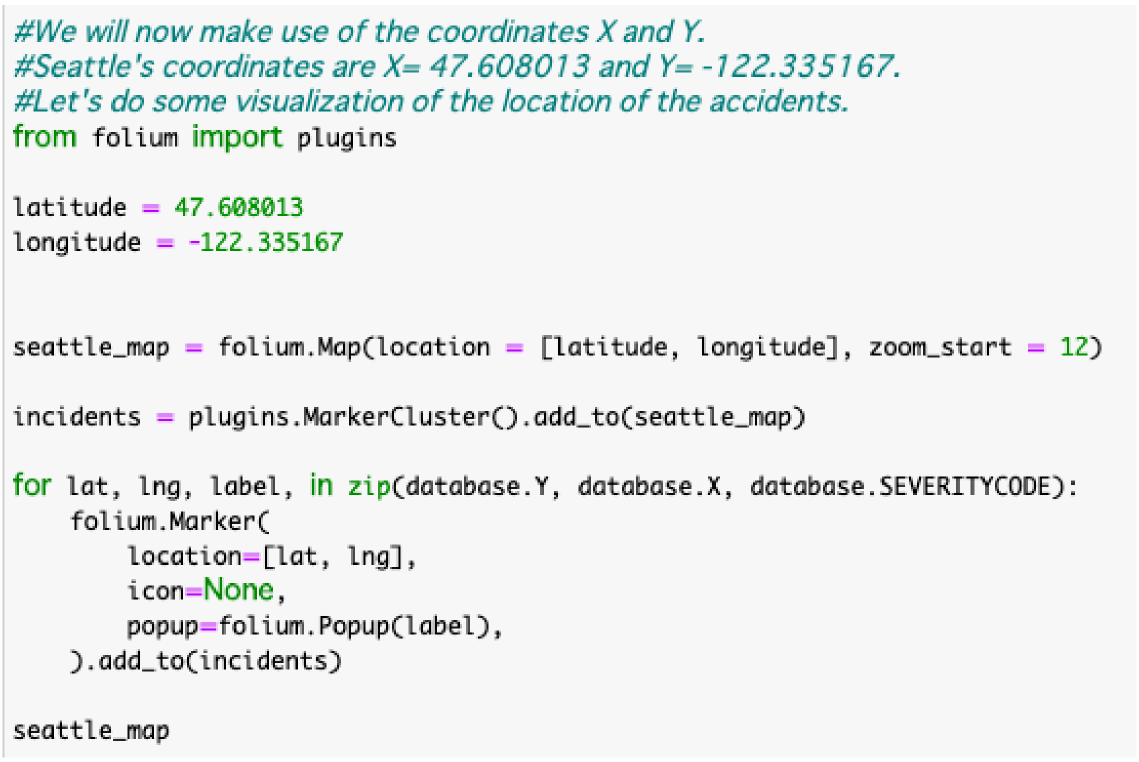
I believe this process really enriched the Database, which had many columns with missing data. Indeed, meaningful connections between columns could be created thanks to these series of steps. But, will it create a high correlation between them? Let's explore that later.

The same procedure is followed for each column that has some missing data, by creating pair of columns and trying to fill the gaps left by the missing values (see Road Condition & Weather / Junction type & Place of accident / Collision description & Collision code).

And just like that, at last, the time to work with the coordinates X and Y has come.

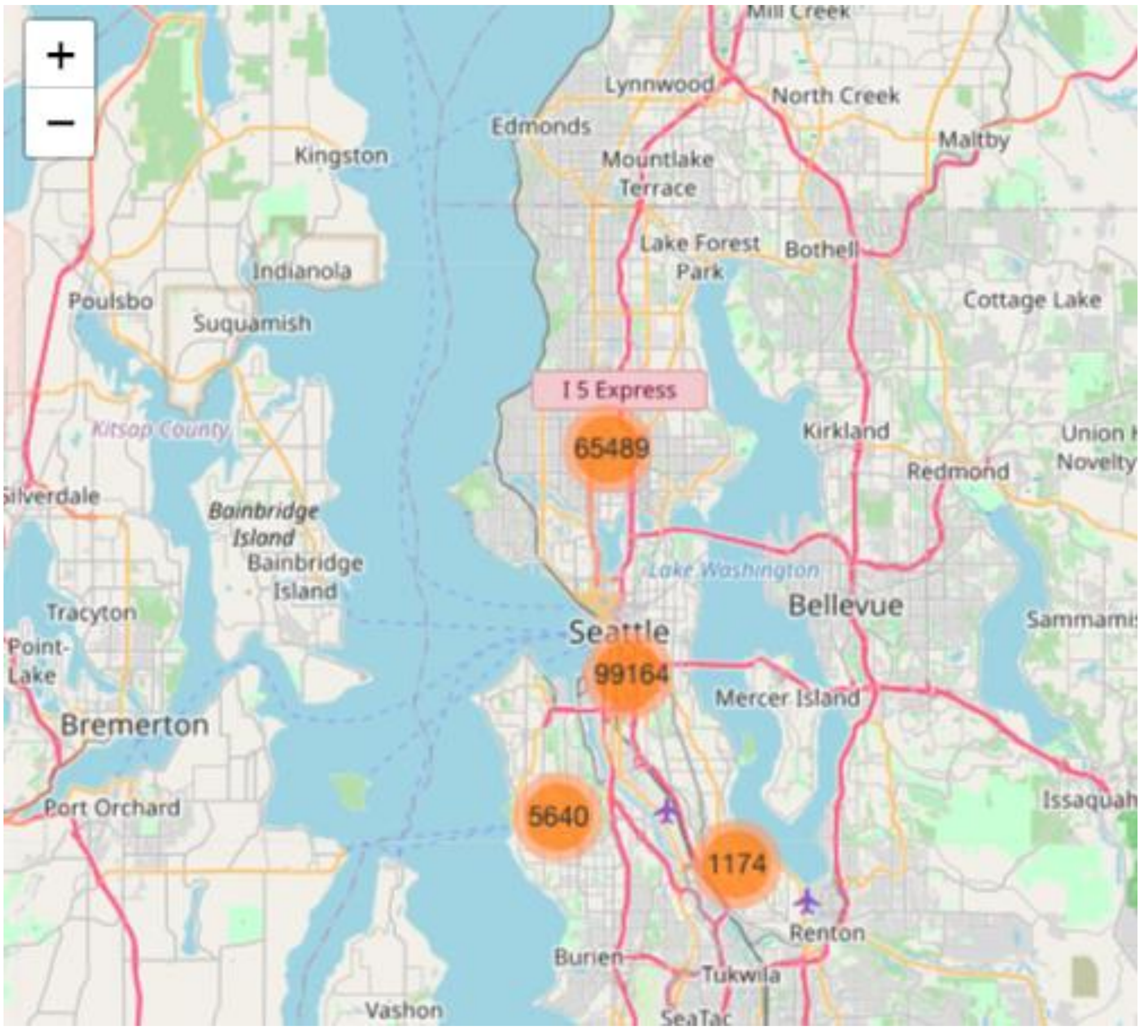
What I want achieve by working with these two features is to divide each accident into geographical clusters and add that data as a new feature into the database.

Folium is a good tool to visualise the accidents grouped by location in an interactive way. The map can be zoomed in or out to check bigger clusters of accidents or even the place where each accident occurred:



```
#We will now make use of the coordinates X and Y.
#Seattle's coordinates are X= 47.608013 and Y= -122.335167.
#Let's do some visualization of the location of the accidents.
from folium import plugins

latitude = 47.608013
longitude = -122.335167


seattle_map = folium.Map(location = [latitude, longitude], zoom_start = 12)

incidents = plugins.MarkerCluster().add_to(seattle_map)

for lat, lng, label, in zip(database.Y, database.X, database.SEVERITYCODE):
    folium.Marker(
        location=[lat, lng],
        icon=None,
        popup=folium.Popup(label),
    ).add_to(incidents)

seattle_map
```

Setting up the map visualization
using Folium



Clusters of accidents in Seattle from our Database

To do the real clustering in the database, K-Means will be used. To reduce the workload, it is advisable to create another dataset, slicing the main database and taking the columns needed.

The dataset created in this case is called “X” and has the coordinates X and Y and the column “OBJECTID”, which will be useful to connect this dataset to the main database once the clustering is finished.

