Automobile Insurance Fraud Prediction Using Machine Learning

Hey everyone, in this article I will be discussing about automobile insurance fraud prediction using few of the machine learning models via python and its libraries.

Problem Description:

Automobile Insurance industry is one of the many complicated industries where there is huge amount of cash flowing in and out, and when there is an existence of whopping amount of money flowing there is always an existence of fraud. Crafty people try to find loop holes in the policies of the insurance to get away with hefty sum of money. In this article we will analyse the automobile insured data and predict if he/she has committed the fraud or not via various factors effecting the outcome.

Dataset:

The first foremost important step in any machine learning is Data Collection. We make use of the collected data and analysed the data and train it using various machine learning model to predict the outcome.

In this machine learning project, I will be making use of the dataset available on github. [Click here] to get the raw csv file URL.

Note: I will be compiling and running the code on Jupyter notebook, there are various platform to perform it though. The text shaded in the black is representation of line of code in this article.

Importing Dependencies:

```
#importing dependencies

import pandas as pd

import numpy as np

from numpy import mean

from numpy import std

import matplotlib.pyplot as plt

import seaborn as sns

from scipy.stats import skew

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestClassifier

from sklearn.linear_model import LogisticRegressio

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import roc_curve, auc, roc_auc_score

from sklearn.metrics import confusion_matrix

from sklearn.metrics import precision_score, recall_score, f1_score, classification_report,
accuracy_score
```

```
from sklearn.datasets import make_classificatio
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
from collections import Counter
from sklearn import metrics
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import LabelEncoder
from scipy.stats import boxcox
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import GridSearchCV as gs
import pickle
```

Dependencies are software components which help us mould the raw data into our desire interest.

Above listed code are the dependencies used in this project

Data pre-processing and analysis:

Loading the dataset using pandas.(the below code helps you load the data)

```
url='https://raw.githubusercontent.com/dsrscientist/Data-Science-ML-Capstone-Projects/master/Automobile_insurance_fraud.csv'
dataset=pd.read_csv(url)
```

The shape of the dataset is 1000 rows and 40 columns

```
dataset.info()
```

```
#   Column                       Non-Null Count  Dtype
--- ------                       --------------  -----
0   months_as_customer           1000 non-null   int64
1   age                          1000 non-null   int64
2   policy_number                1000 non-null   int64
3   policy_bind_date             1000 non-null   object
4   policy_state                 1000 non-null   object
5   policy_csl                   1000 non-null   object

 6   policy_deductable            1000 non-null   int64
 7   policy_annual_premium        1000 non-null   float64
 8   umbrella_limit               1000 non-null   int64
 9   insured_zip                  1000 non-null   int64
 10  insured_sex                  1000 non-null   object
 11  insured_education_level      1000 non-null   object
 12  insured_occupation           1000 non-null   object
 13  insured_hobbies              1000 non-null   object
 14  insured_relationship         1000 non-null   object
 15  capital-gains                1000 non-null   int64
 16  capital-loss                 1000 non-null   int64
 17  incident_date                1000 non-null   object
 18  incident_type                1000 non-null   object
 19  collision_type               1000 non-null   object
 20  incident_severity            1000 non-null   object
 21  authorities_contacted        1000 non-null   object
 22  incident_state               1000 non-null   object
 23  incident_city                1000 non-null   object
 24  incident_location            1000 non-null   object
 25  incident_hour_of_the_day     1000 non-null   int64
 26  number_of_vehicles_involved  1000 non-null   int64
 27  property_damage              1000 non-null   object
 28  bodily_injuries              1000 non-null   int64
 29  witnesses                    1000 non-null   int64
 30  police_report_available      1000 non-null   object
 31  total_claim_amount           1000 non-null   int64
 32  injury_claim                 1000 non-null   int64
 33  property_claim               1000 non-null   int64
 34  vehicle_claim                1000 non-null   int64
 35  auto_make                    1000 non-null   object
36  auto_model                   1000 non-null   object
37  auto_year                    1000 non-null   int64
38  fraud_reported               1000 non-null   object
39  _c39                         0 non-null      float64
```

info() helps in fetching the dataset columns number non null values and datatype of each columns , to feed the data to a machine learning model it is absolutely necessary that our dataset is free of null values i.e. missing values and all the data is in numeric form.

Taking a look at the dataset info except for 40th column, no other column has null value.

As 40th column contains all null values it necessary to drop off the column.

```
dataset.drop(columns=['_c39'],inplace=True)
```

Taking a look at the dataset

```
dataset.head()
```

	months_as_customer	age	policy_number	policy_bind_date	policy_state	policy_csl	policy_deductable	policy_annual_premium	umbrella_limit	insured_zip
0	328	48	521585	17-10-2014	OH	250/500	1000	1406.91	0	466132
1	228	42	342868	27-06-2006	IN	250/500	2000	1197.22	5000000	468176
2	134	29	687698	06-09-2000	OH	100/300	2000	1413.14	5000000	430632
3	256	41	227811	25-05-1990	IL	250/500	2000	1415.74	6000000	608117
4	228	44	367455	06-06-2014	IL	500/1000	1000	1583.91	6000000	610706

Counting of each unique element repeated in each of the column of the dataset. In machine learning nunique() helps in determining the cardinality of the each column in the dataset.

```
dataset.nunique()
```

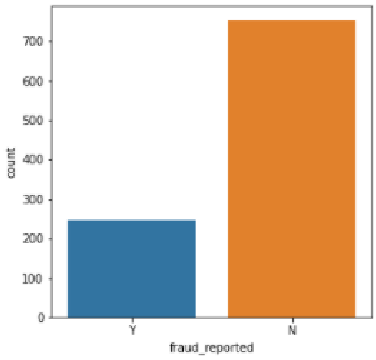
```
months_as_customer             391
age                             46
policy_number                 1000
policy_bind_date               951
policy_state                     3
policy_csl                       3
policy_deductable                3
policy_annual_premium          991
umbrella_limit                  11
insured_zip                    995
insured_sex                      2
insured_education_level          7
insured_occupation              14
insured_hobbies                 20
insured_relationship             6
capital-gains                  338
capital-loss                   354
incident_date                   60
incident_type                    4
collision_type                   4
incident_severity                4
authorities_contacted            5
incident_state                   7
incident_city                    7
incident_location             1000
incident_hour_of_the_day        24
number_of_vehicles_involved      4
property_damage                  3
bodily_injuries                  3
witnesses                        4
police_report_available          3
total_claim_amount             763
injury_claim                   638
property_claim                 626
vehicle_claim                  726
auto_make                       14
auto_model                      39
auto_year                       21
fraud_reported                   2
```

Dropping policy number, insured zip and location as it can be found that it is unique for every other individual and it can't help in our prediction as it can be reflected such as a serial number value.

We can observe that the target variable 'fraud_reported' has only two classifications that are yes and no, which means output is binary i.e. it is either 0 or 1. So we will be using classifiers to train and predict our dataset.

Below is the code used to visualize the countplot:

```
plt.figure(figsize = (5,5))

ax=sns.countplot('fraud_reported',data=dataset)

plt.show()
```



Even so the dataset is free of null values three of the columns in the dataset contains '?' values in the dataset. The columns which contains '?' are 'collision type', 'police report available' and 'property damage'.

```
for i in dataset:

    print(dataset[i].value_counts(),'\n\n')
```

```
Rear Collision     292
Side Collision     276
Front Collision    254
?                  178
Name: collision_type, dtype: int64
```

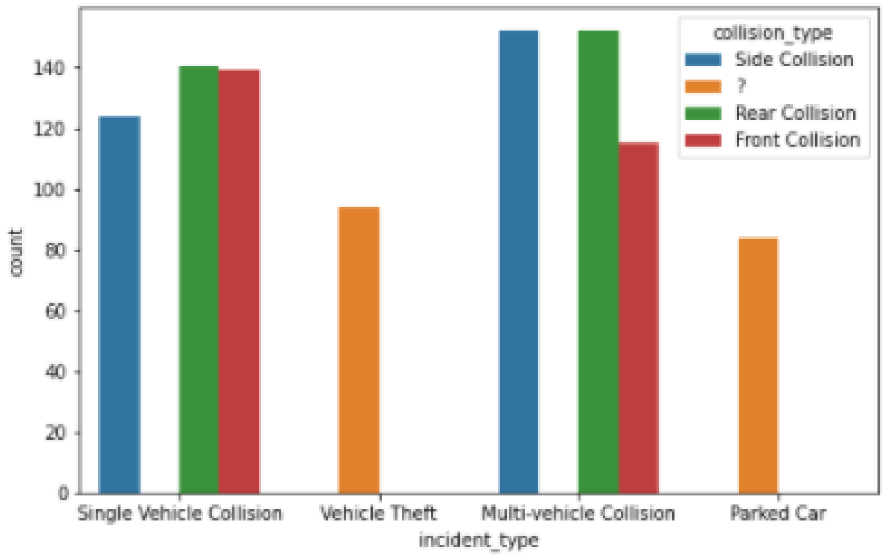
```
?      360
NO     338
YES    302
Name: property_damage, dtype: int64
```

```
?      343
NO     343
YES    314
Name: police_report_available, dtype: int64
```

The count of missing values is large compared to the number row of the dataset so can't drop the rows, have to handle each column accordingly.

