Introduction

Aim :
To use machine learning techniques to predict the severity of highway accidents for the given data.

Application :
Traffic accidents are very common and cause 1.35 million deaths per year around the world. Due to the high number of lives affected, developing a system to predict worst accident prone areas could potentially save many lives. By recognising these accident prone areas of highways, design changes can be made and emergency services can be prepared.

Overview :
Through the course of this project we have worked on highway accident data to analyse the factors on which the severity of accidents depends upon and built machine learning models to predict the severity of accidents. For this purpose we have used various tools and algorithms which we will go through in the next sections.

Background

Dataset :
The used dataset is a countrywide traffic accident dataset, which covers 49 states of the United States. It consists of 4.2 million accidental records in 49 columns which capture details such as the location, time, severity of the accidents as well as various meteorological and traffic backdrops, and reports the severity of the corresponding accident. This was continuously collected from February 2016 till December 2020, using several data providers, including two APIs which provide streaming traffic event data.

About ML :
Traditionally, data analysis was trial and error-based, an approach that becomes impossible when data sets are large and heterogeneous. Machine Learning provides smart alternatives to analyzing vast volumes of data. By developing fast and efficient algorithms and data-driven models for real-time processing of data, Machine Learning can produce accurate results and analysis.

Traffic studies usually involve exceedingly large datasets that record several parameters multiple times over a single day. Over a significantly long period of time, these datasets can grow to extremely huge sizes that are virtually impossible to analyze manually. The dataset that we've used records approximately 10.5 lakh data points in a year. For each of these data points we have about 49 parameters like weather condition, time, traffic condition etc. Machine learning models with powerful processing are capable of analyzing bigger data to predict accidents or accident-prone areas.

Approach :
Our approach towards the problem was to first find out what kind of factors influence road accidents and their severity. We explored the data and visualized trends in it. We found out what variables are interrelated and removed them, leaving one, as it would just put excessive load on the models. We fixed the imbalance prevailing in the dataset. We tried a multitude of models on our final processed dataset to see how well they performed. We tried to reason out why certain models performed better than others, to assess which would work best in a real scenario.

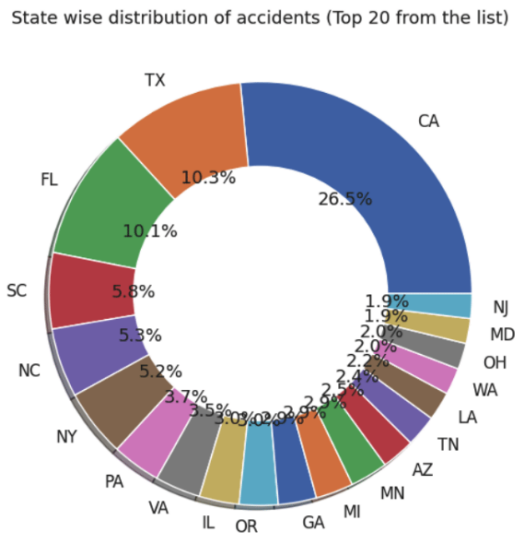
Why this topic :
Traffic accidents are very common and cause 1.35 million deaths per year around the world. Due to the high number of lives affected, developing a system to predict worst accident prone areas could potentially save many lives. If the model, at any time, classifies an area to be prone to a highly severe accident, required measures can be taken to improve road conditions and make it safer.
We have some prior experience in intersection studies and have learnt how prone to accidents they can get if the necessary road conditions are not maintained. In most places there are cases of the intersection rules related to vehicle capacity not being strictly enforced, and as a result, it must be ensured that other road parameters are kept under safety limits.

Exploratory Data Analysis

After we have selected the dataset, we need to analyse, clean and transform it into the desired format. The reason behind this step is that it is possible that the data set contains the constraints which are not needed by the prediction system and including them makes the system complicated and may extend the processing time. Another reason behind data cleaning is the dataset may contain null value and garbage values too. So, the analysis part will help to identify all that so that the data can be further pre-processed accordingly.

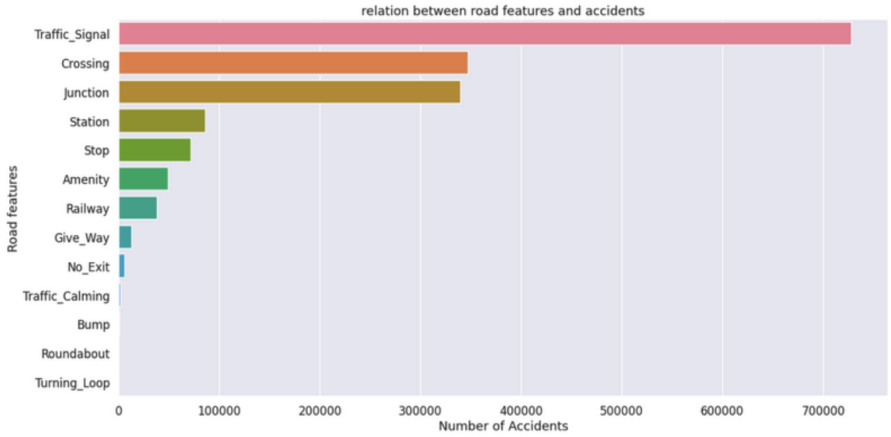
Statewise Distribution:

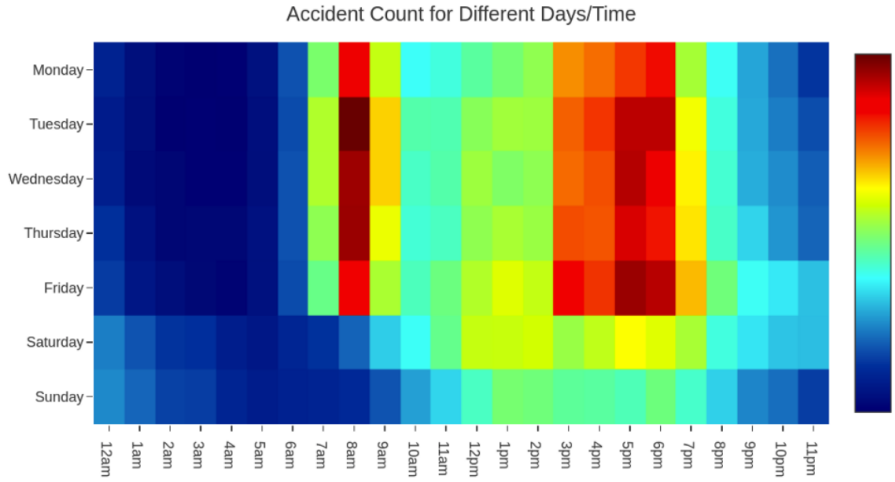
A large number of accidents occur in California, Texas & Florida as shown.