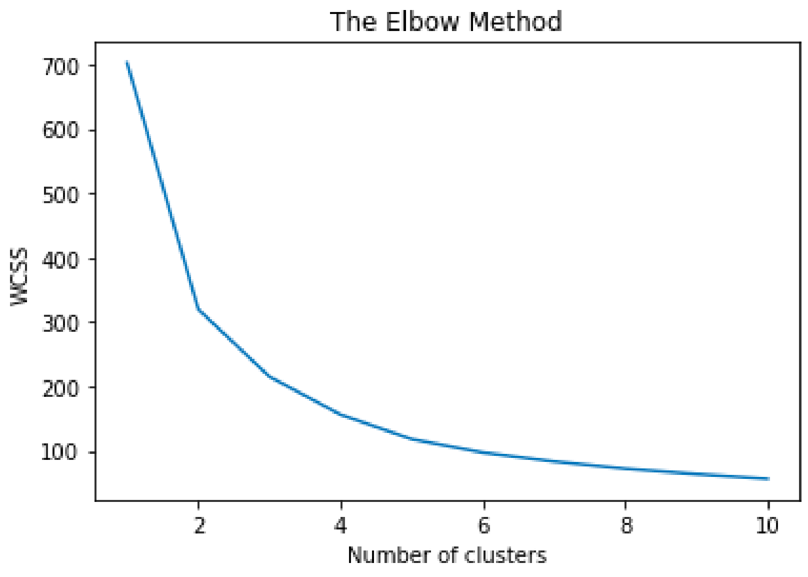
```
X = database.loc[:, ['OBJECTID','X','Y']]
X.head(5)
```

	OBJECTID	X	Y
0	1	-122.323148	47.703140
1	2	-122.347294	47.647172
2	3	-122.334540	47.607871
3	4	-122.334803	47.604803
4	5	-122.306426	47.545739

New “X” dataset is created from our database

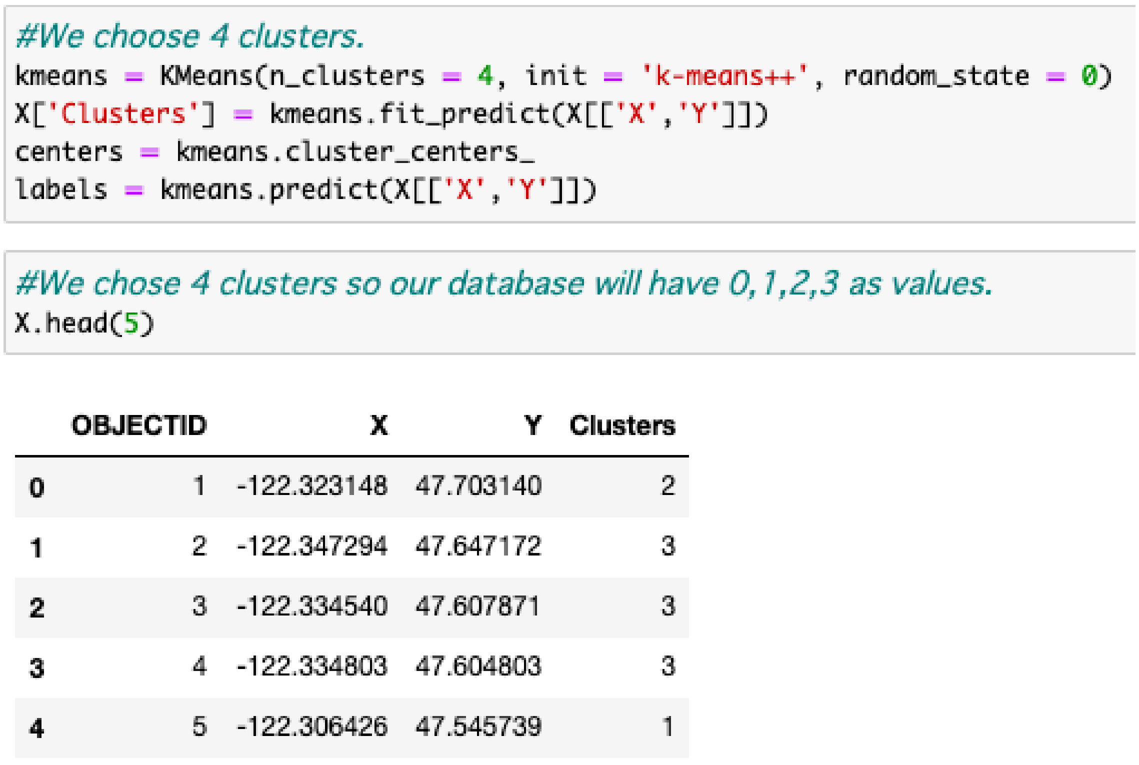
Before fitting K-Means into the “X” dataset, it is necessary to check the optimal number of clusters by using the Elbow Method.

```
#First, we use the Elbow Method to choose the optimal nº of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 0)
    kmeans.fit(X[['X','Y']])
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



Following this step, I ended up choosing 4 clusters and fitting the model into our dataset. Once the clustering is done and each pair of coordinates is assigned to a cluster (0,1,2 or 3), it is time to add the column "Clusters" to the main database.

Finally, the columns used for the clustering are dropped since they won't be needed any further.

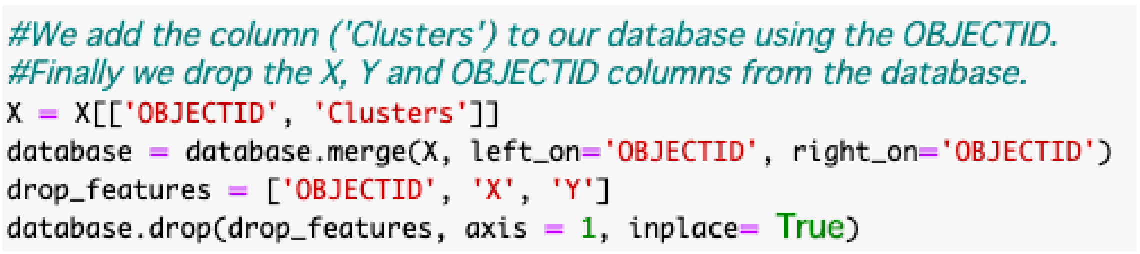


```python
#We choose 4 clusters.
kmeans = KMeans(n_clusters = 4, init = 'k-means++', random_state = 0)
X['Clusters'] = kmeans.fit_predict(X[['X','Y']])
centers = kmeans.cluster_centers_
labels = kmeans.predict(X[['X','Y']])
```

```python
#We chose 4 clusters so our database will have 0,1,2,3 as values.
X.head(5)
```

	OBJECTID	X	Y	Clusters
0	1	-122.323148	47.703140	2
1	2	-122.347294	47.647172	3
2	3	-122.334540	47.607871	3
3	4	-122.334803	47.604803	3
4	5	-122.306426	47.545739	1

Use of K-Means in our "X" dataset

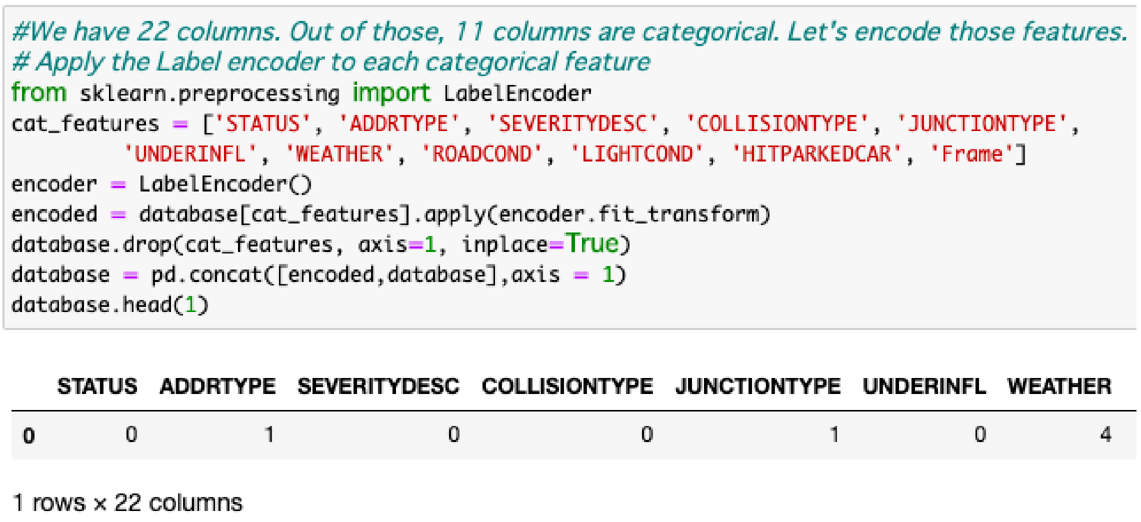


```python
#We add the column ('Clusters') to our database using the OBJECTID.
#Finally we drop the X, Y and OBJECTID columns from the database.
X = X[['OBJECTID', 'Clusters']]
database = database.merge(X, left_on='OBJECTID', right_on='OBJECTID')
drop_features = ['OBJECTID', 'X', 'Y']
database.drop(drop_features, axis = 1, inplace= True)
```

Merging the "X" dataset and our main database using "OBJECTID" feature

Categorical Variable Encoding

In the database both numerical and categorical variable coexist. But, in order to carry on with the following steps, it is required to encode those categorical values into numerical data. Label Encoding will be helpful for this endeavour.