Below is the code used to visualize the countplot:

```
plt.figure(figsize = (8,5))

ax=sns.countplot(x='incident_type',hue='collision_type',data=dataset)

plt.show()
```

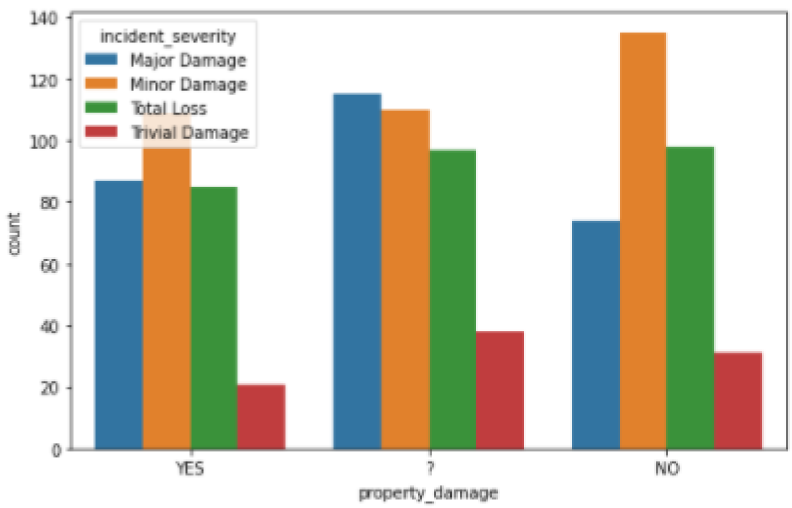



```
dataset['collision_type'].replace('?','No Collision',inplace=True)

dataset['collision_type'].value_counts()
```

Comparing ‘incident_type’ column and ‘collision_type’ column we can see that vehicle theft and parked car have ‘?’ represented in ‘incident_type’ column. It is evident that vehicle theft and parked car can never collide therefore we can assume ‘?’ to be as ‘no collision’.

Below is the code used to visualize the countplot:

```
plt.figure(figsize = (8,5))

ax=sns.countplot(x='property_damage',hue='incident_severity',data=trial_set)

plt.show()
```



To ‘?’ we must first divide the dataset into 4 different dataset according to incident type:

```
major_d=dataset[(dataset.incident_severity=='Major Damage')]

total_d=dataset[(dataset.incident_severity=='Total Loss')]

minor_d=dataset[(dataset.incident_severity=='Minor Damage')]
```

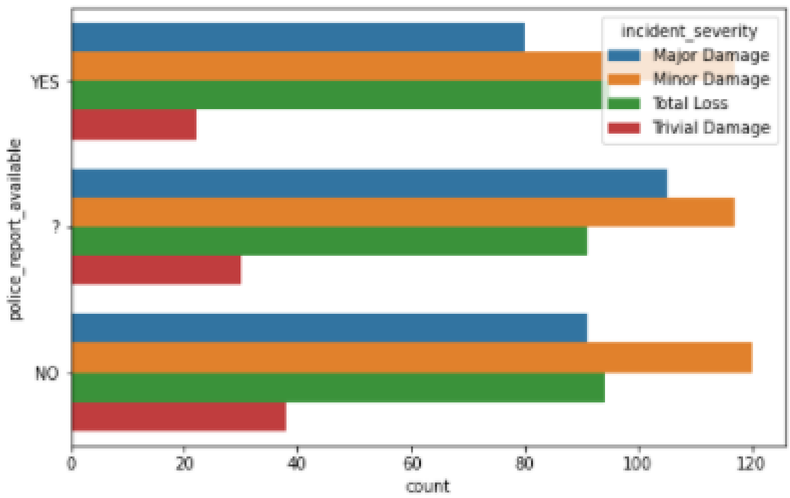
```
trivial_d=dataset[(dataset.incident_severity=='Trivial Damage')]
```

Replacing the '?' property damage column.

```
major_d.property_damage.replace('?','YES',inplace=True)

total_d.property_damage.replace('?','YES',inplace=True)

minor_d.property_damage.replace('?','NO',inplace=True)

trivial_d.property_damage.replace('?','NO',inplace=True)
```

Property damage means the damage done to any of the insured belongings inside the automobile during the incident. Comparing 'property_damage' column and 'incident_severity' column because only major and total loss type of incident can damage any of the property inside the automobile, therefore we will be replacing '?' in property damage column according to the incident severity.

```
plt.figure(figsize = (8,5))

ax=sns.countplot(y='police_report_available',hue='incident_severity',data=trial_set)

plt.show()
```



Same go for police report as well we don't need a police report for a minor or trivial damage, we need police report only for major or total loss. Therefore, we will also replace '?' in police report available column according to incident severity.

```
major_d.police_report_available.replace('?','YES',inplace=True)

total_d.police_report_available.replace('?','YES',inplace=True)

minor_d.police_report_available.replace('?','NO',inplace=True)

trivial_d.police_report_available.replace('?','NO',inplace=True)
```

As we have dealt with all the '?' in the dataset, and we can attach the divided dataset into dataset and form new dataset.

```
frame=[major_d,total_d,minor_d,trivial_d]

new_set=pd.concat(frame)
```

```
new_set=new_set.sort_index()
```

Reducing complexity in the columns:

The policy bind date columns has too many unique elements i.e., to reduce the complexion, I am splitting them into three categories namely date, month and year and append these three columns to our existing dataset.

```
new_set[['policy_day','policy_month','policy_year']]=new_set['policy_bind_date'].str.split('-',expand=True)
```

policy_bind_date
17-10-2014
27-06-2006
06-09-2000
25-05-1990
06-06-2014

policy_day	policy_month	policy_year
17	10	2014
27	06	2006
06	09	2000
25	05	1990
06	06	2014

We will repeat the same process for incident date column as we performed on policy bind date column.

```
new_set[['incident_day','incident_month','incident_year']]=new_set['incident_date'].str.split('-',expand=True)
```

incident_date
25-01-2015
21-01-2015
22-02-2015
10-01-2015
17-02-2015

incident_day	incident_month	incident_year
25	01	2015
21	01	2015
22	02	2015
10	01	2015
17	02	2015

the data of the policy and report date are in the form number visually but it is indicated as an object-type we need to convert them into numerical form. Also I am dropping policy and incident date as we created sub columns of them and appended them to the new dataset. Incident year is same for all the data therefore dropping this column as well.

```
new_set.drop(columns=['incident_year','policy_bind_date','incident_date'],inplace=True)

new_set['policy_day']=pd.to_numeric(new_set['policy_day'])

new_set['policy_month']=pd.to_numeric(new_set['policy_month'])

new_set['policy_year']=pd.to_numeric(new_set['policy_year'])

new_set['incident_day']=pd.to_numeric(new_set['incident_day'])

new_set['incident_month']=pd.to_numeric(new_set['incident_month'])
```

```
new_set.dtypes
```

```
policy_day                  int64
policy_month                int64
policy_year                 int64
incident_day                int64
incident_month              int64
```

Age also has too much complexion so in order to nullify it, I will convert the age into three categories that is youngster indicating between the age of 18-29, bachelors between the age 30-49 and elders 50 and above. I am indicating them with numerical integers 0,1 and 2 respectively.

```
new_set.loc[new_set['age'].between(18,29),'age']=0

new_set.loc[new_set['age'].between(30,49),'age']=1

new_set.loc[new_set['age'].between(50,70),'age']=2
```

Also, the incident hour complexion can be decreased by making the categories within it. I am dividing the categories into 5 parts i.e., mid-night which is from 12am to 3am, early morning from 4am to 7am, mid-day from 8am to 4pm, evening from 5pm to 7pm and finally night from 8pm to 11pm. Also, for this I will replace all values with numerical values ranging from 0~4 (0 and 4 included).

```
new_set.loc[new_set['incident_hour_of_the_day'].between(0,3),'incident_hour_of_the_day']=0

new_set.loc[new_set['incident_hour_of_the_day'].between(4,7),'incident_hour_of_the_day']=1

new_set.loc[new_set['incident_hour_of_the_day'].between(8,16),'incident_hour_of_the_day']=2

new_set.loc[new_set['incident_hour_of_the_day'].between(17,19),'incident_hour_of_the_day']=3

new_set.loc[new_set['incident_hour_of_the_day'].between(20,23),'incident_hour_of_the_day']=4
```

Note: I am converting all the data into numerical values because machine learning model can only be trained when the data is in numerical form.

Injury, property and vehicle claim are already included in the total claim column therefore, I am replacing it as 0 or 1 if they have claimed any amount on behalf of it.

```
new_set.loc[new_set['injury_claim'].between(1,100000),'injury_claim']=1

new_set.loc[new_set['property_claim'].between(1,100000),'property_claim']=1

new_set.loc[new_set['vehicle_claim'].between(1,100000),'vehicle_claim']=1
```

If they haven’t claimed any then it will be automatically zero.