Relation with road features and Daytime:

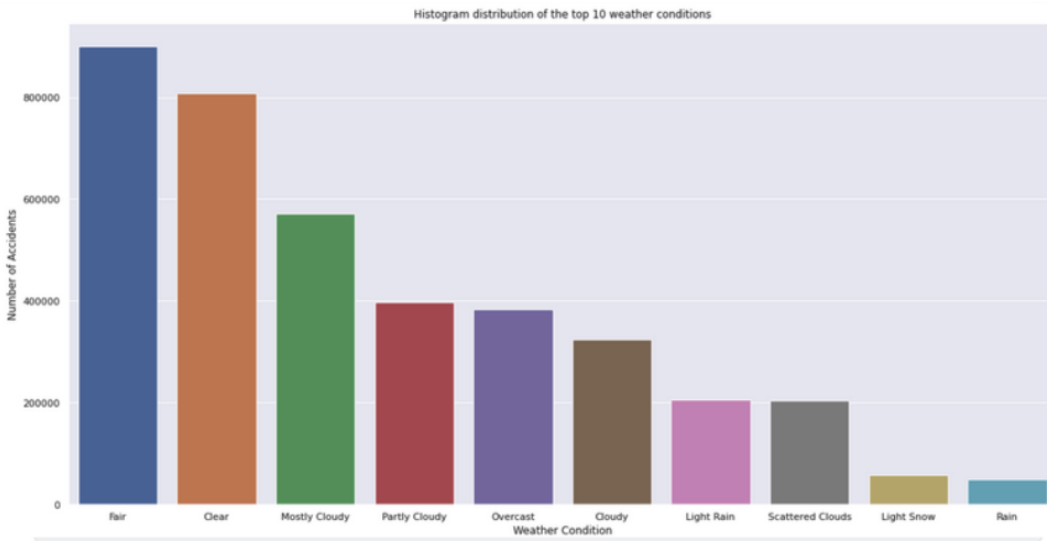
Most accidents are recorded during traffic signals followed by crossing and Junction.





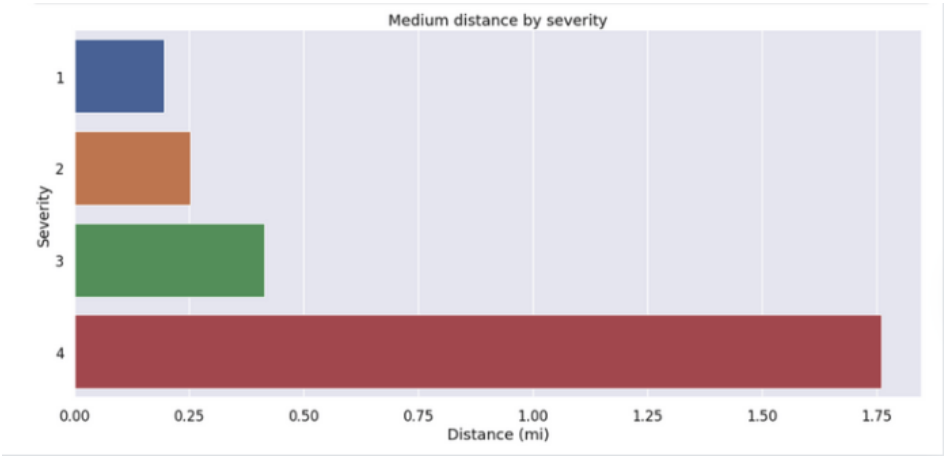
Weather Conditions :

Most accidents occured in clear/good weather conditions. This is pretty counterintuitive because generally it is believed that accidents would be more prone during cloudy/bad weather but maybe due to much more consciousness among people during bad weather, the data reports maximum accidents happening in clean weather.



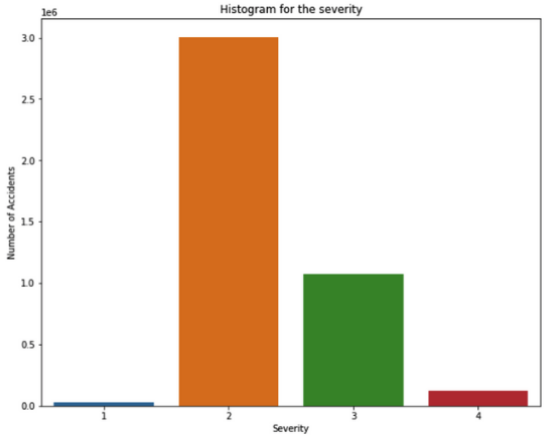
Distance v/s Severity :

Severity of accident increases with the length of the road

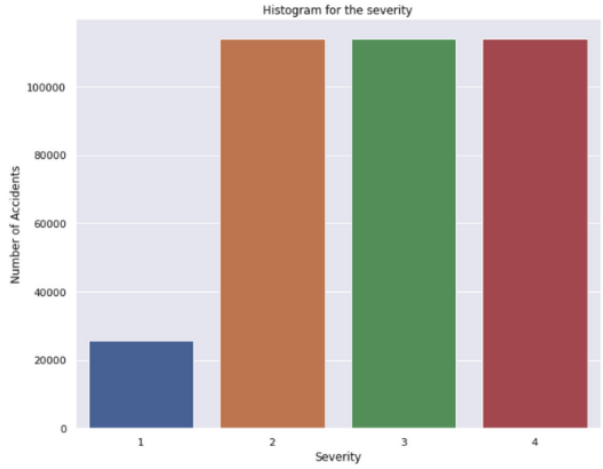


Imbalanced Data :

As we can see the data is unbalanced with less than 1% data with severity 1.