```python
#We have 22 columns. Out of those, 11 columns are categorical. Let's encode those features.
# Apply the Label encoder to each categorical feature
from sklearn.preprocessing import LabelEncoder
cat_features = ['STATUS', 'ADDRTYPE', 'SEVERITYDESC', 'COLLISIONTYPE', 'JUNCTIONTYPE',
         'UNDERINFL', 'WEATHER', 'ROADCOND', 'LIGHTCOND', 'HITPARKEDCAR', 'Frame']
encoder = LabelEncoder()
encoded = database[cat_features].apply(encoder.fit_transform)
database.drop(cat_features, axis=1, inplace=True)
database = pd.concat([encoded,database],axis = 1)
database.head(1)
```

	STATUS	ADDRTYPE	SEVERITYDESC	COLLISIONTYPE	JUNCTIONTYPE	UNDERINFL	WEATHER
0	0	1	0	0	1	0	4

1 rows × 22 columns

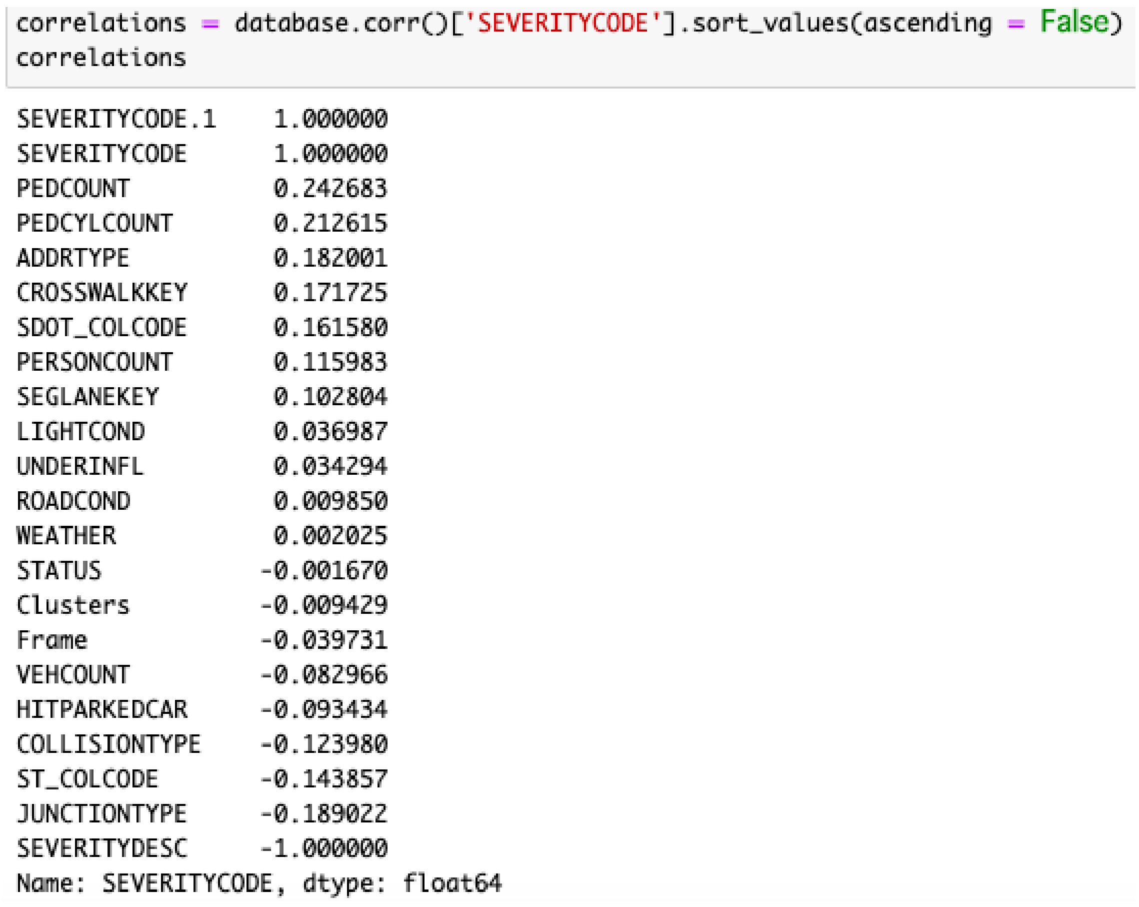
Categorical Variable Encoding

Correlations

The features in the database are correlated both between each other and with the dependent variable (“SEVERITYCODE”). The problem comes when they are highly correlated.

Features with high correlation are more linearly dependent and hence have almost the same effect on the dependent variable. So, when two features have high correlation, we can drop one of the two features.

First, the correlation between the independent features and the dependent variable is checked. The features highly correlated with the dependent variable are dropped(in this case, “SEVERITYDESC” and “SEVERITYCODE.1”).

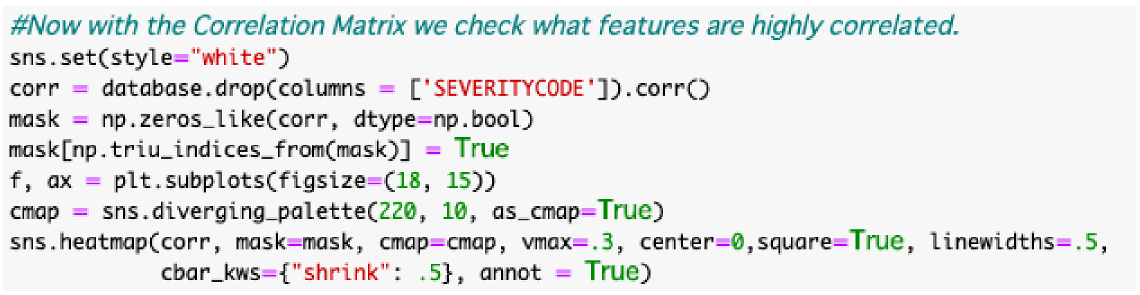


```
correlations = database.corr()['SEVERITYCODE'].sort_values(ascending = False)
correlations
```

