**ML and Python | Credit Card Fraud Detection**

The challenge is to recognize fraudulent credit card transactions so that the customers of credit card companies are not charged for items that they did not purchase.

**Main challenges involved in credit card fraud detection are:**

1. Enormous Data is processed every day and the model build must be fast enough to respond to the scam in time.
2. Imbalanced Data i.e most of the transactions *(99.8%)* are not fraudulent which makes it really hard for detecting the fraudulent ones
3. Data availability as the data is mostly private.
4. Misclassified Data can be another major issue, as not every fraudulent transaction is caught and reported.
5. Adaptive techniques used against the model by the scammers.

**Challenges faced and tackled**

1. The model used must be simple and fast enough to detect the anomaly and classify it as a fraudulent transaction as quickly as possible.
2. Imbalance can be dealt with by properly using some methods which we will talk about in the next paragraph
3. For protecting the privacy of the user the dimensionality of the data can be reduced.
4. A more trustworthy source must be taken which double-check the data, at least for training the model.
5. We can make the model simple and interpretable so that when the scammer adapts to it with just some tweaks we can have a new model up and running to deploy.

**Code : Importing all the necessary Libraries**

|  |
| --- |
| *# import the necessary packages*  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from matplotlib import gridspec |

**code : Loading the Data and understanding the data**

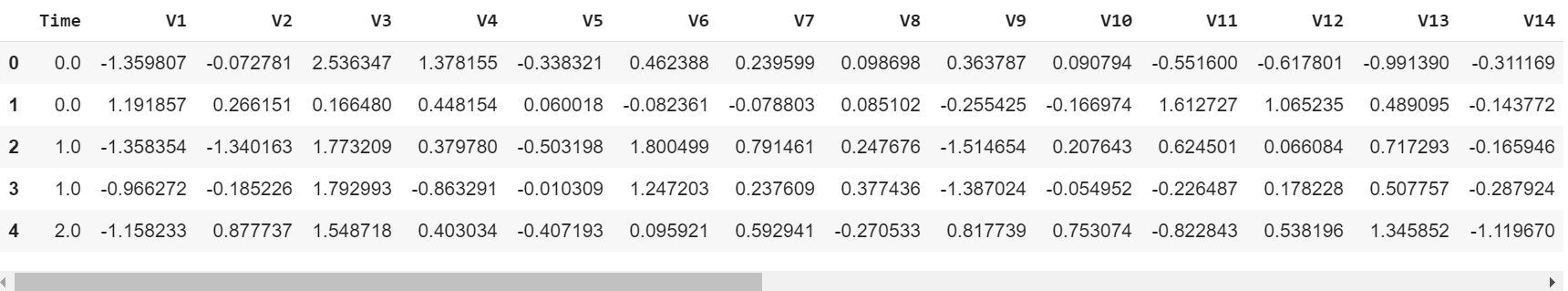
*# Load the dataset from the csv file using pandas*

*# best way is to mount the drive on colab and*

*# copy the path for the csv file*

dataframe = pd.read\_csv("/kaggle/input/creditcard/creditcard.csv")

dataframe.head()



**Code : Describing the Data**

|  |
| --- |
| *# Print the shape of the data*  *# data = data.sample(frac = 0.1, random\_state = 48)*  print(dataframe.shape)  print(dataframe.describe()) |

**Output :**

(284807, 31)