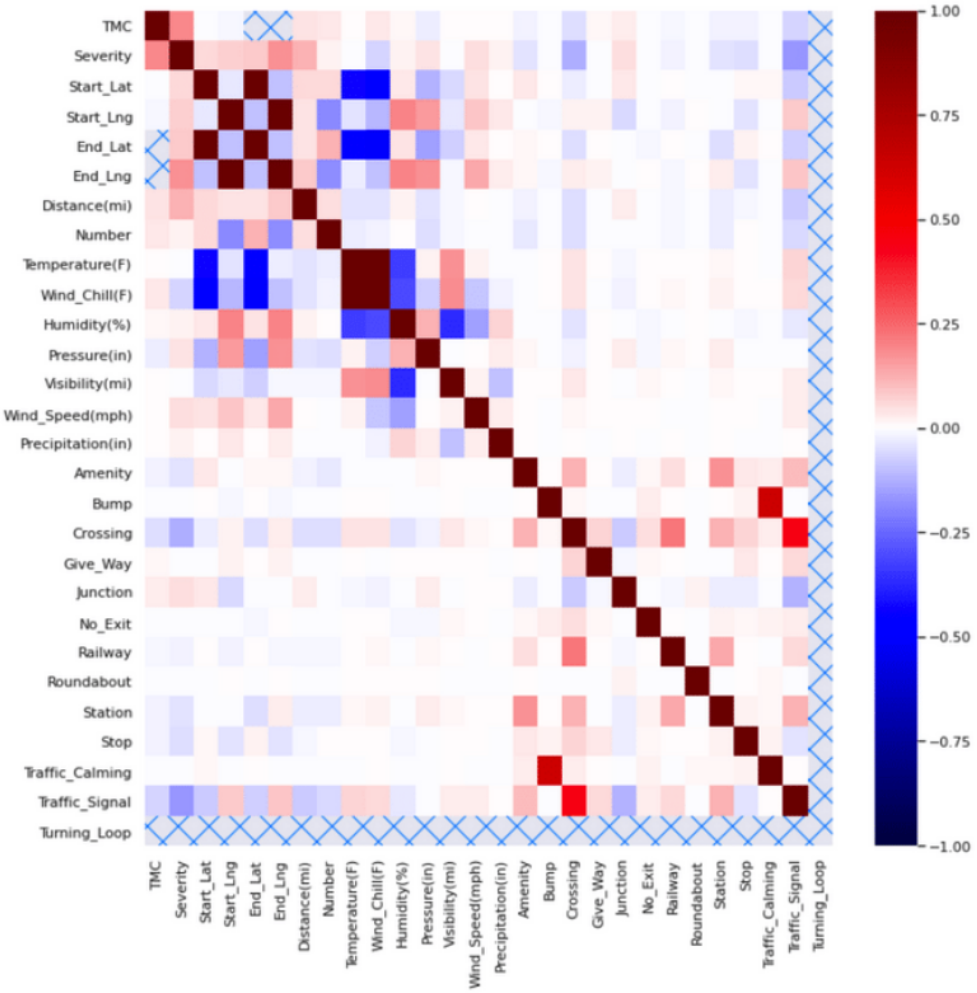


To deal with this we have to under-sample the data of severity 2 & 3 equal to that of severity 4. Then, we can use SWOTE to over-sample data with severity 1.



Correlation between Features :

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.



Data Preprocessing

After the data is cleaned and transformed it's ready to process further. We divide the whole dataset into the two parts(70-30). The larger portion of the data is for the processing. The algorithm is applied on that part of data which helps the algorithm to learn on its own and make predictions for the future data or the unknown data.

1. **Feature Selection :**
   Working with a large number of features may affect the performance because training time increases exponentially with the number of features. Even, it has also the risk of overfitting with the increasing number of features. So, for getting a more accurate prediction, feature selection is a critical factor.

2. **Feature Addition :**
   From Start_Time feature- year, month, day, date, time were extracted.

3. **Dropping Duplicates :**
   Dropping duplicated data reduces the data by 200,000 rows which would significantly minimize the processing time.

4. **Dropping few columns**

```
df.drop(["ID", "Source", "TMC", "Country", "Description","Start_Lat",'Start_Lng", "End_Lat", "End_Lng", "Wind_Chill(F)",
         "Traffic_Signal", "Traffic_Calming", "Turning_Loop", "Start_Time", "End_Time", "Street", "County", "State", "Zipcode
df.head()
```

[13]:

	Severity	Distance(mi)	Number	Side	City	Timezone	Airport_Code	Temperature(F)	Humidity(%)	Pressure(in)	...	Sunrise_Sunset	Civil_Twilight	Nautical_Twilight
0	3	0.01	NaN	R	Dayton	US/Eastern	KFFO	36.9	91.0	29.68	...	Night	Night	Nigh
1	2	0.01	2584.0	L	Reynoldsburg	US/Eastern	KCMH	37.9	100.0	29.65	...	Night	Night	Nigh
2	2	0.01	NaN	R	Williamsburg	US/Eastern	KI69	36.0	100.0	29.67	...	Night	Night	Da
3	3	0.01	NaN	R	Dayton	US/Eastern	KDAY	35.1	96.0	29.64	...	Night	Day	Da
4	2	0.01	NaN	R	Dayton	US/Eastern	KMGY	36.0	89.0	29.65	...	Day	Day	Da

5 rows × 35 columns

   Some of the unnecessary columns were dropped.

5. **Handling missing values :**
   We will investigate each column with total missing values. We will not be imputing any mean or median value since the dataset is big enough to perform analysis.

   → For side 1 data point with missing value is dropped

   → For pressure and visibility missing data i represented as 0

   → All 0 values are dropped

   → Numbers and precipitation have many missing entries. Hence, we drop these 2 features.

[17]:

	Pressure(in)	Visibility(mi)
count	3969974.000	3948635.000
mean	29.704	9.114
std	0.852	2.830
min	0.000	0.000
25%	29.650	10.000
50%	29.930	10.000
75%	30.080	10.000
max	58.040	140.000

As the minimum value is 0.00. There must be some missing data which is replaced with zeros.

Dropping data with zeros in Pressure and Visibility

```
df = df[df["Pressure(in)"] != 0]
df = df[df["Visibility(mi)"] != 0]
df[["Pressure(in)", "Visibility(mi)"]].describe().round(3)
```

6. **Handling weather conditions data :**

   →Weather conditions have many unique values, therefore number of unique values is reduced

   → The same is done for wind direction.

7. **Checking feature variance :**

   → Checking the feature variance and removing the features with low variance and very high variance to avoid overfitting or underfitting the data points

   → The variance of pressure is very less, but in nature pressure never varies much. Hence, we will not drop the feature.