```
SEVERITYCODE.1     1.000000
SEVERITYCODE       1.000000
PEDCOUNT           0.242683
PEDCYLCOUNT        0.212615
ADDRTYPE           0.182001
CROSSWALKKEY       0.171725
SDOT_COLCODE       0.161580
PERSONCOUNT        0.115983
SEGLANEKEY         0.102804
LIGHTCOND          0.036987
UNDERINFL          0.034294
ROADCOND           0.009850
WEATHER            0.002025
STATUS            -0.001670
Clusters          -0.009429
Frame             -0.039731
VEHCOUNT          -0.082966
HITPARKEDCAR      -0.093434
COLLISIONTYPE     -0.123980
ST_COLCODE        -0.143857
JUNCTIONTYPE      -0.189022
SEVERITYDESC      -1.000000
Name: SEVERITYCODE, dtype: float64
```

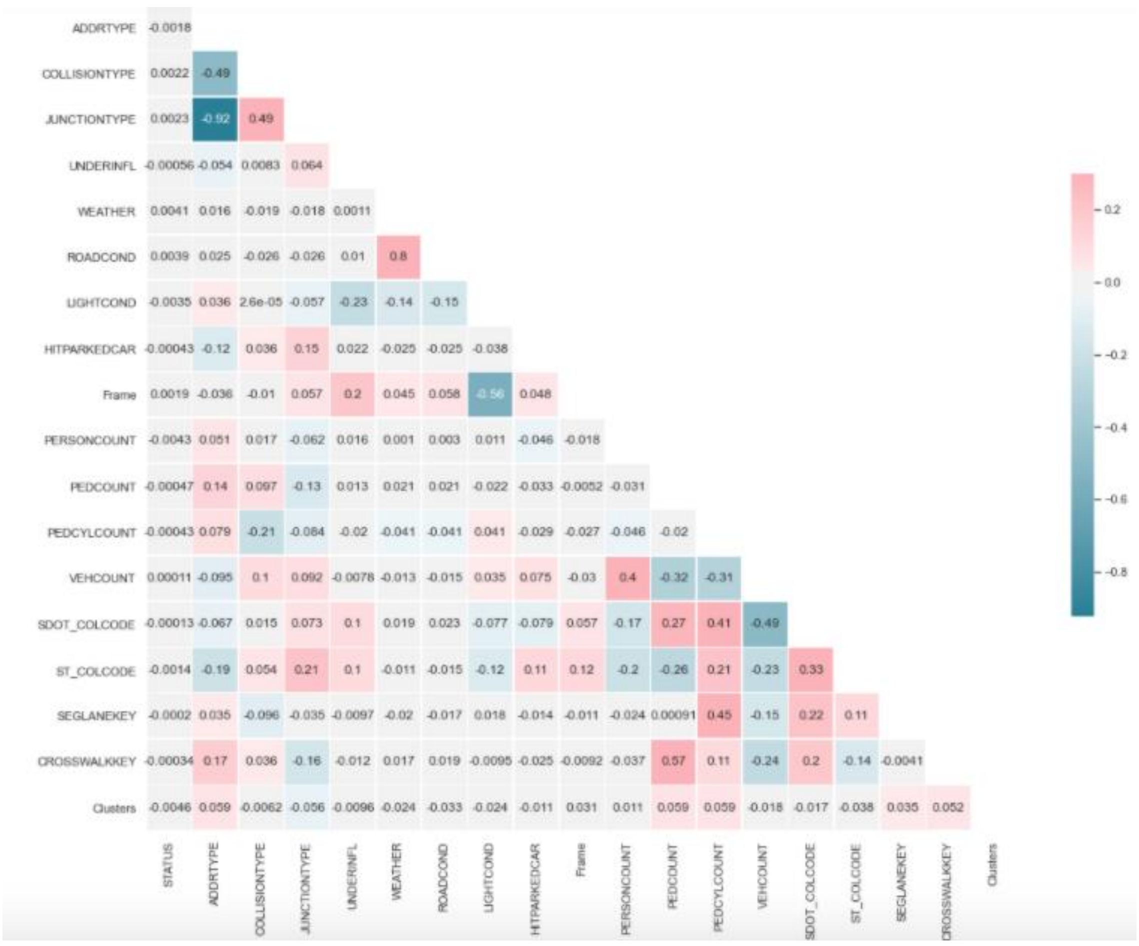
Correlation chart with our dependent variable as the reference

After that, we check how the independent variables are correlated between each other using the Correlation Matrix:



```
#Now with the Correlation Matrix we check what features are highly correlated.
sns.set(style="white")
corr = database.drop(columns = ['SEVERITYCODE']).corr()
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
f, ax = plt.subplots(figsize=(18, 15))
cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,square=True, linewidths=.5,
            cbar_kws={"shrink": .5}, annot = True)
```

Correlation Matrix for our independent
features



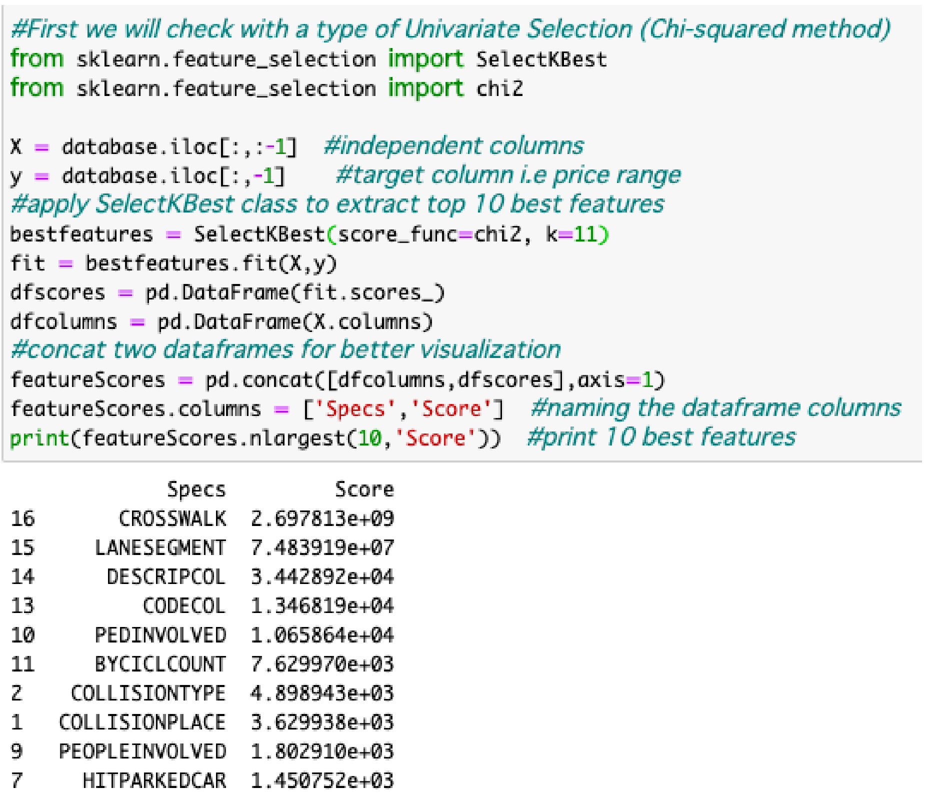
Visualisation of the Correlation Matrix

The features “ADDRTYPE” and “JUNCTIONTYPE” are highly correlated (0,92) so it is advisable to drop one of them. In this case “JUNCTIONTYPE” is the one chosen since it had more missing values to begin with.

Feature Selection

Another important step on building a meaningful model is to select only the most meaningful features to feed our model. Remember: *Garbage in, Garbage out*. That’s the reason why this step is so important and why I decided to select the top 10 most relevant features in the database.

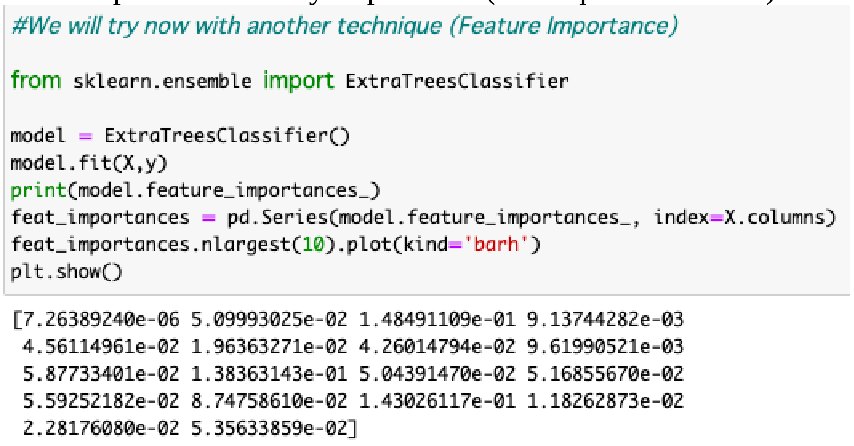
What’s more, in an attempt to compare different selection models, two different Feature Selection methods will be applied for this project: Chi-Squared and Feature Importance.



```
#First we will check with a type of Univariate Selection (Chi-squared method)
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

X = database.iloc[:,:-1]  #independent columns
y = database.iloc[:,-1]    #target column i.e price range
#apply SelectKBest class to extract top 10 best features
bestfeatures = SelectKBest(score_func=chi2, k=11)
fit = bestfeatures.fit(X,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
print(featureScores.nlargest(10,'Score'))  #print 10 best features
```

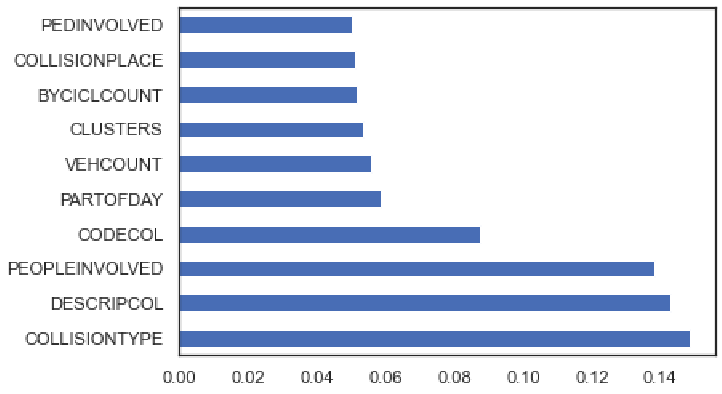
```
            Specs         Score
16      CROSSWALK  2.697813e+09
15    LANESEGMENT  7.483919e+07
14     DESCRIPCOL  3.442892e+04
13        CODECOL  1.346819e+04
10    PEDINVOLVED  1.065864e+04
11    BYCICLCOUNT  7.629970e+03
2   COLLISIONTYPE  4.898943e+03
1  COLLISIONPLACE  3.629938e+03
9  PEOPLEINVOLVED  1.802910e+03
7    HITPARKEDCAR  1.450752e+03
```

Top 10 Features by importance (Chi-Squared Method)