I am creating a dataset to visualize the data in pie chart form

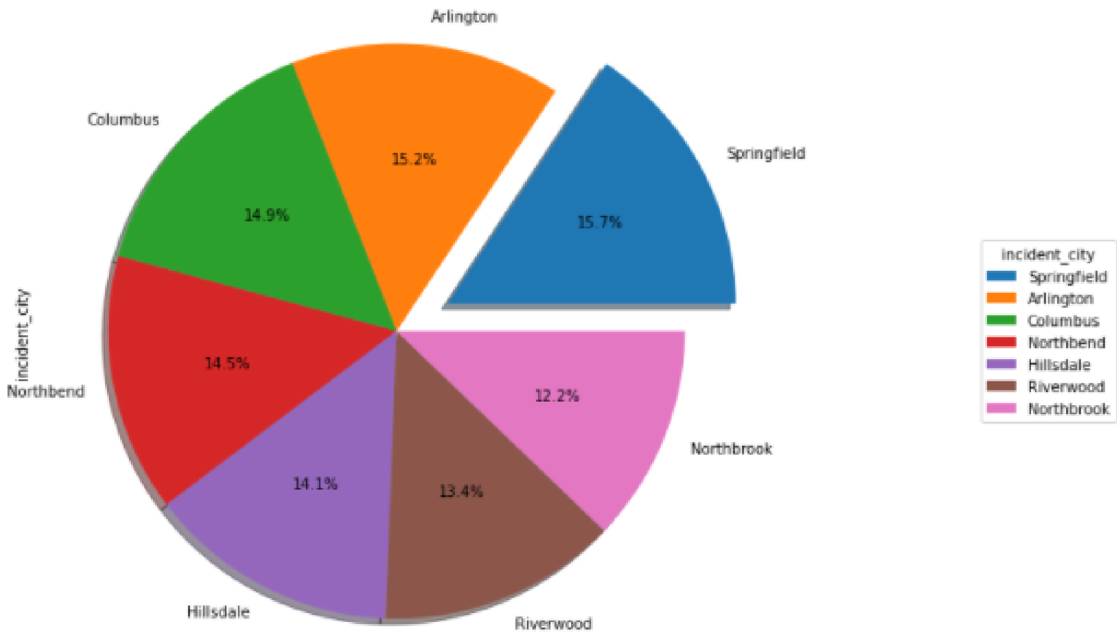
```
visual_set=new_set.drop(columns=['total_claim_amount','months_as_customer','policy_annual_premium','auto_make','capital-gains','insured_occupation','insured_hobbies','capital-loss','auto_year','auto_model','policy_day','policy_year'])
```

As the visual data set is ready, we can perform uni-variate analysis. Uni-variate analysis means analysing one variable to describe its purpose and to find pattern that particular variable data has to offer so that we can summarize it.

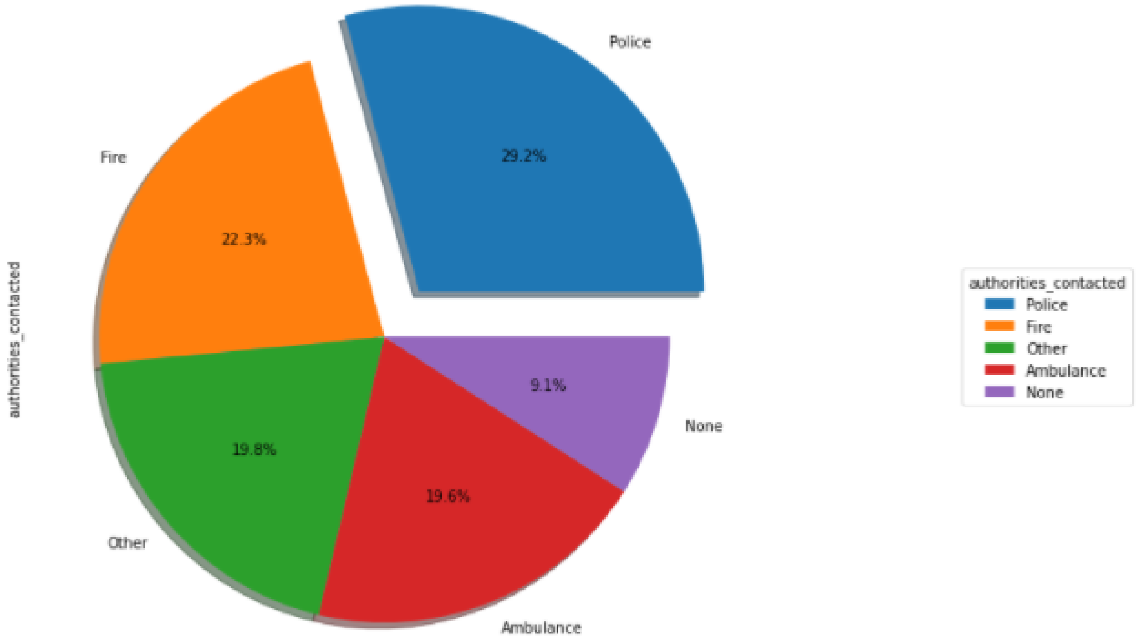
```
for i in visual_set:
    l=i
    print('\033[1m'+l+'\033[1m')
    y=visual_set[i].value_counts()
    exp=[0.2]
    j=int(visual_set[i].nunique())
    k=1
    while k < j:
        exp.append(0)
        k+=1
    z=y.plot.pie(figsize=(9,9),explode=exp, autopct='%2.1f%%', shadow=True)
    z.legend(title =i,loc ="center left",bbox_to_anchor =(1.3, 0, 0.5, 1))
    plt.show()
    print('\n\n')
```

Below figures are few of the outputs of the above code.

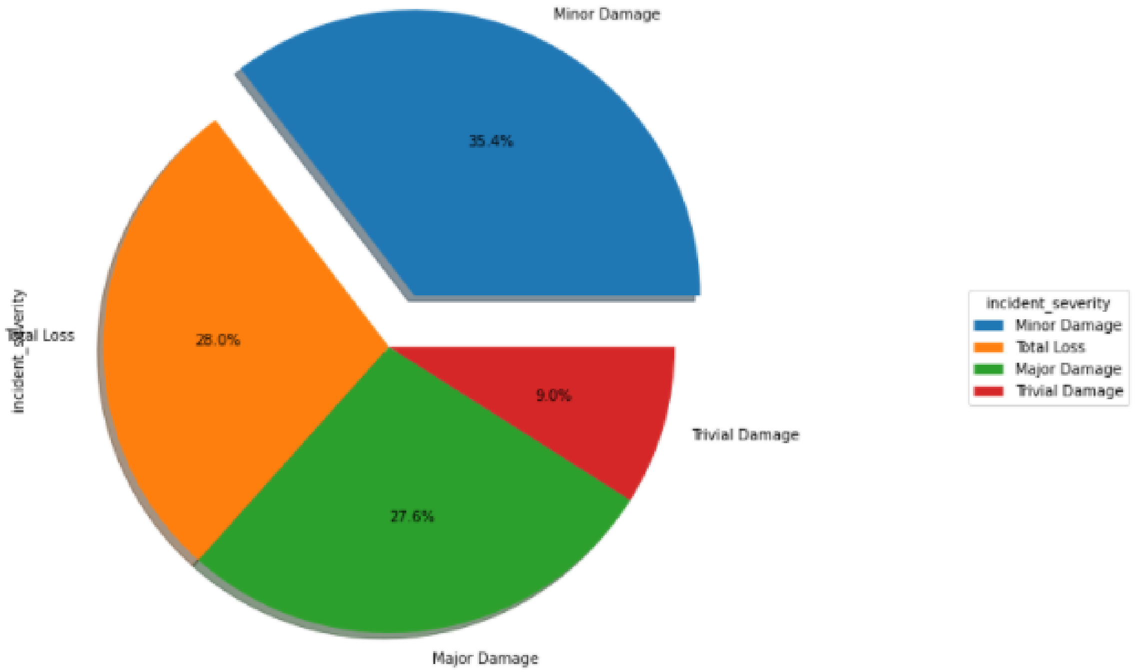
incident_city



authorities_contacted



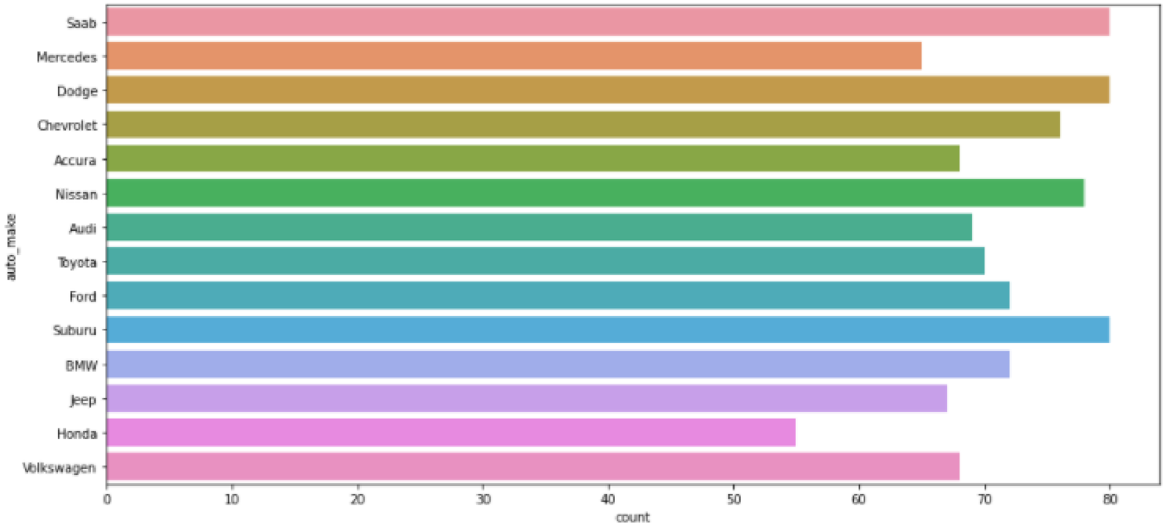
incident_severity



For columns with more unique elements, I visualized using count plot to represent the data in a more eye pleasing way.

```
plt.figure(figsize = (15,7))
ax=sns.countplot(y='auto_make',data=dataset)
plt.show()
```