8. **Feature scaling :**

   There are few columns that we will standardize, so it would not affect negatively on our machine learning algorithms. The continuous data was normalized.

```
scaler = MinMaxScaler()
features = ['Temperature(F)','Distance(mi)','Humidity(%)','Pressure(in)','Visibility(mi)','Wind_Speed(mph)',
            'Year', 'Month','Weekday','Day','Hour']
df[features] = scaler.fit_transform(df[features])
df.describe().round(3)
```

[27]:

	Severity	Distance(mi)	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Speed(mph)	Year	Month	Weekday	Day	Hour	
count	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	367661.000	3676
mean	2.860	0.004	0.553	0.646	0.948	0.076	0.012	0.600	0.522	0.429	0.496	0.532	
std	0.938	0.014	0.103	0.237	0.034	0.024	0.007	0.335	0.313	0.303	0.290	0.246	
min	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
25%	2.000	0.000	0.485	0.480	0.947	0.083	0.007	0.250	0.273	0.167	0.233	0.348	
50%	3.000	0.000	0.561	0.673	0.958	0.083	0.011	0.750	0.545	0.500	0.500	0.565	
75%	4.000	0.003	0.631	0.847	0.963	0.083	0.015	1.000	0.818	0.667	0.733	0.739	
max	4.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	

9. Feature Encoding :

→ Categorical features were encoded, this is done to reduce memory usage

→ Binary encoding Airport_Code and City and One-Hot encoding the rest

10. Handling unbalanced data :

→ As seen earlier, the original data is very unbalanced.

→ Used SMOTE algorithm to oversample the class with the least number of entries to match that with the number of entries of the second lowest class. The entries of the other two classes have been reduced to match these classes.

```
size = len(df[df["Severity"]==4].index)
X = pd.DataFrame()
for i in range(1,5):
    S = df[df["Severity"]==i]
    if i == 1:
        new_size = len(df[df["Severity"]==i].index)
        X = X.append(S.sample(new_size, random_state=21))
    else:
        X = X.append(S.sample(size, random_state=21))
df = X
severity_counts = df["Severity"].value_counts()
plt.figure(figsize=(10, 8))
plt.title("Histogram for the severity")
sns.barplot(x = severity_counts.index,y =  severity_counts.values)
plt.xlabel("Severity")
plt.ylabel("Number of Accidents")
plt.show()
del X
del S

severity_counts = df["Severity"].value_counts()
print("\n",severity_counts)
```

Models

Multinomial Logistic Regression

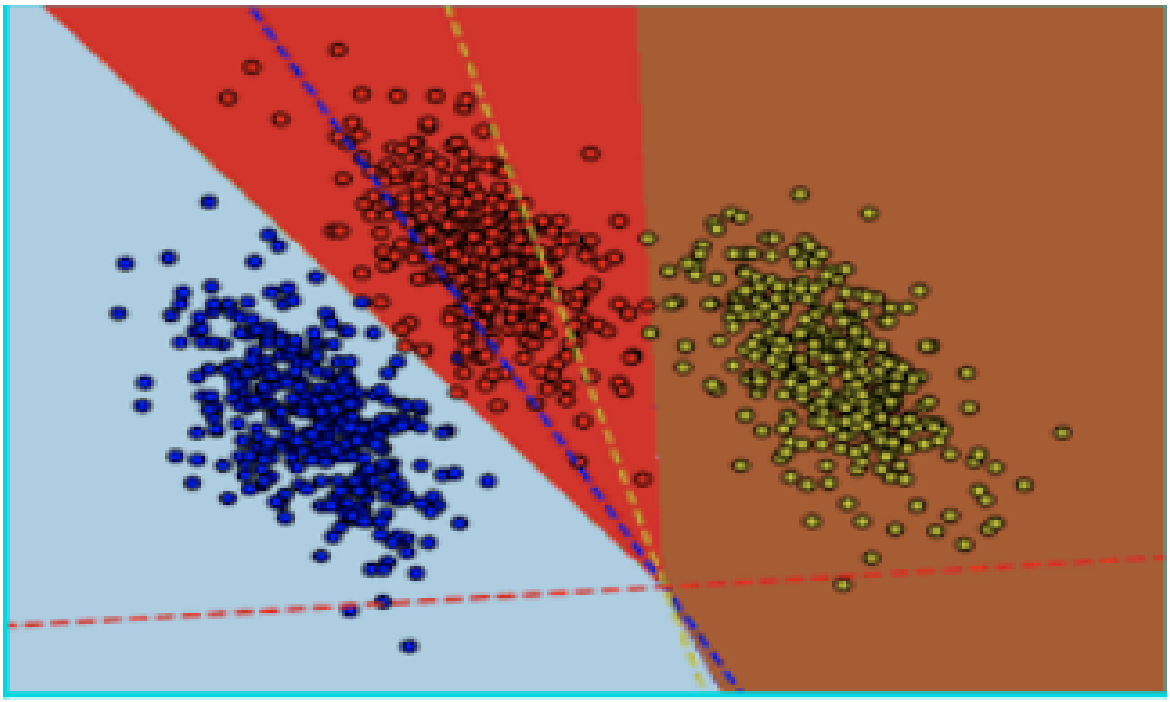
Logistic regression is a fundamental classification technique. Classification is an area of supervised machine learning that tries to predict which class or category some entity belongs to, based on its features. It belongs to the group of linear classifiers and is similar to polynomial and linear regression. It is fast and relatively uncomplicated.

Logistic regression models can be classified into two types:

1. Binary logistic regression - In this the dependent variable has exactly two finite outputs to choose from like 0 and 1, True and False, Positive or Negative.
2. Multinomial logistic regression - In this the dependent variable has three or more finite outputs to choose from.

```
#Starting the timer
start_time = time.perf_counter()

lr = LogisticRegression(n_jobs=-1)
params = {"solver": ["saga"], "penalty" : ["l1"]} #Params to check with
grid_search = GridSearchCV(lr, params, n_jobs=-1, verbose=5, cv=5, return_train_score = True)
grid_search.fit(X, y)

print("Best parameters scores:")
print(grid_search.best_params_)
print("Mean Train Score: ", grid_search.cv_results_['mean_train_score'][0])
print("Mean Validation Score:", grid_search.best_score_) #Mean cross-validation score of best estimator Estimated: 0.63046865
print("Test Score: ", grid_search.best_estimator_.score(X_test,y_test)) # Estimated: 0.4016917344357712

#Ending the timer
end_time = time.perf_counter()
total_time = end_time-start_time

print("It took {} secs for completing Logistic Regression training.".format(total_time))
```

This model is used to predict probabilities and classify samples using continuous data points. There are multiple thresholds in the graph that correspond to the dependent value.
It is a particular solution to classification problems that use a linear combination of the observed features and specific parameters to estimate the probability of each particular value of the dependent variable. The best values of the parameters for a given problem are usually determined from the training dataset.