```
#We will try now with another technique (Feature Importance)

from sklearn.ensemble import ExtraTreesClassifier

model = ExtraTreesClassifier()
model.fit(X,y)
print(model.feature_importances_)
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()
```

```
[7.26389240e-06 5.09993025e-02 1.48491109e-01 9.13744282e-03
 4.56114961e-02 1.96363271e-02 4.26014794e-02 9.61990521e-03
 5.87733401e-02 1.38363143e-01 5.04391470e-02 5.16855670e-02
 5.59252182e-02 8.74758610e-02 1.43026117e-01 1.18262873e-02
 2.28176080e-02 5.35633859e-02]
```



Top 10 Features by importance (Feature Importance Method)

From the images above, it is noticeable that the features chosen by each method are different.

For this reason, we will build two different modelling processes using the results of each one of the Feature Selection methods. At the end, we will compare the results and see which one had a better accuracy in its predictions.

Undersampling

It is not wise to jump onto the modelling just like that. As I explained in the beginning, the dependent variable ("SEVERITYCODE") has two values (1 = "property damage" / 2 = "injury"). The issue here is that it doesn't exist a balance between the number of rows that have one value and those that have the other.

In fact, the database has around three times more samples/rows where the dependent variable is equal to 1 than when it is equal to 2.

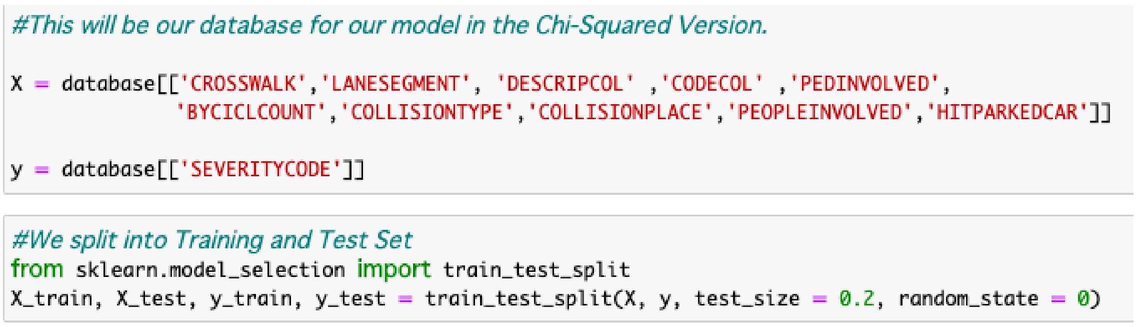
This bias in the training dataset can influence many machine learning algorithms, leading some to ignore the minority class entirely.

Thus, it is needed to adjust the sampling so that both values of the dependent variable appear the same number of times. This will be done during the training of the model.

Modelling

It is finally time to start the first modelling. The features selected by the Chi-Squared method are the ones that will be used in this first modelling.

For starters, the database is split into Training and Test Set in a ratio 8:2.

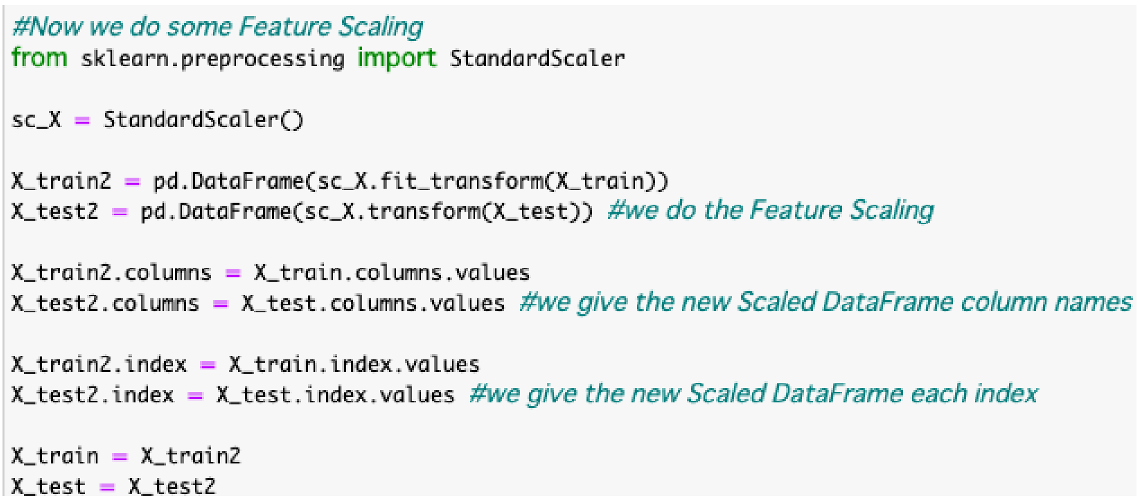


```
#This will be our database for our model in the Chi-Squared Version.

X = database[['CROSSWALK','LANESEGMENT', 'DESCRIPCOL' ,'CODECOL' ,'PEDINVOLVED',
              'BYCICLCOUNT','COLLISIONTYPE','COLLISIONPLACE','PEOPLEINVOLVED','HITPARKEDCAR']]

y = database[['SEVERITYCODE']]
```

```
#We split into Training and Test Set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Splitting into Training and Test Set (Chi-Squared version)

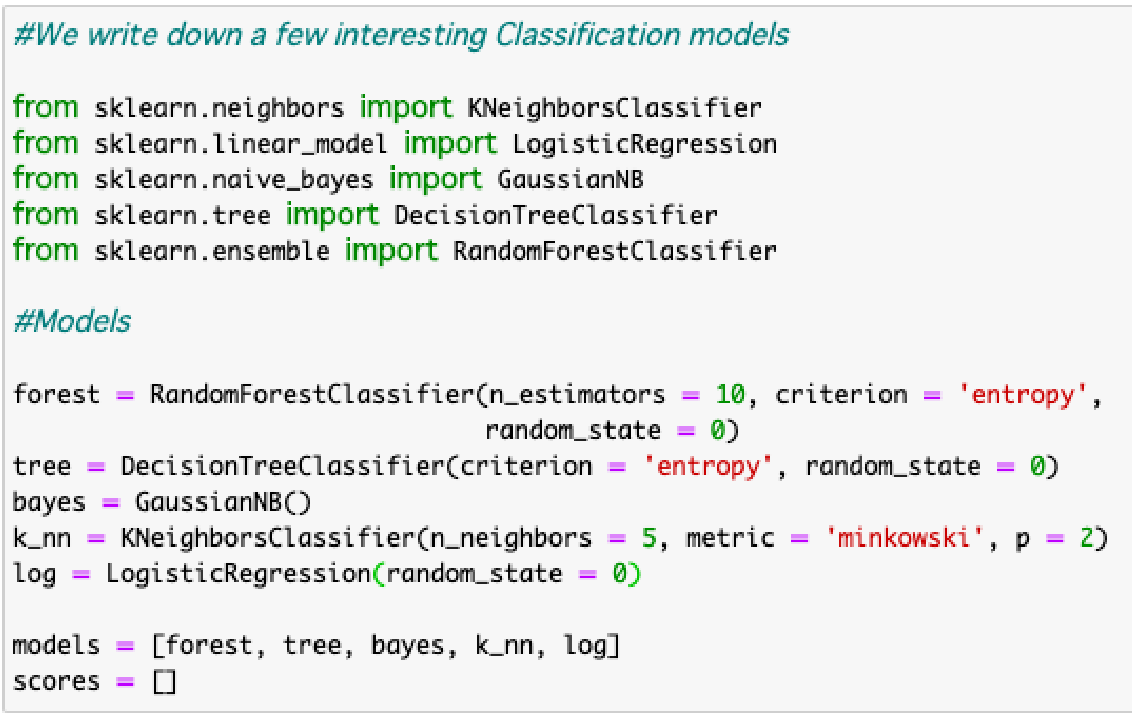
After this, Feature Scaling is applied to even the values of the independent features. It is really important to always do it AFTER splitting the database to avoid data leakage into the Machine Learning model.



```python
#Now we do some Feature Scaling
from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()

X_train2 = pd.DataFrame(sc_X.fit_transform(X_train))
X_test2 = pd.DataFrame(sc_X.transform(X_test)) #we do the Feature Scaling

X_train2.columns = X_train.columns.values
X_test2.columns = X_test.columns.values #we give the new Scaled DataFrame column names

X_train2.index = X_train.index.values
X_test2.index = X_test.index.values #we give the new Scaled DataFrame each index

X_train = X_train2
X_test = X_test2
```