Below is the output of the above code:



Uni-variate Analysis observation:

1. There are more Bachelors involved.

2. policy state csl and deductible are balanced

3. most customer have zero umbrella limit

4. insured education and relationship is also balanced

5. multi-vehicle and single vehicle collision are most dominant in the dataset when it comes to incident type

6. Rear collision is slightly more than side and front collision

7. minor damage is more in number in incident severity

8. most customers contacted police when the incident occurred

9. NY is the state has with most incident occurring

10. incident of city is balanced between all 7 cities

11. most incidents occurred at mid-day

12. most incidents involved only one vehicle

13. body injuries is ranged between 0~2 and is balanced

14. witnesses are ranged between 0~3 and is balanced

15. almost all the clients have claimed for all the 3 types of claims

16. Fraud reported is imbalanced therefore the dataset is imbalanced

17. policy month is balanced

18. policy date is balanced

19. there are only 3 incident months i.e., jan, feb, and march and march has very minimum incidents

20. most insure are interest in reading and mentioned it as their hobbies

21. there are variety of car model though wrangler is the most used model of all the car model amongst the clients

22. year-1995 has most automobiles purchase

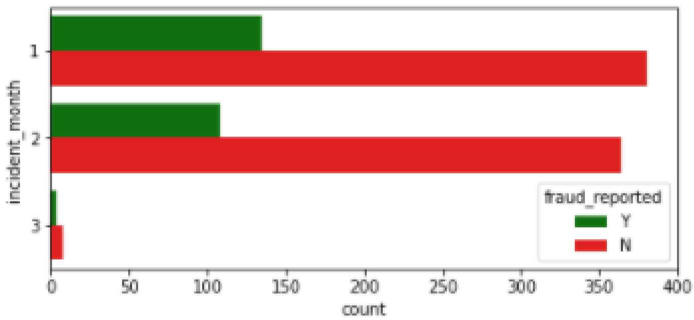
23. Saab, Dodge, Suburu have equal and highest count amongst all the automakers

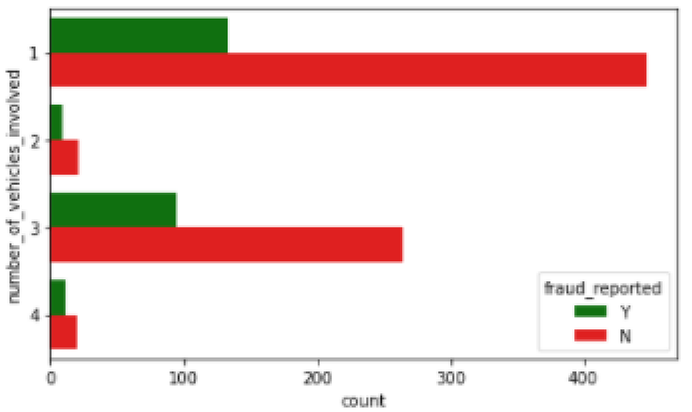
Bi-variate Analysis:

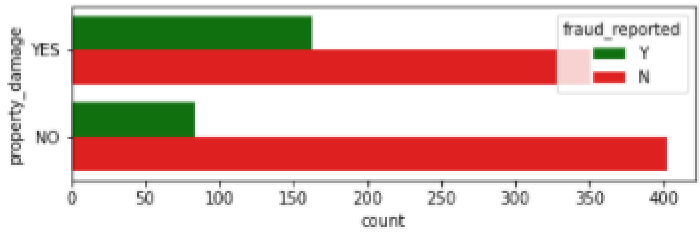
Bi-variate analysis is same as the Uni-variate analysis but we compare it with target variable to find pattern that are affecting the outcome of the predict i.e., via visualization of the data.

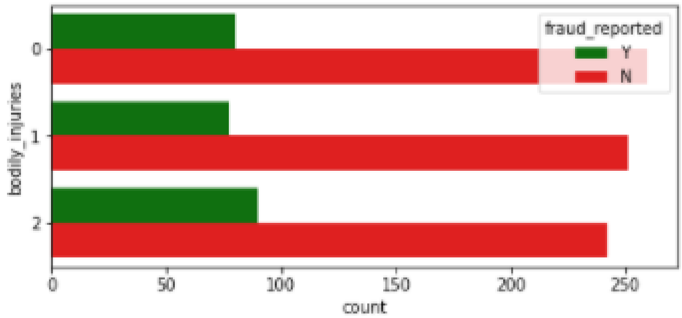
```
for i in visual_set:
    x=visual_set[i].nunique()
    plt.figure(figsize = (7,x))
    ax=sns.countplot(y=i,hue='fraud_reported',data=visual_set,palette=['green','red'])
    plt.show()
```

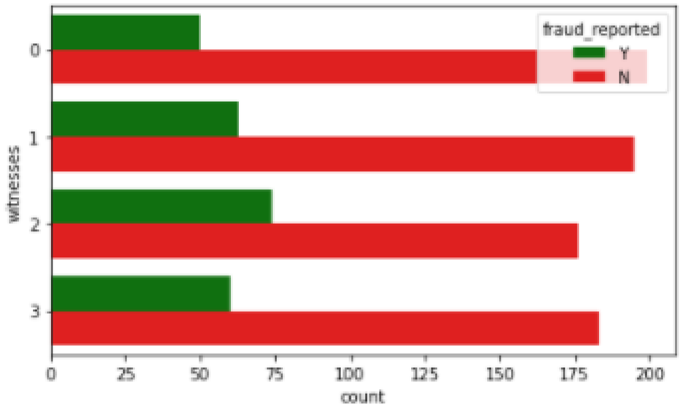
Below figures are few of the outputs of the above code:











Bi-variate Analysis Observation:

1. client with zero umbrella limit have most frauds

2. JD and MD educated are mostly involved in frauds

3. client who are claiming for single and multi-collision are partially fraud

4. major damage are mostly fraud cases

5. incident occurring in board day light have most fraud

Now that the dataset is clean and simple, we can now convert all of the ordinal data into numerical data.

Below is the code to convert ordinal data to numerical data:

```
le=LabelEncoder()

for i in new_set:

    if new_set[i].dtype=='object':

        new_set[i]=le.fit_transform(new_set[i])
```

Label Encoder helps in converting the ordinal form of data into numerical form i.e., by replacing all the similar elements in that column with a particular numerical integer i.e., starting from value 0.

Looking at the column data types:

```
new_set.dtypes
```

Below image is output of the above code.

```
months_as_customer               int64
age                              int64
policy_state                     int32
policy_csl                       int32
policy_deductable                int64
policy_annual_premium          float64
umbrella_limit                   int64
insured_sex                      int32
insured_education_level          int32
insured_occupation               int32
insured_hobbies                  int32
insured_relationship             int32
capital-gains                    int64
capital-loss                     int64
incident_type                    int32
collision_type                   int32
incident_severity                int32
authorities_contacted            int32
incident_state                   int32
incident_city                    int32
incident_hour_of_the_day         int64
number_of_vehicles_involved      int64
property_damage                  int32
bodily_injuries                  int64
witnesses                        int64
police_report_available          int32
total_claim_amount               int64
injury_claim                     int64
property_claim                   int64
vehicle_claim                    int64
auto_make                        int32
auto_model                       int32
auto_year                        int64
fraud_reported                   int32
policy_day                       int64
policy_month                     int64
policy_year                      int64
incident_day                     int64
incident_month                   int64
```

All the data is now represented in numerical form.

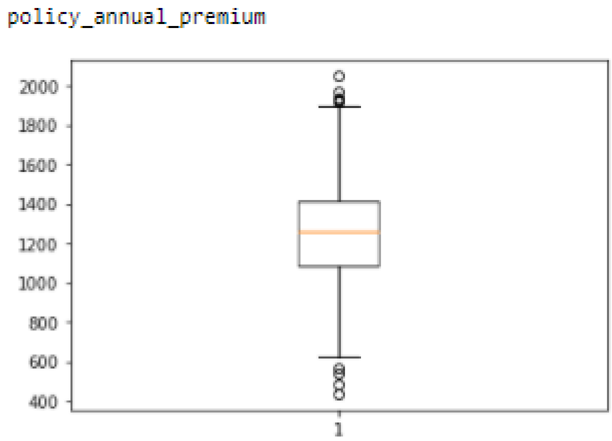
After cleaning and making dataset simple we must arrest the outliers in the dataset. Outliers can be basically defined as error data or we see them as the value that are off the limits i.e., in much simpler words the existing value doesn't make generally sense when compared all data of the dataset. To arrest outliers, one must have general idea of the variable limits. For example, one can simply say that human life span is up until 80 and 90 to 100 which farfetched and if value is around 180 years, we can say it as an outlier.

Box-plot help in visualizing the outliers in the dataset. Below is the code to visualize outliers in the dataset.

```
fig = plt.figure(figsize =(5, 5))
```

```
for i in new_set:
    print(i)
    plt.boxplot(new_set[i])
    plt.show()
```

Below figure is one of the outputs of the above code:



As we can see policy annual premium has some outliers, we must arrest the outliers.