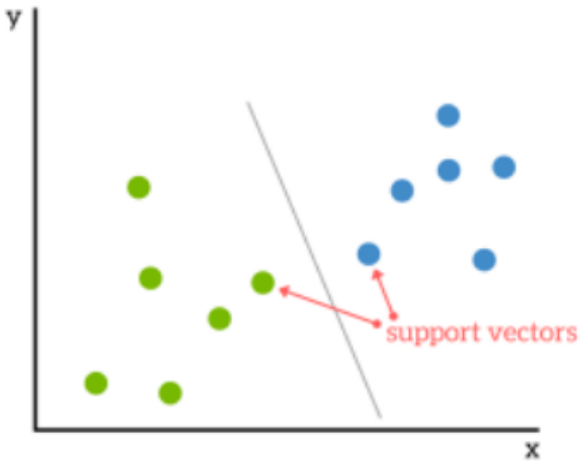
On high dimensional datasets, it may lead to the model being over-fit on the training set and may not be able to predict accurate results on the test set.

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Best parameters scores:
{'penalty': 'l1', 'solver': 'saga'}
Mean Train Score:  0.6328269513360016
Mean Validation Score: 0.6312523299582107
Test Score:  0.43150107433296164
It took 516.6899143929995 secs for completing Logistic Regression training.
```

Support Vector Machine

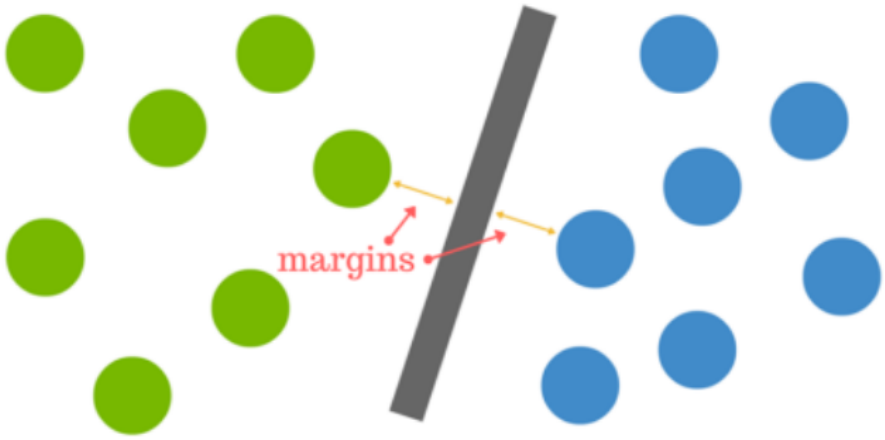
A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems. A support vector machine is a maximum margin classifier that attempts to find a separating hyperplane between the classes of data and maximize the distance between the hyperplane and the data points closest to it.

```
df = pd.DataFrame(y, columns = ['Severity'])
X_svm = pd.concat([X, df], axis = 1)
sample = X_svm.sample(50_000, random_state=21)
y_sample = sample["Severity"]
X_sample = sample.drop("Severity", axis=1)

svc = svm.SVC(verbose=5, gamma='auto')
params = {'C': [1.0], 'degree':[2], 'kernel': ['rbf']} #Best params
grid_search = GridSearchCV(svc, params, n_jobs=-1, verbose=5, cv=5, return_train_score = True)
grid_search.fit(X_sample, y_sample)

print("Best parameters scores:")
print(grid_search.best_params_)
print("Mean Train Score: ", grid_search.cv_results_['mean_train_score'][0])
print("Mean Validation score:", grid_search.best_score_) #Mean cross-validation score of best estimator Expected: 0.6083000000000001
print("Test score: ", grid_search.best_estimator_.score(X_test,y_test))#Mean test score of best estimator Expected: 0.4185546821878315

#Ending the timer
end_time = time.perf_counter()
total_time = end_time-start_time