Applying Feature Scaling to X_train and X_test

Now it is time to use a few interesting Classification Models and compare their individual performance. Also, in order to have a solid grasp on their accuracy, I will be using Cross-Validation and rank each one of them on a customised chart. All of this will be carried out in the Training Set.



```python
#We write down a few interesting Classification models

from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

#Models

forest = RandomForestClassifier(n_estimators = 10, criterion = 'entropy',
                                random_state = 0)
tree = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
bayes = GaussianNB()
k_nn = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
log = LogisticRegression(random_state = 0)

models = [forest, tree, bayes, k_nn, log]
scores = []
```

Classification models chosen for this project

Important to note is that I will be using Stratified K-Fold Cross Validation instead of the usual K-Fold Cross Validation since we are dealing with an unbalanced database, as mentioned before.

The idea behind this is to train the model on a balanced Training Set, so that it learns from the same number of samples where the value of the dependent variable is 1 and 2.

Once trained on the balanced Training Set, we will do the predictions on the unbalanced Test Set, which is the real one and check how accurate the trained model is. The reason why we cannot test it on a balanced Test Set is because the results wouldn't be adjusted to the reality of the database.

```python
#Since we have an unbalanced database we use StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
for mod in models:
    mod.fit(X_train, y_train.values.ravel())
    acc = cross_val_score(mod, X_train, y_train.values.ravel(),
                          scoring = "accuracy", cv = kfold, n_jobs = -1)
    scores.append(acc.mean())
```

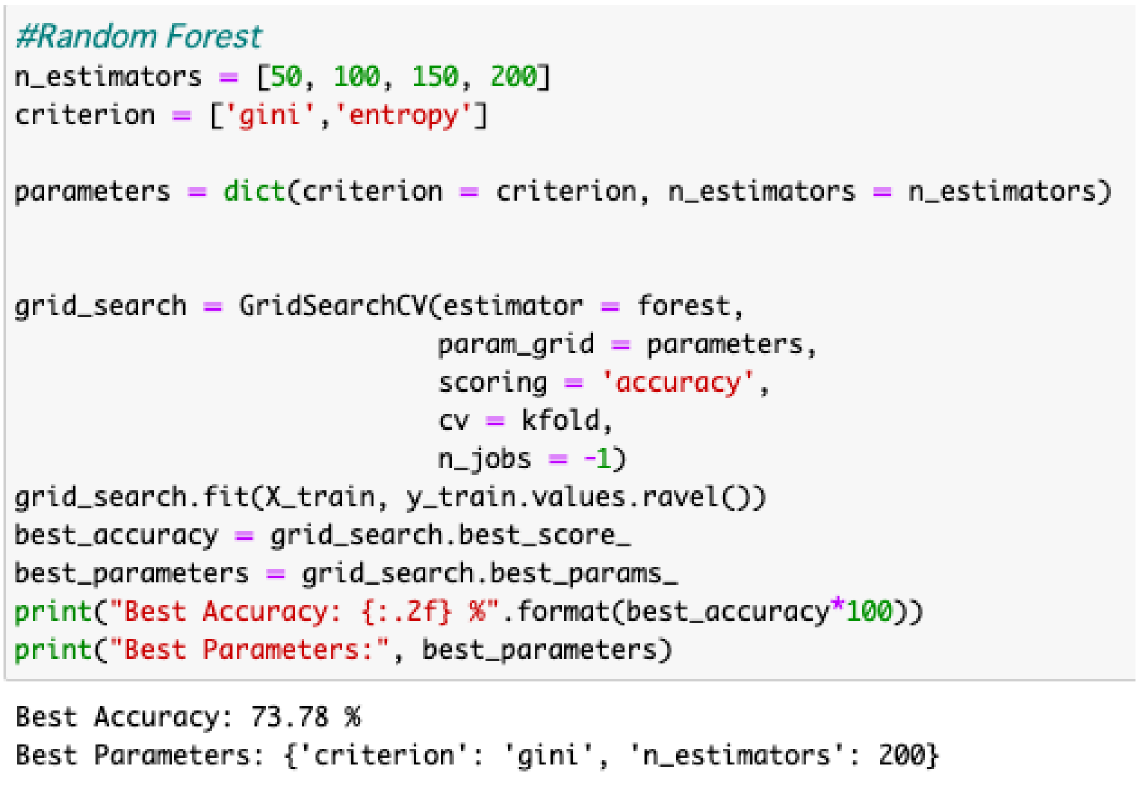
```python
#Here we create a table for visualization
results = pd.DataFrame({
    'Model': ['Random Forest', 'Decision Tree', 'Naive Bayes',
              'K Nearest Neighbour', 'Logistic Regression'],
    'Accuracy (Chi-Squared)': scores})

result_df = results.sort_values(by='Accuracy (Chi-Squared)',
                                ascending=False).reset_index(drop=True)
result_df
```

	Model	Accuracy (Chi-Squared)
0	Random Forest	0.737135
1	Decision Tree	0.736384
2	Logistic Regression	0.732068
3	Naive Bayes	0.730173
4	K Nearest Neighbour	0.711992

Stratified Cross-Validation and chart of each ML model

The results can be further improved. To that end, the next step will be to optimise their hyper-parameters.

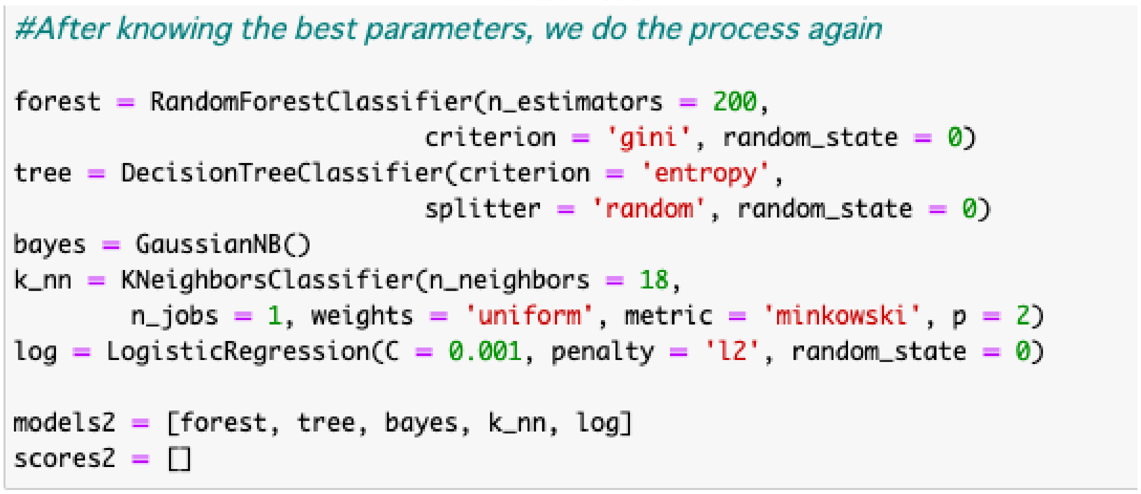
Once the hyper-parameters of each model are adjusted, it is time to train the model again (using Stratified K-Fold Cross-Validation) and check their performance on the Training Set one more time.