IQR method is one the best methods to arrest outliers in a dataset:

I am defining a function to arrest the outlier using IQR Method.

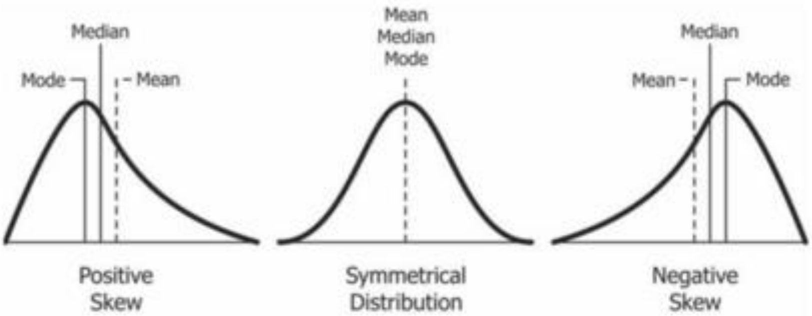
```
def arr_out(df,column):
    Q1=df[column].quantile(0.25)
    Q3=df[column].quantile(0.75)
    IQR=Q3-Q1
    whisker_width = 1.5
    news_outliers = df[(df[column] < Q1 - whisker_width*IQR) | (df[column] > Q3 + df*IQR)]
    lower_whisker = Q1 -(whisker_width*IQR)
    upper_whisker = Q3 + (whisker_width*IQR)

df[column]=np.where(df[column]>upper_whisker,upper_whisker,np.where(df[column]<lowe
r_whisker,lower_whisker,df[column]))


arr_out(new_set,'policy_annual_premium')
```

And the outliers inside the dataset are arrested. Now we can check the normal distribution of each column:

Normal distribution is done to visually check the skewness of the columns, it is better to not have skewness in the data.

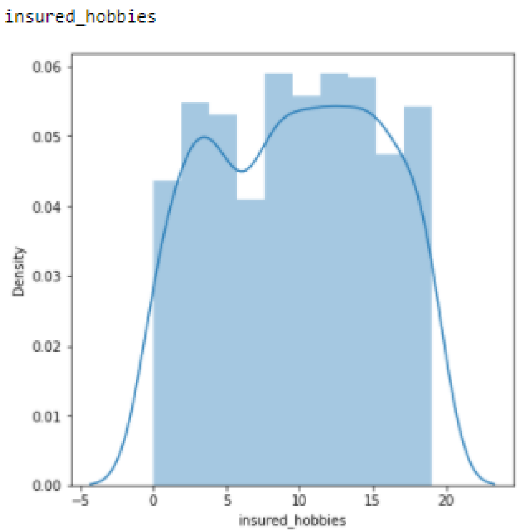
Note: It is better to not alter the skewness if the variable has good corelation with the target column. Log and sqrt method are used to reduce the skewness.

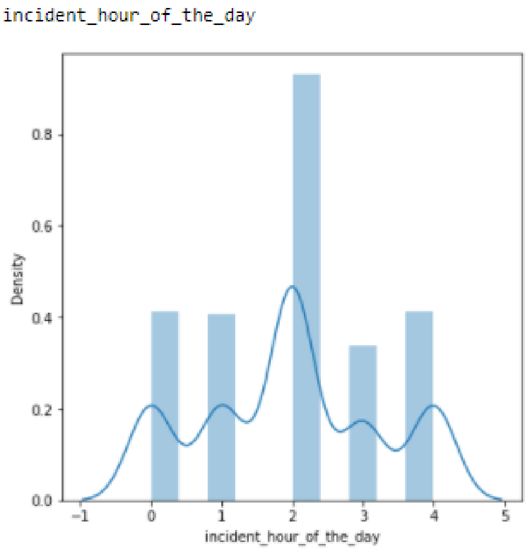
For symmetrical distribution the skewness is zero. As you can see in the below figure you can get a visualize idea of what a skewness is:




```
for i in new_set:
    print(i)
    plt.figure(figsize=(6,6))
    sns.distplot(new_set[i])
    plt.show()
```

Below images are few outputs of the above code.





Skewness can also be checked by using skew()

```
new_set.skew()
```

```
months_as_customer             0.362177
policy_state                  -0.026177
policy_csl                     0.088928
policy_deductable              0.477887
policy_annual_premium          0.016003
umbrella_limit                 1.806712
insured_sex                    0.148630
insured_education_level       -0.000148
insured_occupation            -0.058881
insured_hobbies               -0.061563
insured_relationship           0.077488
capital-gains                  0.478850
capital-loss                  -0.391472
collision_type                -0.177814
incident_severity              0.279016
authorities_contacted         -0.121744
incident_state                -0.148865
incident_city                  0.049531
incident_hour_of_the_day       0.052528
number_of_vehicles_involved    0.502664
property_damage               -0.056106
bodily_injuries                0.014777
witnesses                      0.019636
police_report_available       -0.040068
total_claim_amount            -0.595351
injury_claim                  -6.094015
property_claim                -7.056933
auto_make                     -0.018797
auto_model                    -0.080773
auto_year                     -0.048289
fraud_reported                 1.175051
policy_day                     0.053237
policy_month                  -0.016994
policy_year                    0.052511
incident_day                   0.039711
incident_month                 0.267378
```

One must always look into corelation between each and every independent variable in order to check if there is any multi-collinearity between the independent variables as multi-collinearity affects the machine learning performance as result the outcome of the prediction will be shambolic.

Corelation heatmap is one of the best ways to look into the co-relation between one and every variable.

```
plt.figure(figsize=(25,40))

sns.heatmap(new_set.corr(), annot=True)

plt.show()
```

it is better to have corelation value ranging between -0.5 to +0.5.

incident type, vehicle claim and age has very corelation with many of the independent variables. Therefore, dropping off these variables.

```
new_set.drop(columns=['incident_type','vehicle_claim','age'],inplace=True)
```

Now that data is all ready, we can separate independent and target variable and normalize the data and feed it to machine learning model.

```
#separating feature columns and target column

X=new_set.drop(columns=['fraud_reported'])

Y=new_set['fraud_reported']
```

As fraud reported column is the target data, we separating it from the rest of the dataset.

Normalizing / Standardizing the data is important as it makes the data unbiased and lets machine model give equal importance to all the data in the dataset. Standard Scalar is one of the sklearn libraries which helps to standardize the data.

```
scalar= StandardScaler()

X_scaled= scalar.fit_transform(X)
```

Checking the VIF (variance inflation factor) value of each column after normalizing the data helps us to cross check multi-collinearity between the columns. It is better to constrain the VIF value below 5.

Note: If the VIF value is more than 5 look for the almost similar VIF value column and drop the column which has less corelation with the target column

```
vif_data = pd.DataFrame()

vif_data["feature"] = X.columns

vif_data["VIF"] = [variance_inflation_factor(X_scaled, i) for i in range(X_scaled.shape[1])]

vif_data
```

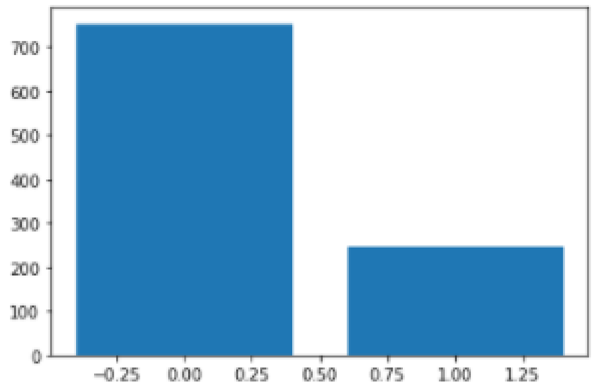
below are the VIF values of few of the dataset columns:

	feature	VIF
0	months_as_customer	1.054523
1	policy_state	1.038372
2	policy_csl	1.032477
3	policy_deductable	1.041521
4	policy_annual_premium	1.031743
5	umbrella_limit	1.034872
6	insured_sex	1.022829
7	insured_education_level	1.053597
8	insured_occupation	1.016504
9	insured_hobbies	1.043976
10	insured_relationship	1.047821
11	capital-gains	1.038238
12	capital-loss	1.041099

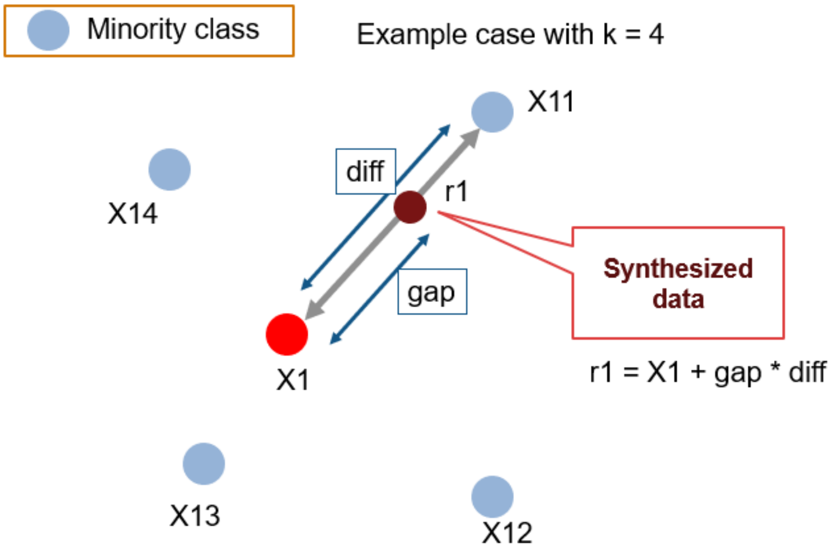
We know that the dataset is imbalance, let's us just once again look how imbalance it is:

```
counter = Counter(Y)
for k,v in counter.items():
    per = v / len(Y) * 100
    print('Class=%d, n=%d (%.3f%%)' % (k, v, per))
# plot the distribution
pyplot.bar(counter.keys(), counter.values())
pyplot.show()
```

```
Class=1, n=247 (24.700%)
Class=0, n=753 (75.300%)
```