print("It took {} secs for completing SVM training.".format(total_time))
```



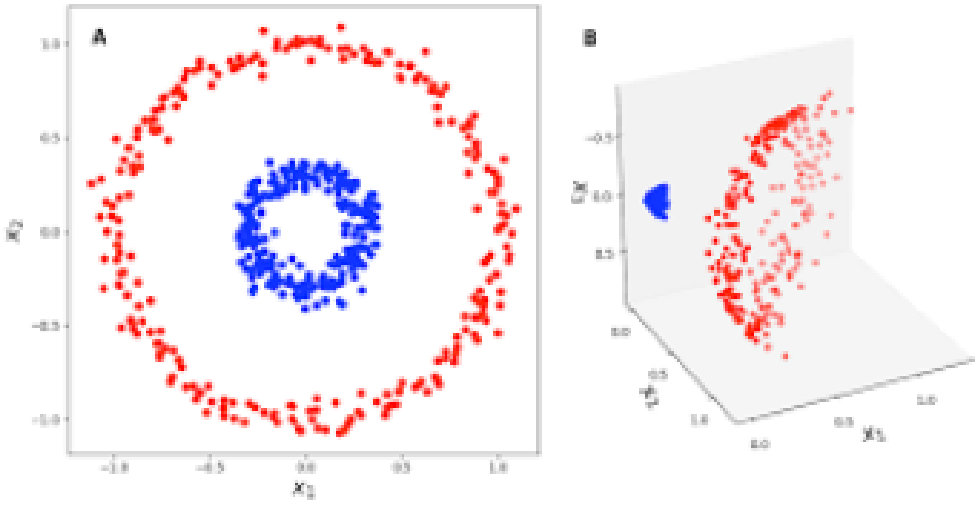
Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.For a classification task with only two features one can think of a hyperplane as a line that linearly separates and classifies a set of data. Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it.

So when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.



The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly.

The Radial basis function (RBF) kernel or Gaussian kernel is used in cases where a clear separating boundary cannot be found between the classes.
The kernel transforms the data into higher dimensions as shown, and then attempts to find a separating hyperplane. The kernel function is used to find the relation between two given data points.



SVM algorithm has a high efficiency as it uses a subset of training points and works well on smaller and cleaner datasets.

It is not suitable for large datasets as the training time with SVM can be significantly high. It is sensitive to outliers and can lead to misclassifications. It is less effective on noisier datasets with overlapping classes.

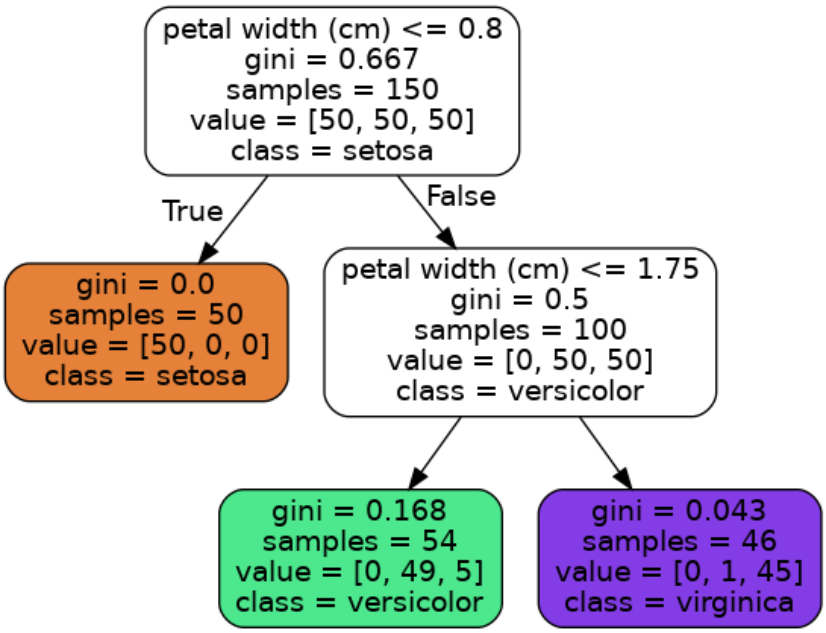
```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[LibSVM]Best parameters scores:
{'C': 1.0, 'degree': 2, 'kernel': 'rbf'}
Mean Train Score:  0.6148450000000001
Mean Validation score: 0.6083000000000001
Test score:  0.4195066227867381
It took 2118.8661723679998 secs for completing SVM training.
```

Random Forest Classifier

Random Forest is an ensemble of Decision Trees, generally trained via bagging method (or sometimes pasting). So in order to understand random forest let's take a look at Decision Trees.

Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multi output tasks. They are powerful algorithms, capable of fitting complex datasets. We will use the iris data set available from the sklearn library to visualize the working of the Decision Tree Algorithm.

The data set consists of 150 records divided in 3 classes namely “setosa”, “versicolor” and “virginica”.



Consider the above image, the coloured nodes are called as leaves and the ones which split in two are called as nodes. Each level of the tree is called as depth starting with depth 0. Hence, in the above diagram the depth of the tree is 3. At each node the algorithm decides a feature and its limit and classifies the tree to a new depth. For example, for petal_width ≤ 0.8 the tree splits into two depending on the records that satisfy the above condition and the one's which do not satisfy. The sample's parameter in the leaf tells the number of samples qualifying/not qualifying the criterion. The value parameter shows the number of entries of each class present in the sample. For example, the orange node shows that 50 records qualify the criteria that the pedal width ≤ 0.8 and the value parameter shows that all the 50 entries belong to class 1. The criterion for choosing the classification feature (i.e. the value based on which the nodes are made) is called as "gini". The formula for calculating gini is :

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

For example, the green leaf has a gini value of 0.168 which can be calculated as follows:

$$1 - \left(\frac{0}{54}\right)^2 - \left(\frac{49}{54}\right)^2 - \left(\frac{5}{54}\right)^2 = 0.168$$

In this manner the nodes are formed based on the criterion which gives the lowest "gini" value. This is the way that a decision tree is used for classification problems. A random forest is formed using a bunch of such decision trees and the classes are predicted based on the average probability given by each tree.

Coming back to our model, we have used the RandomForsetClassifier function from sk-learn library. We have also hypertuned the parameters using GridSearchCV which breaks the data set into various training and validation sets and tries various combinations of hyperparameters on them returning the best parameters and the training and validation accuracy.

```
#Starting the timer
start_time = time.perf_counter()

rfc = RandomForestClassifier(n_jobs=-1)
params = [{"n_estimators": [500], "max_depth": [35], "criterion" : ["gini"]}]
grid_search = GridSearchCV(rfc, params, n_jobs=-1, verbose=5, cv=2, return_train_score = True)
grid_search.fit(X, y)

print("Best parameters scores:")
print(grid_search.best_params_)
print("Mean Train Score:", grid_search.cv_results_['mean_train_score'][0])
print("Mean Validation score:", grid_search.best_score_) #Mean cross-validation score of best estimator Expected: 0.7847685987422364
print("Test score: ", grid_search.best_estimator_.score(X_test,y_test))#Expected: 0.6068485326515626

#Ending the timer
end_time = time.perf_counter()
total_time = end_time-start_time