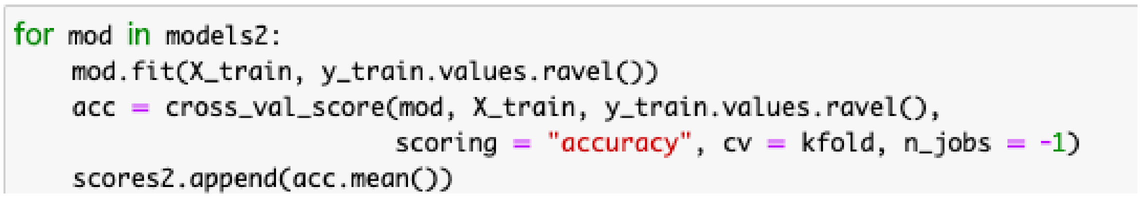
```
#Random Forest
n_estimators = [50, 100, 150, 200]
criterion = ['gini','entropy']

parameters = dict(criterion = criterion, n_estimators = n_estimators)


grid_search = GridSearchCV(estimator = forest,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = kfold,
                           n_jobs = -1)
grid_search.fit(X_train, y_train.values.ravel())
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
print("Best Accuracy: {:.2f} %".format(best_accuracy*100))
print("Best Parameters:", best_parameters)
```

```
Best Accuracy: 73.78 %
Best Parameters: {'criterion': 'gini', 'n_estimators': 200}
```

Example of Hyper-parameter Tuning with the Random Forest model

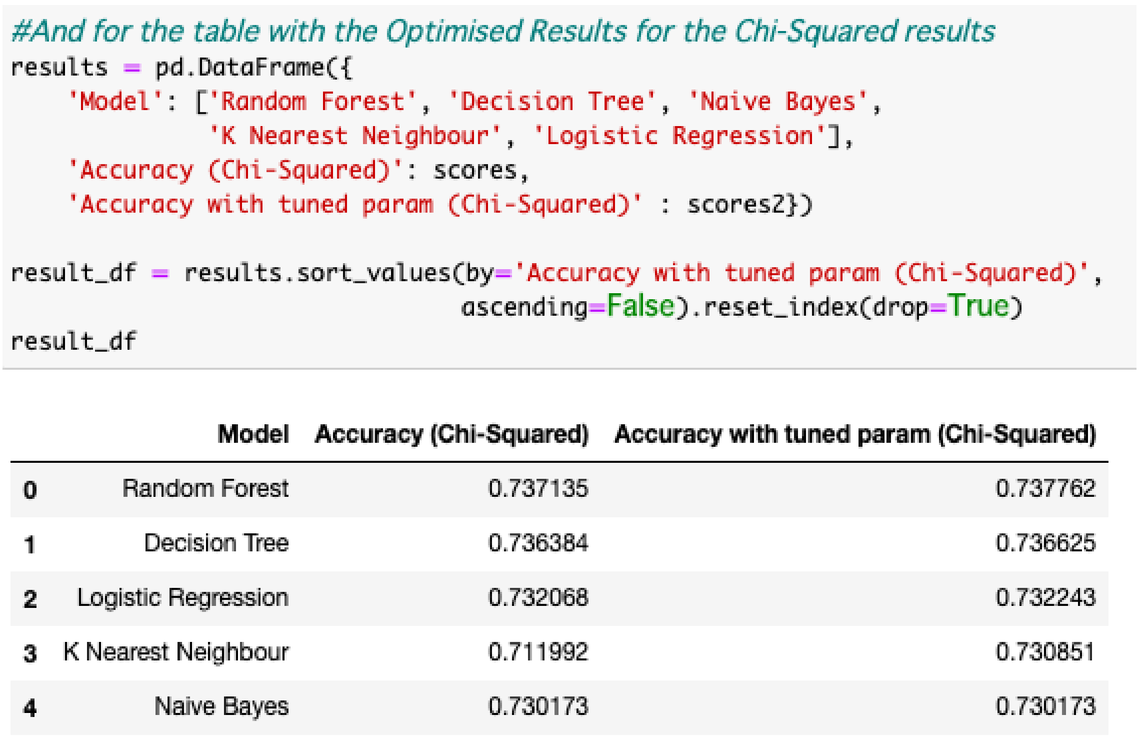


```
#After knowing the best parameters, we do the process again

forest = RandomForestClassifier(n_estimators = 200,
                                criterion = 'gini', random_state = 0)
tree = DecisionTreeClassifier(criterion = 'entropy',
                              splitter = 'random', random_state = 0)
bayes = GaussianNB()
k_nn = KNeighborsClassifier(n_neighbors = 18,
        n_jobs = 1, weights = 'uniform', metric = 'minkowski', p = 2)
log = LogisticRegression(C = 0.001, penalty = 'l2', random_state = 0)

models2 = [forest, tree, bayes, k_nn, log]
scores2 = []
```



```
for mod in models2:
    mod.fit(X_train, y_train.values.ravel())
    acc = cross_val_score(mod, X_train, y_train.values.ravel(),
                          scoring = "accuracy", cv = kfold, n_jobs = -1)
    scores2.append(acc.mean())
```

Stratified K-Fold Cross Validation with the optimised models

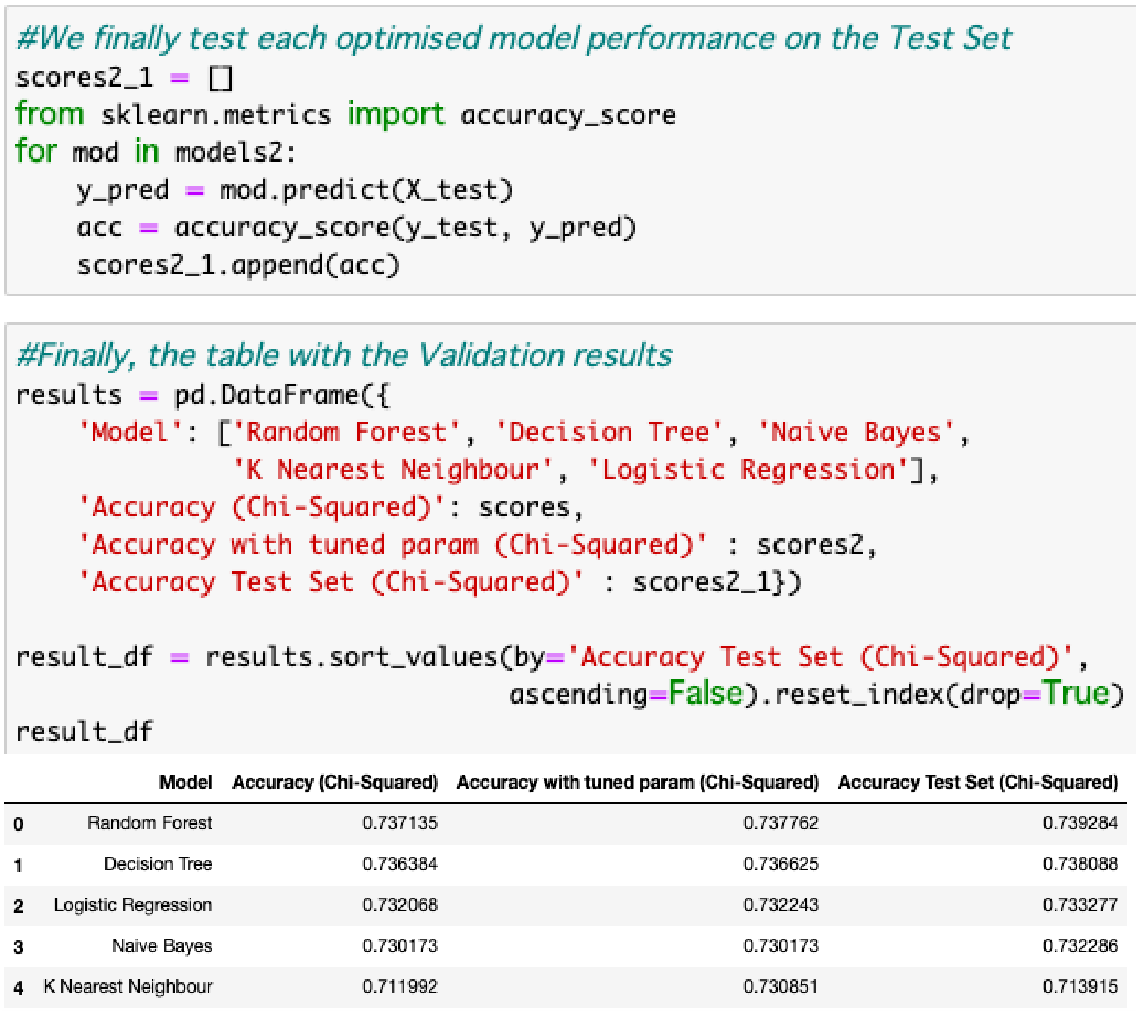


```python
#And for the table with the Optimised Results for the Chi-Squared results
results = pd.DataFrame({
    'Model': ['Random Forest', 'Decision Tree', 'Naive Bayes',
              'K Nearest Neighbour', 'Logistic Regression'],
    'Accuracy (Chi-Squared)': scores,
    'Accuracy with tuned param (Chi-Squared)' : scores2})

result_df = results.sort_values(by='Accuracy with tuned param (Chi-Squared)',
                                ascending=False).reset_index(drop=True)
result_df
```