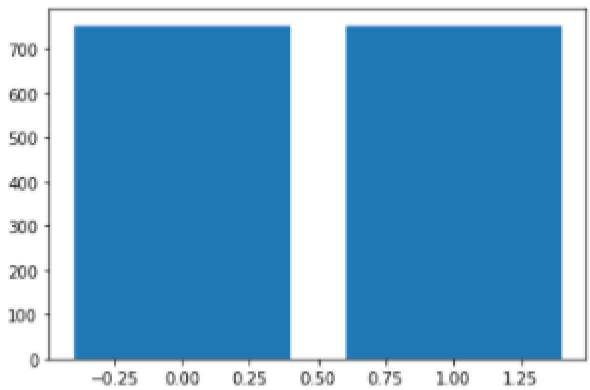


As we can see about 75% of the data is not fraud, so even if the model has an accuracy of 75% it is a poor model. Imbalanced dataset always causes the dataset biased due which the model gets biased. There are two methods to balance the dataset i.e., up sampling and under sampling, when the dataset is large, we prefer under sampling and when the dataset is small, we prefer up sampling. SMOTE analysis is one of the methods to up sample the dataset. Synthetic Minority Oversampling Technique (SMOTE) is used to oversample the minor value in the target data in-order to make the dataset balance and help the model to over-come overfitting. SMOTE can be used by importing imblearn libraries.




```
#balancing the dataset
oversample = SMOTE()
X_over, Y_over = oversample.fit_resample(X_scaled, Y)
counter = Counter(Y_over)
for k,v in counter.items():
    per = v / len(Y_over) * 100
    print('Class=%d, n=%d (%.3f%%)' % (k, v, per))
# plot the distribution
pyplot.bar(counter.keys(), counter.values())
pyplot.show()
```

```
Class=1, n=753 (50.000%)
Class=0, n=753 (50.000%)
```



Exploratory Data Analysis Summary:

1. Dataset has 1000 rows and 40 columns
2. Check and eliminated Null values apart from null values found '?' elements in the dataset, replaced them accordingly in a suitable sense.
3. Checked unique number of values in each column.
4. Noted the count of each unique value.
5. Decreased the complexity of the dataset.
6. Runed Uni and Bi Variate Analysis to find pattern and factor that affect the outcome.
7. Arrested Outliers in the dataset.
8. Checked the Skewness of the columns.
9. Removed multi-collinearity from the dataset
10. Oversampled the data in-order to make the dataset balanced.

Training and testing the data using machine models:

Before Training the data, we must split the data into two sets i.e., train data and test data. Train test split help in dividing the data into two different sets according to how much percentage of data we want to split for test and train.

```
X_train, X_test, Y_train, Y_test = train_test_split(X_over, Y_over, train_size=0.8)
```

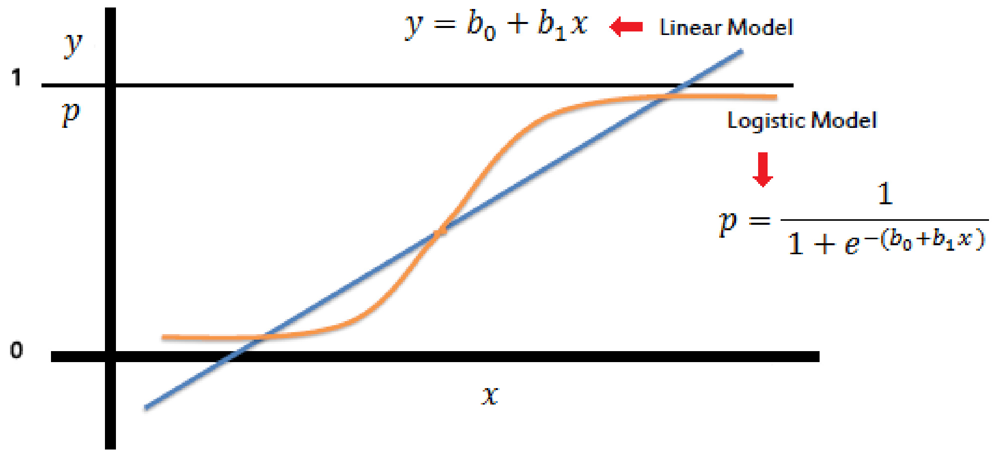
X_train contains all the independent variables and Y_train contains all the target variable corresponding to the X_train, same goes for X_test and Y_test.

train_size of 0.8 indicates that I am splitting the data into 80% train and 20% test.

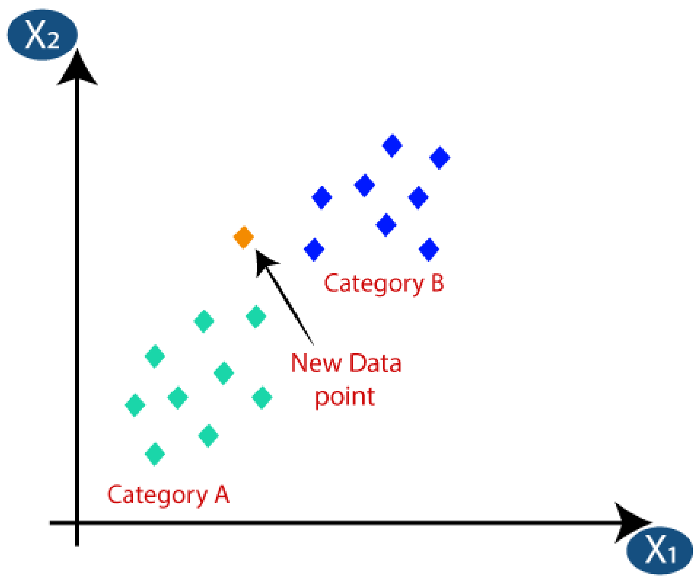
When the outcome of the prediction is binary, we use Classifier machine learning model for prediction. For this dataset I am using four models:

i. Logistic Regression
ii. KNeighbors Classifer
iii. XGB Classifier
iv. Random Forest Classifier

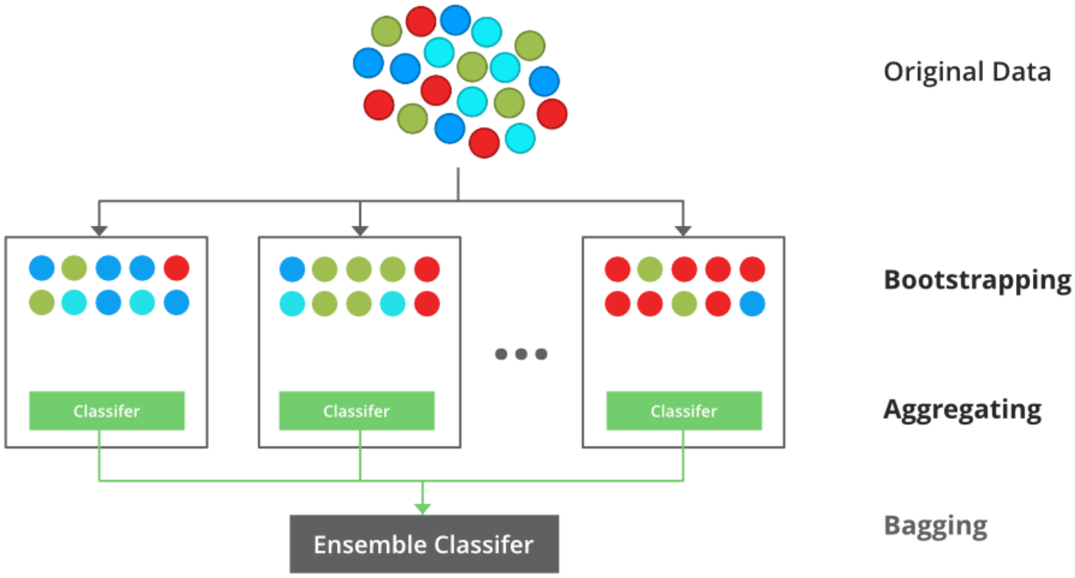
Logistic Regression is a supervised learning classification algorithm used to predict the probability of a target variable. The nature of target or dependent variable is dichotomous, which means there would be only two possible classes



K Nearest Neighbor algorithm falls under the Supervised Learning category and is used for classification (most commonly) and regression. It is a versatile algorithm also used for imputing missing values and resampling datasets. As the name (K Nearest Neighbor) suggests it considers K Nearest Neighbors (Data points) to predict the class or continuous value for the new Datapoint

