Credit Card Fraud Detection

Name:Dipranajan Gupta

213011003

Problem Statement:

For many banks, retaining high profitable customers is the number one business goal. Banking fraud, however, poses a significant threat to this goal for different banks. In terms of substantial financial losses, trust and credibility, this is a concerning issue to both banks and customers alike.

In the banking industry, credit card fraud detection using machine learning is not only a trend but a necessity for them to put proactive monitoring and fraud prevention mechanisms in place. Machine learning is helping these institutions to reduce time-consuming manual reviews, costly chargebacks and fees as well as denials of legitimate transactions.

In this project we will detect fraudulent credit card transactions with the help of Machine learning models. We will analyse customer-level data that has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group.

Data Understanding :

The data set includes credit card transactions made by European cardholders over a period of two days in September 2013. Out of a total of 2,84,807 transactions, 492 were fraudulent. This data set is highly unbalanced, with the positive class (frauds) accounting for 0.172% of the total transactions. The data set has also been modified with principal component analysis (PCA) to maintain confidentiality. Apart from ‘time’ and ‘amount’, all the other features (V1, V2, V3, up to V28) are the principal components obtained using PCA. The feature 'time' contains the seconds elapsed between the first transaction in the data set and the subsequent transactions. The feature 'amount' is the transaction amount. The feature 'class' represents class labelling, and it takes the value of 1 in cases of fraud and 0 in others.

Table of Contents

Double-click (or enter) to edit

Importing Dependencies

```
# Importing the libraries
import numpy as np
import pandas as pd
import time

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from scipy import stats
from scipy.stats import norm, skew
from scipy.special import boxcox1p
from scipy.stats import boxcox_normmax

from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

import sklearn
from sklearn import metrics
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.metrics import average_precision_score, precision_recall_curve

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

from sklearn.linear_model import Ridge, Lasso, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from xgboost import XGBClassifier
from xgboost import plot_importance
from sklearn.ensemble import AdaBoostClassifier

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

Exploratory data analysis

```
# Mounting the google drive
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
# Loading the data
df = pd.read_csv('gdrive/MyDrive/Colab Notebooks/creditcard.csv')
# df = pd.read_csv('./data/creditcard.csv')
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.09869
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.08510
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.24767
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.37743
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.27053

```
# Checking the shape
df.shape
```

```
(284807, 31)
```

```
# Checking the datatypes and null/non-null distribution
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
```

```
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

```
# Checking distribution of numerical values in the dataset
df.describe()
```

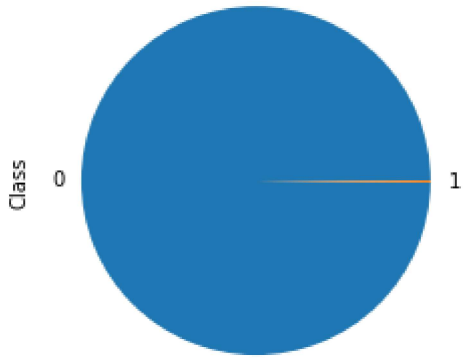
```
Time            V1            V2            V3            V4
```

```
# Checking the class distribution of the target variable
df['Class'].value_counts()
```

```
0    284315
1       492
Name: Class, dtype: int64
```

```
# Checking the class distribution of the target variable in percentage
print((df.groupby('Class')['Class'].count()/df['Class'].count()) *100)
((df.groupby('Class')['Class'].count()/df['Class'].count()) *100).plot.pie()
```