Time V1 V2 V3 V4 \

count 284807.000000 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05

mean 94813.859575 1.168375e-15 3.416908e-16 -1.379537e-15 2.074095e-15

std 47488.145955 1.958696e+00 1.651309e+00 1.516255e+00 1.415869e+00

min 0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00

25% 54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01

50% 84692.000000 1.810880e-02 6.548556e-02 1.798463e-01 -1.984653e-02

75% 139320.500000 1.315642e+00 8.037239e-01 1.027196e+00 7.433413e-01

max 172792.000000 2.454930e+00 2.205773e+01 9.382558e+00 1.687534e+01

V5 V6 V7 V8 V9 \

count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05

mean 9.604066e-16 1.487313e-15 -5.556467e-16 1.213481e-16 -2.406331e-15

std 1.380247e+00 1.332271e+00 1.237094e+00 1.194353e+00 1.098632e+00

min -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01

25% -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01

50% -5.433583e-02 -2.741871e-01 4.010308e-02 2.235804e-02 -5.142873e-02

75% 6.119264e-01 3.985649e-01 5.704361e-01 3.273459e-01 5.971390e-01

max 3.480167e+01 7.330163e+01 1.205895e+02 2.000721e+01 1.559499e+01

... V21 V22 V23 V24 \

count ... 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05

mean ... 1.654067e-16 -3.568593e-16 2.578648e-16 4.473266e-15

std ... 7.345240e-01 7.257016e-01 6.244603e-01 6.056471e-01

min ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00

25% ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01

50% ... -2.945017e-02 6.781943e-03 -1.119293e-02 4.097606e-02

75% ... 1.863772e-01 5.285536e-01 1.476421e-01 4.395266e-01

max ... 2.720284e+01 1.050309e+01 2.252841e+01 4.584549e+00

V25 V26 V27 V28 Amount \

count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 284807.000000

mean 5.340915e-16 1.683437e-15 -3.660091e-16 -1.227390e-16 88.349619

std 5.212781e-01 4.822270e-01 4.036325e-01 3.300833e-01 250.120109

min -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01 0.000000

25% -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02 5.600000

50% 1.659350e-02 -5.213911e-02 1.342146e-03 1.124383e-02 22.000000

75% 3.507156e-01 2.409522e-01 9.104512e-02 7.827995e-02 77.165000

max 7.519589e+00 3.517346e+00 3.161220e+01 3.384781e+01 25691.160000

Class

count 284807.000000

mean 0.001727

std 0.041527

min 0.000000

25% 0.000000

50% 0.000000

75% 0.000000

max 1.000000

[8 rows x 31 columns]

**Code : Imbalance in the data** Time to explain the data we are dealing with.

|  |
| --- |
| *# Determine number of fraud cases in dataset*  fraud = dataframe[dataframe['Class'] == 1]  valid = dataframe[dataframe['Class'] == 0]  outlierFraction = len(fraud)/float(len(valid))  print(outlierFraction)  print('Fraud Cases: **{}**'.format(len(dataframe[dataframe['Class'] == 1])))  print('Valid Transactions: **{}**'.format(len(dataframe[dataframe['Class'] == 0]))) |

**OUTPUT**

0.0017304750013189597

Fraud Cases: 492

Valid Transactions: 284315

Only *0.17%* fraudulent transaction out all the transactions. The data is highly Unbalanced. Lets first apply our models without balancing it and if we don’t get a good accuracy then we can find a way to balance this dataset. But first, let’s implement the model without it and will balance the data only if needed.

**Code : Print the amount details for Fraudulent Transaction**

|  |
| --- |
| print('Amount details of the fraudulent transaction')  fraud.Amount.describe() |