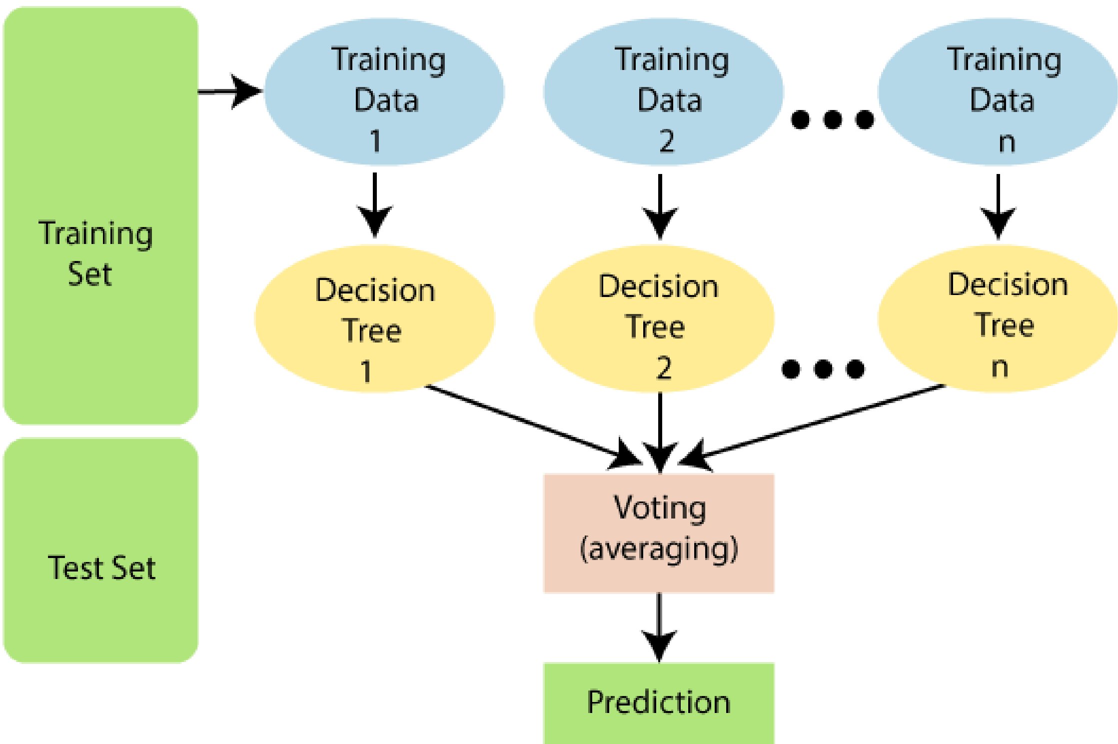


XGB is an implementation of gradient boosted decision trees designed for speed and performance that is dominative competitive machine learning



Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.



Model 1:

```
logr = LogisticRegression()

solvers = ['newton-cg', 'lbfgs', 'liblinear']

penalty = ['l2']

c_values = [100, 10, 1.0, 0.1, 0.01]

# define grid search

grid = dict(solver=solvers,penalty=penalty,C=c_values)

cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

grid_search = gs(estimator=logr, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',error_score=0)

grid_search.fit(X_train,Y_train)
```

loading the model and creating parameter to run grid search cv. Grid search Cross validation is a hyperparameter tuning method by which we can find best parameter to build a model for our model training.

For 1st model I am taking logistic regression and defining parameters for it i.e., solvers, penalty and c_values. There are various cross validation techniques I am using Stratified KFold for this method. Stratified KFold help in cross check the model with balance 0 and 1s in this process. In cross validation the data is divided into splits i.e., if n =10 splits the model uses 9 splits to train and 1 to test and after each process the test split becomes the train and one of the other 9 splits become test split, this process is repeated for 10 cycles as so.

By fitting X_train and Y_train in grid_search CV we train the model for all the given parameters to find the best suited parameters for the dataset.

```
grid_search.best_score_
```

```
0.7619077134986227
```

As we can see the best score for the best parameters now let fetch the model best parameters.

```
grid_search.best_estimator_
```

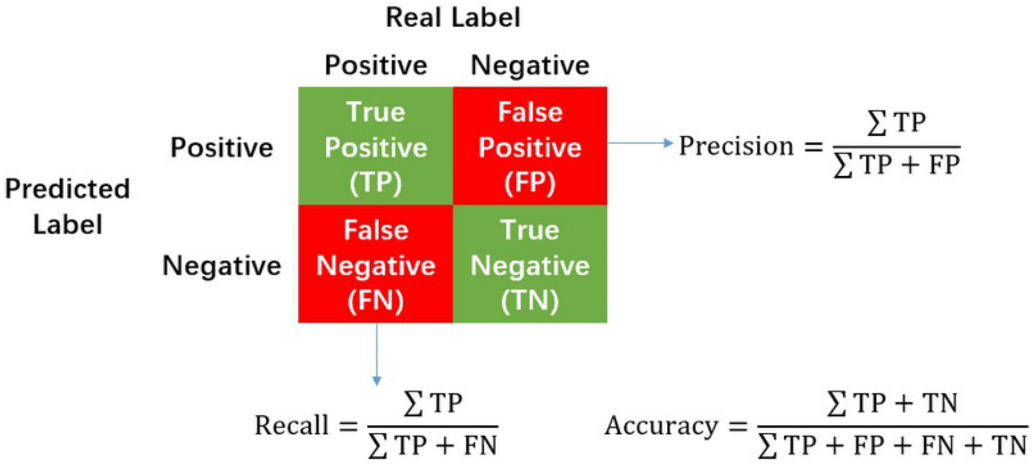
```
LogisticRegression(C=0.01, solver='newton-cg')
```

The above image indicates the best parameters for the model that we should build for getting best score while prediction of the dataset.

```
model1=LogisticRegression(C=0.01, solver='newton-cg')

model1.fit(X_train,Y_train)

p1=model1.predict(X_test)

print(classification_report(p1, Y_test))
```

we are creating the model using the best estimators and training them. After creating the model, we fit the model for X_train and Y_train. After training we predict the result for X_test and compare it with Y_test.

Classification report is used for classifier machine learning model in order to get a better look at the results as for classification dataset accuracy doesn't justify when there is imbalance in the dataset due which we take a look at precision, recall and f1 score of the model.



The above image will give you good perspective about precision and recall. The below image is the output of the above code.

	precision	recall	f1-score	support
0	0.81	0.76	0.78	165
1	0.73	0.79	0.76	137
accuracy			0.77	302
macro avg	0.77	0.77	0.77	302
weighted avg	0.77	0.77	0.77	302

As we can see precision, recall and f1 score for 0 and 1 are almost balance. i.e., the model has 77% accuracy.

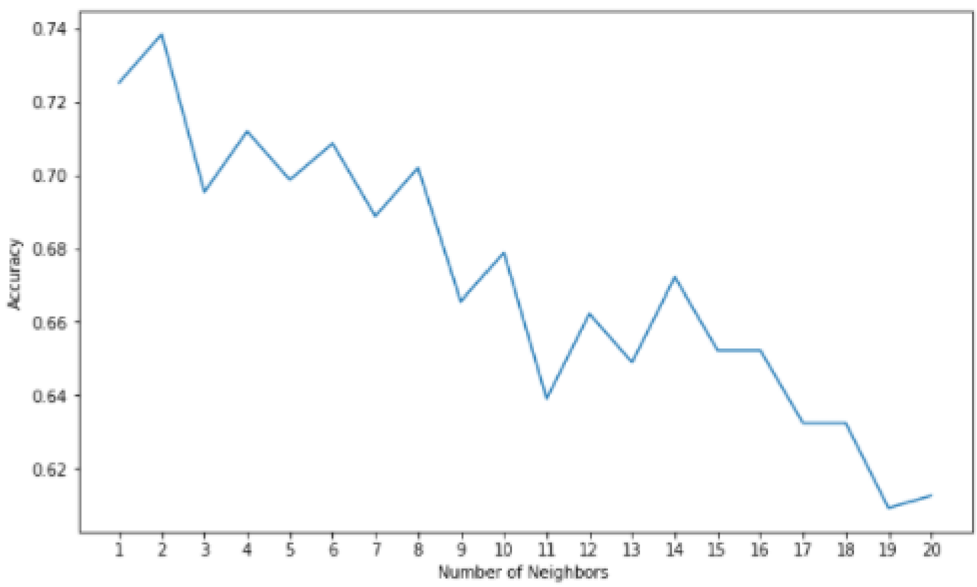
Model 2:

Before we get to Hyperparameter tuning of KNN Classifier we must get best neighbor values. For that I am using the below code to figure it out:

```
knc = KNeighborsClassifier()
mean_acc = np.zeros(20)
for i in range(1,21):
    #Train Model and Predict
    knc = KNeighborsClassifier(n_neighbors = i).fit(X_train,Y_train)
    yhat2= knc.predict(X_test)
    mean_acc[i-1] = accuracy_score(Y_test, yhat2)
```

```
loc = np.arange(1,21,step=1.0)
plt.figure(figsize = (10, 6))
plt.plot(range(1,21), mean_acc)
plt.xticks(loc)
plt.xlabel('Number of Neighbors ')
plt.ylabel('Accuracy')
plt.show()
```

Output:



As we can see the accuracy is more for 1,2 and 3 neighbors.

```
knn=KNeighborsClassifier()
para={
        'n_neighbors':[1,2,3],
        'weights':['uniform', 'distance'],
        'algorithm':['auto', 'ball_tree', 'kd_tree', 'brute','auto'],
        'leaf_size':[30,40,50,60],
        'p':[2,3],
        'metric':['minkowski']


}
knn_gs= gs(estimator =knn, param_grid=para,cv=10, n_jobs=5)
```

```
knn_gs.fit(X_train,Y_train)
```

We repeating same process as we did for model 1 creating the model and defining the parameters for the model and hyperparameter tuning using gridsearch cv and fit X_train and Y_train to find best estimators.

```
knn_gs.best_score_
```

```
0.7666460055096419
```

```
knn_gs.best_estimator_
```

```
KNeighborsClassifier(n_neighbors=2)
```

```
model2=KNeighborsClassifier(n_neighbors=2)

model2.fit(X_train,Y_train)

p2=model2.predict(X_test)

print(classification_report(p2, Y_test))
```

```
              precision    recall  f1-score   support

           0       0.71      0.88      0.79       125
           1       0.90      0.75      0.82       177

    accuracy                           0.80       302
   macro avg       0.81      0.82      0.80       302
weighted avg       0.82      0.80      0.81       302
```

For KNN Classifier the precision difference between 0 and 1 is drastically high so this model not suitable for our dataset.