print("It took {} secs for completing Decision Tree training.".format(total_time))
```

As it can be seen, the “params” variable shows the best hyperparameters found out after hyperparameter tuning.

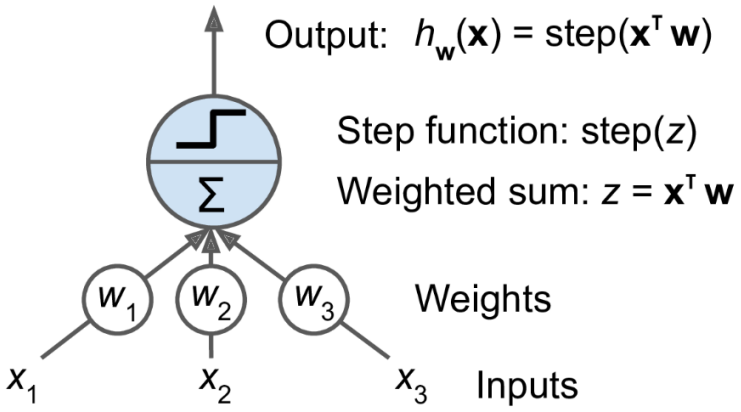
```
Fitting 2 folds for each of 1 candidates, totalling 2 fits
Best parameters scores:
{'criterion': 'gini', 'max_depth': 35, 'n_estimators': 500}
Mean Train Score: 0.9995010805864372
Mean Validation score: 0.7848124063005004
Test score:  0.5846547175456251
It took 646.6605703770001 secs for completing Decision Tree training.
```

The above image shows the Training score, Validation score and Test score of the model. It can be seen that the model has overfit the data set. The model can be further improved by properly regularizing it.

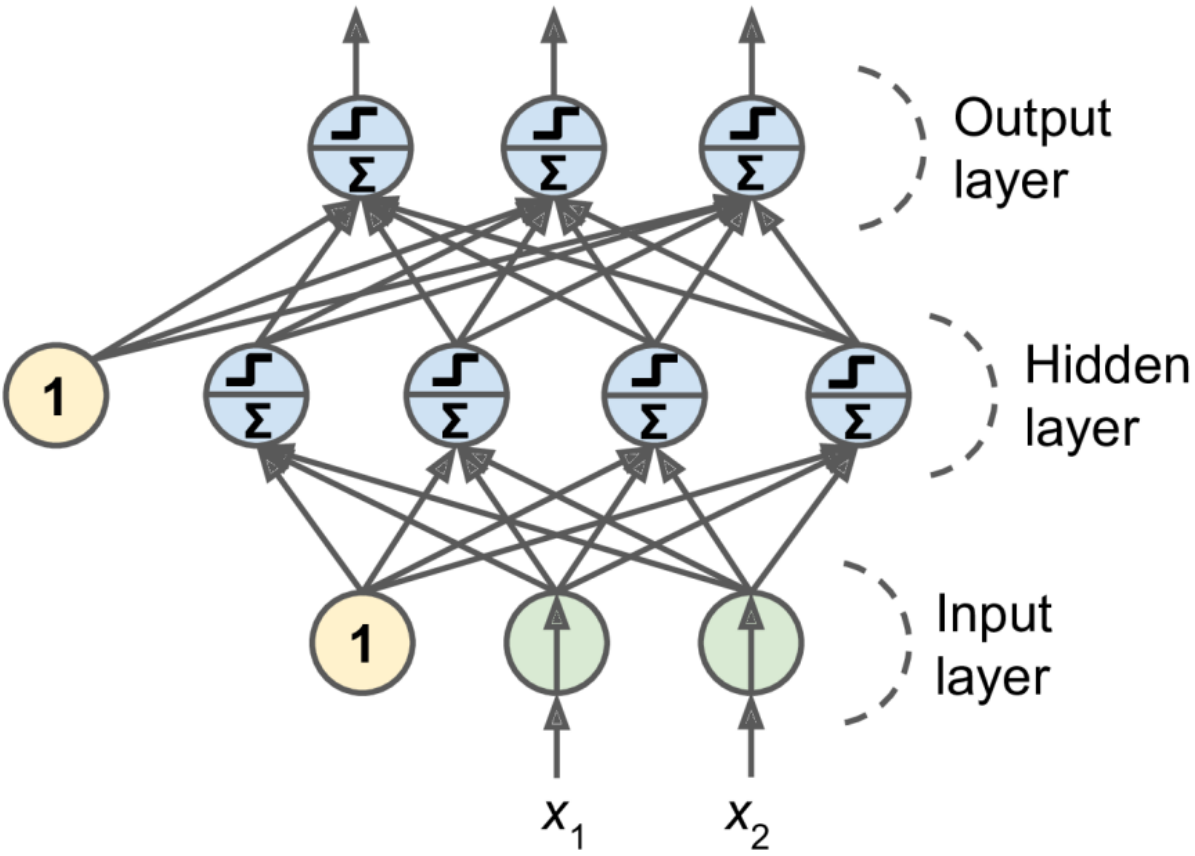
Multi-Layer Perceptron

The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron called a threshold logic unit (TLU), or sometimes a linear threshold unit (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs (z = w 1 x 1 + w 2 x 2 + + w n x n = x ⊺ w), then applies a step function to that sum and outputs the result:

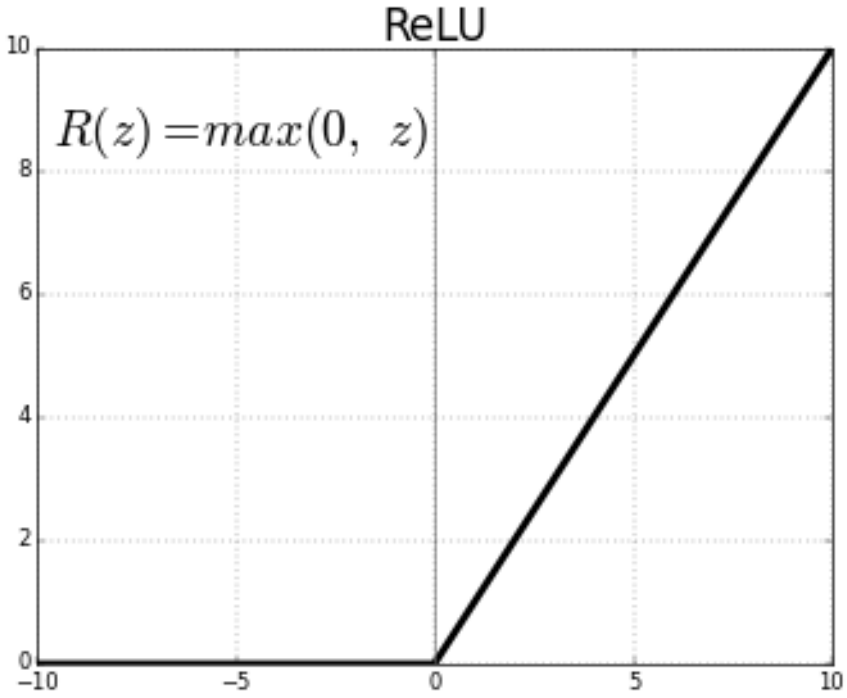
$$h_w(x) = step(z), where\ z = x^T w$$



A Multilayer Perceptron (MLP) is composed of one (passthrough) input layer, one or more layers of TLUs, called hidden layers, and one final layer of TLUs called the output layer. The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers.



The model used is made up of three hidden layers of size 32, 64 and 32 units respectively. The maximum iteration limit is set to 350 and the optimizer used is “adam” which is similar to Stochastic Gradient Descent but the learning rate is customized for every parameter.