	Model	Accuracy (Chi-Squared)	Accuracy with tuned param (Chi-Squared)
0	Random Forest	0.737135	0.737762
1	Decision Tree	0.736384	0.736625
2	Logistic Regression	0.732068	0.732243
3	K Nearest Neighbour	0.711992	0.730851
4	Naive Bayes	0.730173	0.730173

Accuracy with Tuned Parameters for each model

It is time to wrap up the first modelling. Once the Training is finished and the model has learned from the training data, all that is left is to check how does each model perform (in terms of accuracy) in the unbalanced Test Set (which, as said before, represents the reality of the database). This will be the final hurdle for the models.



```python
#We finally test each optimised model performance on the Test Set
scores2_1 = []
from sklearn.metrics import accuracy_score
for mod in models2:
    y_pred = mod.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    scores2_1.append(acc)
```

```python
#Finally, the table with the Validation results
results = pd.DataFrame({
    'Model': ['Random Forest', 'Decision Tree', 'Naive Bayes',
              'K Nearest Neighbour', 'Logistic Regression'],
    'Accuracy (Chi-Squared)': scores,
    'Accuracy with tuned param (Chi-Squared)' : scores2,
    'Accuracy Test Set (Chi-Squared)' : scores2_1})

result_df = results.sort_values(by='Accuracy Test Set (Chi-Squared)',
                                ascending=False).reset_index(drop=True)
result_df
```

	Model	Accuracy (Chi-Squared)	Accuracy with tuned param (Chi-Squared)	Accuracy Test Set (Chi-Squared)
0	Random Forest	0.737135	0.737762	0.739284
1	Decision Tree	0.736384	0.736625	0.738088
2	Logistic Regression	0.732068	0.732243	0.733277
3	Naive Bayes	0.730173	0.730173	0.732286
4	K Nearest Neighbour	0.711992	0.730851	0.713915

Predictions on the Test Set (Chi-Squared version)

The results seem to be fairly consistent, with a slightly better accuracy on the Test Set than on the Training Set, which shows the good performance of the trained models; specially the Random Forest method, with a 73,93% accuracy on the Test Set.

Now it is time to move into the second modelling, but since the modelling based on the features selected from the Feature Importance method follows the same prior steps, I will skip it and move into the final showdown: the comparison between the Accuracy results of both modelling versions on the Test Set.

At last, the results of each modelling are concatenated and we finally get closure for our modelling competition:

```
#And we put both tables together
results_chi = result_df[['Model', 'Accuracy Test Set (Chi-Squared)']]
results_feat = result_df_fi[['Accuracy Test Set (Feature Importance)']]
final_results = results_chi.join(results_feat)
final_results
```

	Model	Accuracy Test Set (Chi-Squared)	Accuracy Test Set (Feature Importance)
0	Random Forest	0.739284	0.735697
1	Decision Tree	0.738088	0.733773
2	Logistic Regression	0.733277	0.733569
3	Naive Bayes	0.732286	0.733364
4	K Nearest Neighbour	0.713915	0.733131

Comparison between the two versions of our modelling based on each Feature Selection

Results

The results from the comparison between both modelling versions show that the Random Forest Classification Model has given the best results, with a 73,93% of accuracy of its predictions in the Test Set.

Now, with this trained model and given a dataset with the features selected by the Chi-Squared method, it would be possible to predict with that much accuracy how likely an accident will end up in property damage or injury.

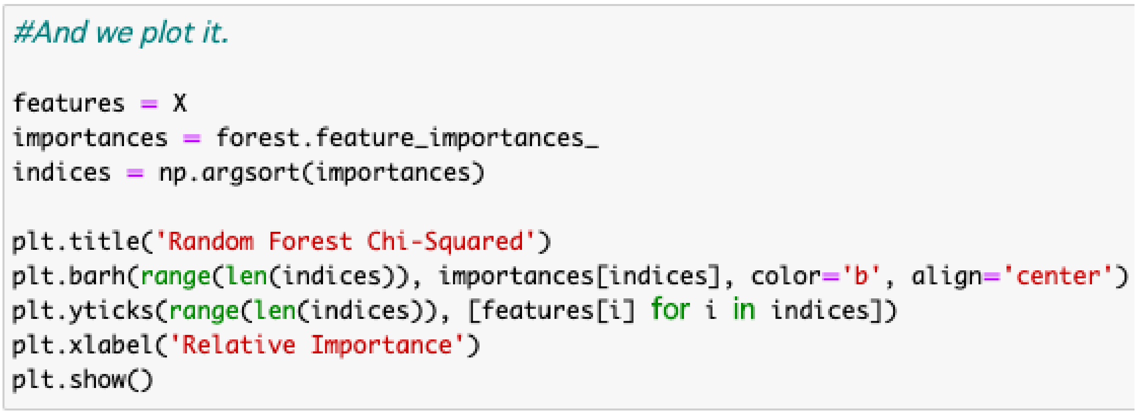
But, let's not forget that another objective of this project was to get a better understanding on the importance of the key factors involved.

The model with the best results (Random Forest) together with the features selected by the Chi-squared method will be the chosen combination to achieve this.

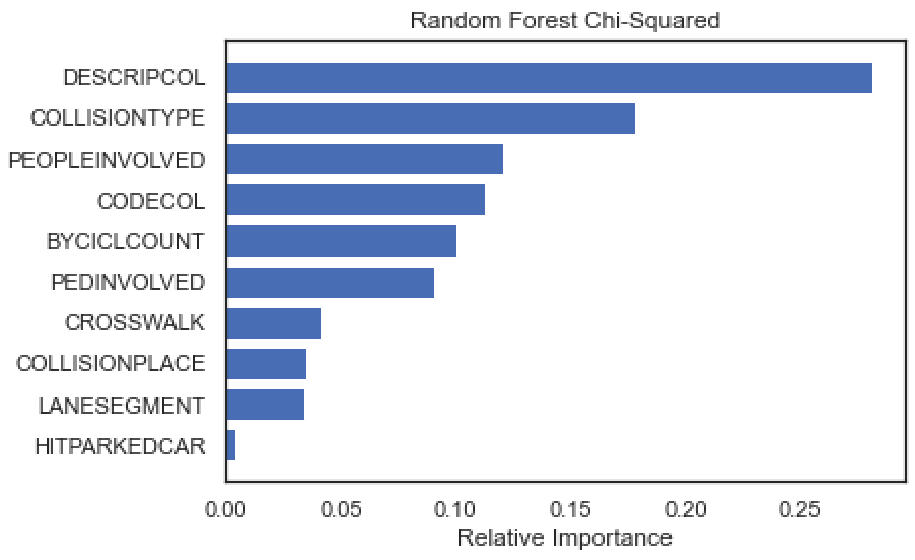
```
X = 'CROSSWALK','LANESEGMENT', 'DESCRIPCOL' ,'CODECOL' ,'PEDINVOLVED','BYCICLCOUNT', \
    'COLLISIONTYPE','COLLISIONPLACE','PEOPLEINVOLVED','HITPARKEDCAR'

forest = RandomForestClassifier(n_estimators = 200,
                                criterion = 'gini', random_state = 0)
forest.fit(X_train, y_train.values.ravel())
for name, importance in zip(X, forest.feature_importances_):
    print(name, "=", "{:.2f} %".format(importance*100))
```

```
CROSSWALK = 4.11 %
LANESEGMENT = 3.39 %
DESCRIPCOL = 28.13 %
CODECOL = 11.29 %
PEDINVOLVED = 9.11 %
BYCICLCOUNT = 10.03 %
COLLISIONTYPE = 17.85 %
COLLISIONPLACE = 3.54 %
PEOPLEINVOLVED = 12.12 %
HITPARKEDCAR = 0.42 %
```

Importance of each feature for the model



```
#And we plot it.

features = X
importances = forest.feature_importances_
indices = np.argsort(importances)

plt.title('Random Forest Chi-Squared')
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Visualisation of the importance of each feature for the model

Discussion

It is quite surprising that factors like the weather or light conditions don't play as big of a role as to predict how will the accident turn. Maybe it would have been more helpful if what we wanted to predict was if there was an accident or not; instead of assuming that the accident happened already.

Conclusion

With this project, we have built a model that can predict with a 73,93 % of accuracy how likely an accident will end only with property damage (type1) or with an injury (type2), given a set of features.

Specially interesting is the information regarding the features that are critical when it comes to predicting which type of accident will occur. A better design on the traffic infrastructure by reducing the number of parked cars, better accommodating bicycles and pedestrians or even improving the signposting of crosswalks could significantly ease the severity of road accidents.