**Output :**

Amount details of the fraudulent transaction

count 492.000000

mean 122.211321

std 256.683288

min 0.000000

25% 1.000000

50% 9.250000

75% 105.890000

max 2125.870000

Name: Amount, dtype: float64

**Code : Print the amount details for Normal Transaction**

|  |
| --- |
| print('details of valid transaction')  valid.Amount.describe() |

**Output :**

Amount details of valid transaction

count 284315.000000

mean 88.291022

std 250.105092

min 0.000000

25% 5.650000

50% 22.000000

75% 77.050000

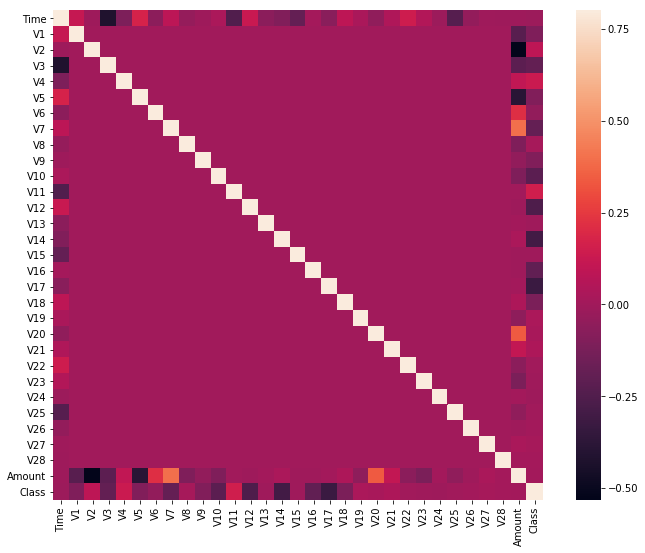
max 25691.160000

Name: Amount, dtype: float64

The average Money transaction for the fraudulent ones is more.

**Code : Plotting the Correlation Matrix** The correlation matrix graphically gives us an idea of how features correlate with each other and can help us predict what are the features that are most relevant for the prediction.

|  |
| --- |
| *# Correlation matrix*  corrmat = dataframe.corr()  fig = plt.figure(figsize = (10, 8))  sns.heatmap(corrmat, vmax = .8, square = True)  plt.show() |

In the HeatMap we can clearly see that most of the features do not correlate to other features but there are some features that either has a positive or a negative correlation with each other. For example, *V2* and *V5* are highly negatively correlated with the feature called *Amount*. We also see some correlation with *V20* and *Amount*. This gives us a deeper understanding of the Data available to us.

**Code : Separating the X and the Y values** Dividing the data into inputs parameters and outputs value format

|  |
| --- |
| *# dividing the X and the Y from the dataset*  X = dataframe.drop(['Class'], axis = 1)  Y = dataframe["Class"]  print(X.shape)  print(Y.shape)  *# getting just the values for the sake of processing*  *# (its a numpy array with no columns)*  xData = X.values  yData = Y.values |

**Output :**

(284807, 30)

(284807, )

**Training and Testing Data Bifurcation**

 Dividing the dataset into two main groups. One for training the model and the other for Testing our trained model’s performance.

|  |
| --- |
| *# Using Scikit-learn to split data into training and testing sets*  from sklearn.model\_selection import train\_test\_split  *# Split the data into training and testing sets*  xTrain, xTest, yTrain, yTest = train\_test\_split(  xData, yData, test\_size = 0.2, random\_state = 42) |

**Code : Building a Random Forest Model using scikit learn**

|  |
| --- |
| *# Building the Random Forest Classifier (RANDOM FOREST)*  from sklearn.ensemble import RandomForestClassifier  *# random forest model creation*  rfc = RandomForestClassifier()  rfc.fit(xTrain, yTrain)  *# predictions*  yPred = rfc.predict(xTest) |

**Code : Building all kinds of evaluating parameters**

|  |
| --- |
| *# Evaluating the classifier*  *# printing every score of the classifier*  *# scoring in anything*  from sklearn.metrics import classification\_report, accuracy\_score  from sklearn.metrics import precision\_score, recall\_score  from sklearn.metrics import f1\_score, matthews\_corrcoef  from sklearn.metrics import confusion\_matrix  n\_outliers = len(fraud)  n\_errors = (yPred != yTest).sum()  print("The model used is Random Forest classifier")  acc = accuracy\_score(yTest, yPred)  print("The accuracy is **{}**".format(acc))  prec = precision\_score(yTest, yPred)  print("The precision is **{}**".format(prec))  rec = recall\_score(yTest, yPred)  print("The recall is **{}**".format(rec))  f1 = f1\_score(yTest, yPred)  print("The F1-Score is **{}**".format(f1))  MCC = matthews\_corrcoef(yTest, yPred)  print("The Matthews correlation coefficient is**{}**".format(MCC)) |