```
#Starting the timer
start_time = time.perf_counter()

mlp = MLPClassifier(random_state=42, verbose=False)
params = [{"hidden_layer_sizes": [(32, 64, 32)], "max_iter": [350], "solver": ["adam"], "activation":["relu"]}]
grid_search = GridSearchCV(mlp, params, n_jobs=-1, verbose=5, cv=2, return_train_score = True)
grid_search.fit(X, y)

print("Best parameters scores:")
print(grid_search.best_params_)
print("Mean Train Score:", grid_search.cv_results_['mean_train_score'][0])
print("Mean Validation score:", grid_search.best_score_) #Mean cross-validation score of best estimator Expected: 0.7486176281614454
print("Test score: ", grid_search.best_estimator_.score(X_test,y_test)) #Expected: 0.544822722442277

#Ending the timer
end_time = time.perf_counter()
total_time = end_time-start_time

print("It took {} secs for completing Multilayer Perceptron Training.".format(total_time))
```

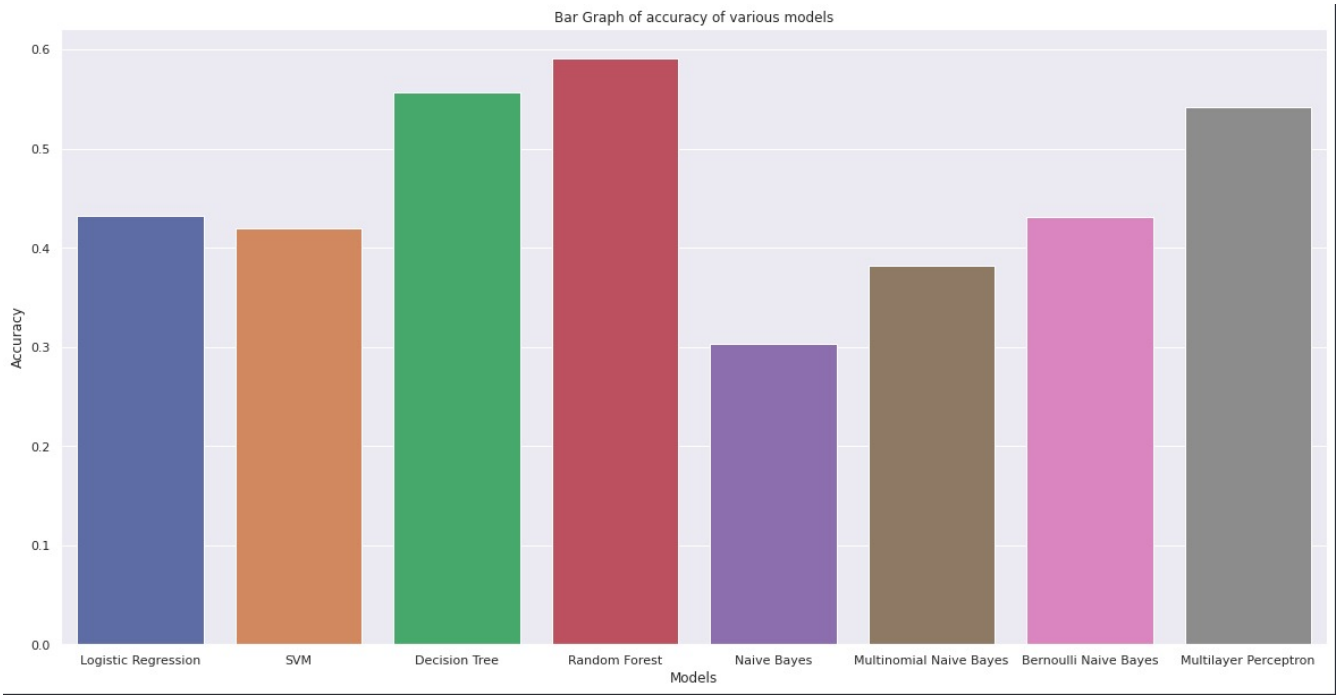
From the above image we can see that, as done with RandomForestClassifier GridSearchCV has been used for hyperparameter tuning in the Multilayer Perceptron model also.

```
Fitting 2 folds for each of 1 candidates, totalling 2 fits
Best parameters scores:
{'activation': 'relu', 'hidden_layer_sizes': (32, 64, 32), 'max_iter': 350, 'solver': 'ada
m'}
Mean Train Score: 0.7707964214092404
Mean Validation score: 0.7506911859192773
Test score:  0.5411646313270052
It took 6399.238747433001 secs for completing Multilayer Perceptron Training.
```

The above image shows the results of the model. On trying different units and layers of hidden layers,the above parameters are the best for the dataset. From the training and test score we can see that the model has not overfit nor under fit the data but has failed to perform likewise on the test data set. The poor performance of the model on the test data set can be due to oversampling of the data set we performed in the Data Preprocessing step.

Result

Out of the 4 models discussed in detail the best model is Random forest model followed by Decision tree and multilayer perceptron.



Summary

For this course project we chose the topic of accident prediction as we identified highway engineering as a field where use of machine learning techniques would give great insights and generate results.
We used a countrywide traffic accident survey dataset of the US which had 4.2 million data points and 49 columns.

Various python libraries were used to perform EDA and analyse and understand trends in data. The data was further cleaned to remove unrequired columns, feature scaling was done and data imbalance was handled. After this ML models were generated.
The following machine learning models gave good results for the data-

- Multinomial logistic regression
- SVM
- Random forest classifier
- Multi-layer perceptron

Random forest gave the best result with a validation score of 0.785 and a test score of 0.585.

Conclusion:

Using this model of acceptable (but improvable) results we can predict the accident prone areas of a highway. By identifying these zones we can correct the design of the highway. The emergency medical services can be at zones with most severe accidents helping reduce fatalities. Law enforcement should also be made these zones so as to ensure all vehicles follow traffic rules.

We would also like to thank Prof. Benny Rafael for giving us the opportunity of doing a project which expanded our knowledge of the applications of Machine learning techniques in solving problems in civil engineering. By working on the project we were able to learn how to build ML models, using python libraries and learn how to apply machine learning algorithms. During the course of this project we also improved our interpersonal skills of communication and teamwork.

References

Aurélien Géron - Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow_ Concepts, Tools, and Techniques to Build Intelligent Systems(2019)