Model 3:

Creating XGB Classifier and defining parameter for hyperparameter tuning of the model.

```
xgb= XGBClassifier()

param={

    'n_estimators':[200,250,300,350],

    'learning_rate':[0.01,0.1,0.15,0.2],

    'subsample':[0.3,0.4,0.6],

    'max_depth':[3,5,7,9,10],

    'colsample_bytree':[0.1,0.2,0.3,0.4],

    'min_child_weight':[1,2,3,4,5]

}

xgb_C=gs(xgb,param_grid=param,cv=10,refit=True,n_jobs=10)

xgb_C.fit(X_train,Y_train)
```

We repeating same process as we did for model 1 creating the model and defining the parameters for the model and hyperparameter tuning using gridsearch cv and fit X_train and Y_train to find best estimators.

```
xgb_C.best_score_
```

```
0.895378787878788
```

```
xgb_C.best_estimator_
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.4,
              enable_categorical=False, gamma=0, gpu_id=-1,
              importance_type=None, interaction_constraints='',
              learning_rate=0.2, max_delta_step=0, max_depth=9,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=350, n_jobs=8, num_parallel_tree=1, predictor='auto',
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=0.3, tree_method='exact', validate_parameters=1,
              verbosity=None)
```

By far of all the model we have trained XGB has the best score of all. Let check its classification report.

```
model3=XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
          colsample_bynode=1, colsample_bytree=0.4,
          enable_categorical=False, gamma=0, gpu_id=-1,
          importance_type=None, interaction_constraints='',
          learning_rate=0.2, max_delta_step=0, max_depth=9,
          min_child_weight=1, monotone_constraints='()',
          n_estimators=350, n_jobs=8, num_parallel_tree=1, predictor='auto',
          random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
          subsample=0.3, tree_method='exact', validate_parameters=1,
          verbosity=None)
model3.fit(X_train,Y_train)
p3=model3.predict(X_test)
print(classification_report(p3, Y_test))
```

	precision	recall	f1-score	support
0	0.94	0.87	0.91	166
1	0.86	0.93	0.89	136
accuracy			0.90	302
macro avg	0.90	0.90	0.90	302
weighted avg	0.90	0.90	0.90	302

For XGB Classifier model has best precision, recall and f1-scores and also the difference 0 and 1 is marginal as well.

Model 4:

Creating Random Forest Classifier and defining parameter for the model to run hyperparameter tuning using grid search CV.

```
rfc=RandomForestClassifier()
paras={
      'max_depth':[1,2,3,4,5],
    'min_samples_split':[1,2,3,4],
    'max_leaf_nodes':[10,20,30,40,50],
    'min_samples_leaf':[100,200,300,400],
    'n_estimators':[100,200,300,400],
    'max_samples': [0.1,0.2,0.3,0.4],
    'max_features':[15,20,25,30,34]
}
rfc_gs= gs(estimator =rfc, param_grid=paras,cv=10, n_jobs=10)
rfc_gs.fit(X_train,Y_train)
rfc_gs.best_score_
```

```
0.7998415977961433
```

```
rfc_gs.best_estimator_
```

```
RandomForestClassifier(max_depth=1, max_features=15, max_leaf_nodes=40,
                       max_samples=0.3, min_samples_leaf=100,
                       min_samples_split=4, n_estimators=200)
```

```
model4=RandomForestClassifier(max_depth=1, max_features=15, max_leaf_nodes=40,
                       max_samples=0.3, min_samples_leaf=100,
                       min_samples_split=4, n_estimators=200)
model4.fit(X_train,Y_train)
p4=model4.predict(X_test)
print(classification_report(p4, Y_test))
```

```
              precision    recall  f1-score   support

           0       0.88      0.78      0.83       173
           1       0.74      0.85      0.79       129

    accuracy                           0.81       302
   macro avg       0.81      0.82      0.81       302
weighted avg       0.82      0.81      0.81       302
```

Random Forest Classifier has good scores compared to Logistic Regression and KNeighbor Classifier but has fallen short of XGB Classifier.

Before selecting desired model for our dataset let's first cross verify our model with ROC_AUC_SCORE and ROC_CURVE to pick the best of the lot.

ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes.

```
false_positive_rate1, true_positive_rate1, threshold1 = roc_curve(Y_test, p1)

false_positive_rate2, true_positive_rate2, threshold2 = roc_curve(Y_test, p2)

false_positive_rate3, true_positive_rate3, threshold3 = roc_curve(Y_test, p3)

false_positive_rate4, true_positive_rate4, threshold4 = roc_curve(Y_test, p4)



print('roc_auc_score for Logistic Regression: ', roc_auc_score(Y_test, p1))

print('roc_auc_score for KNeighbors Classifier: ', roc_auc_score(Y_test, p2))

print('roc_auc_score for XGB Classifier: ', roc_auc_score(Y_test, p3))

print('roc_auc_score for Random Forest Classifier: ', roc_auc_score(Y_test, p4))
```

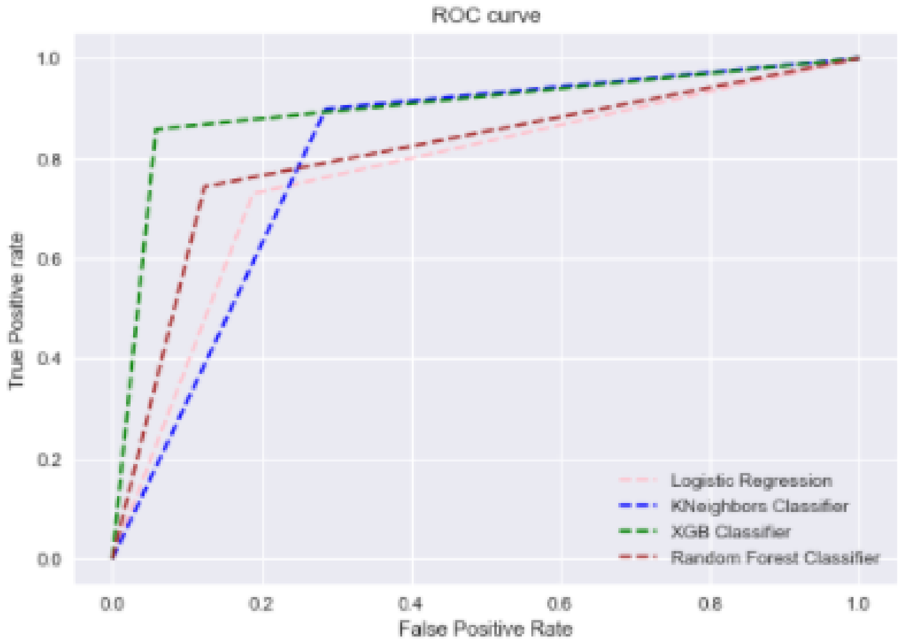
```
roc_auc_score for Logistic Regression:  0.7707090207090208
roc_auc_score for KNeighbors Classifier:  0.8064671814671815
roc_auc_score for XGB Classifier:  0.8998332748332749
roc_auc_score for Random Forest Classifier:  0.8099333099333099
```

XGB Classifier has best roc_auc_score of all the model I have created so far.

```
plt.style.use('seaborn')

plt.plot(false_positive_rate1, true_positive_rate1, linestyle='--', color='pink', label='Logistic Regression')

plt.plot(false_positive_rate2, true_positive_rate2, linestyle='--', color='blue', label='KNeighbors Classifier')

plt.plot(false_positive_rate3, true_positive_rate3, linestyle='--',color='green', label='XGB Classifier')

plt.plot(false_positive_rate4, true_positive_rate4, linestyle='--',color='brown', label='Random Forest Classifier')

plt.title('ROC curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive rate')

plt.legend(loc='best')

plt.savefig('ROC',dpi=300)
```

```
plt.show()
```

Output:



From the above graph we can say that XGB Classifier has the best ROC_CURVE therefore, XGB Classifier is the best suited model for our dataset. And finally, I will be using pickle to save my model.

```
#saving the model
XGB_classifier_auto= pickle.dumps(model3)
```

Conclusion:

To create a good machining model, one must have good source of knowledge over the dataset that they work on as it gives us better insight about how to handle data in the dataset and extract good information from that data using data analysis which in the end helps us in achieving clean dataset. One can customize the data and model accordingly as how one approach in dealing with problem statement therefore every individual has a unique block of code for the same problem outcome.

Thank you for investing your valuable time in reading my article, have a great day ahead!