**Output :**

The model used is Random Forest classifier

The accuracy is 0.9995611109160493

The precision is 0.9866666666666667

The recall is 0.7551020408163265

The F1-Score is 0.8554913294797689

The Matthews correlation coefficient is0.8629589216367891

**Code : Visualizing the Confusion Matrix**

*# printing the confusion matrix*

LABELS = ['Normal', 'Fraud']

conf\_matrix = confusion\_matrix(yTest, yPred)

plt.figure(figsize =(12, 12))

sns.heatmap(conf\_matrix, xticklabels = LABELS,

yticklabels = LABELS, annot = True, fmt ="d");

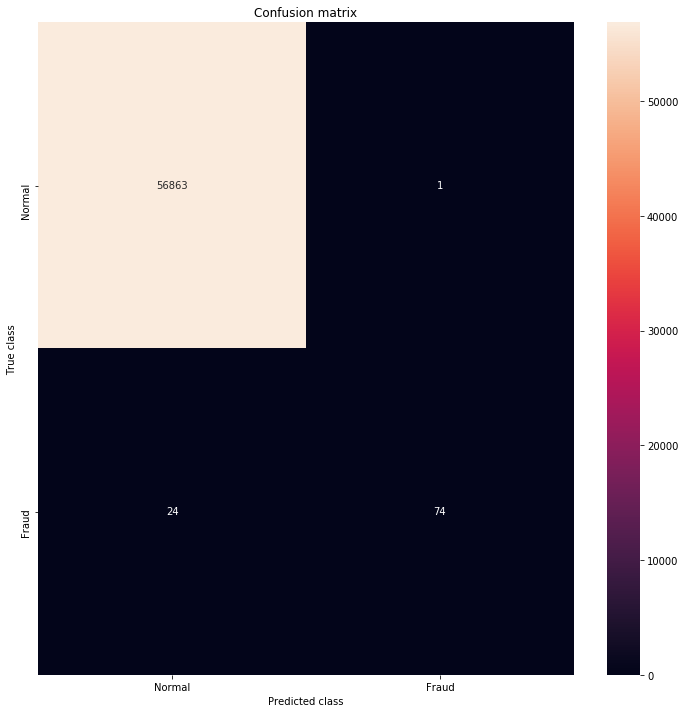
plt.title("Confusion matrix")

plt.ylabel('True class')

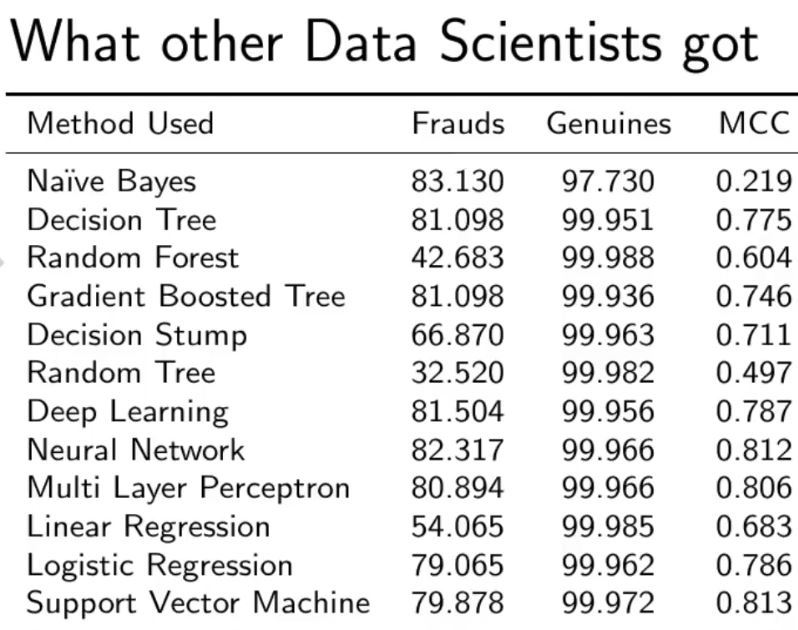
plt.xlabel('Predicted class')

plt.show()

**Output :**



Comparison with other algorithms without dealing with the imbalancing of the data

.

RESULT:

Random Forest Model we are getting a better result even for the recall which is the most tricky part.