```
Class
0    99.827251
1     0.172749
Name: Class, dtype: float64
<matplotlib.axes._subplots.AxesSubplot at 0x7fb71ea65f90>
```

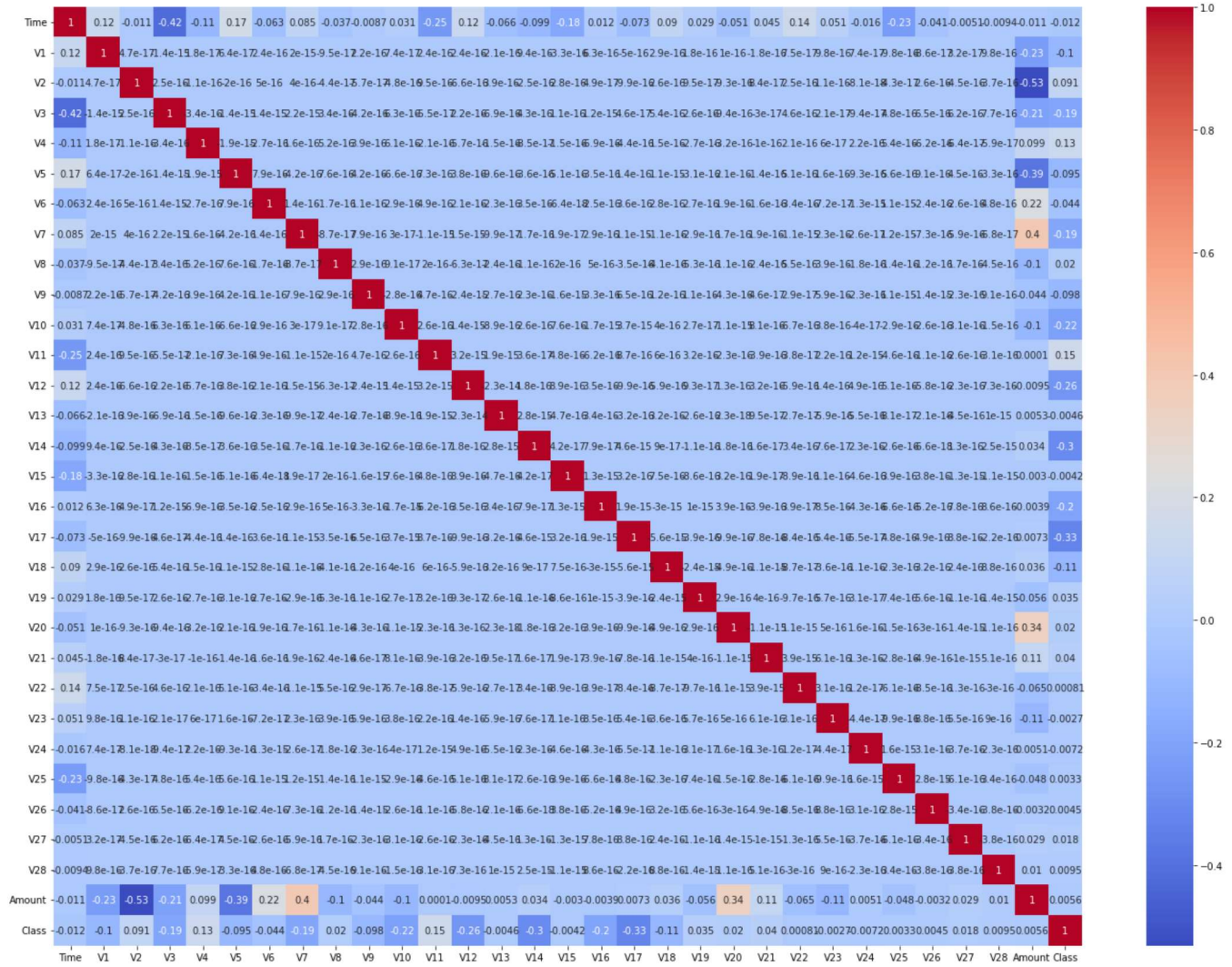



```
# Checking the correlation
corr = df.corr()
corr
```

	Time	V1	V2	V3	V4	V5
Time	1.000000	1.173963e-01	-1.059333e-02	-4.196182e-01	-1.052602e-01	1.730721e-01
V1	0.117396	1.000000e+00	4.697350e-17	-1.424390e-15	1.755316e-17	6.391162e-17
V2	-0.010593	4.697350e-17	1.000000e+00	2.512175e-16	-1.126388e-16	-2.039868e-16
V3	-0.419618	-1.424390e-15	2.512175e-16	1.000000e+00	-3.416910e-16	-1.436514e-15
V4	-0.105260	1.755316e-17	-1.126388e-16	-3.416910e-16	1.000000e+00	-1.940929e-15
V5	0.173072	6.391162e-17	-2.039868e-16	-1.436514e-15	-1.940929e-15	1.000000e+00
V6	-0.063016	2.398071e-16	5.024680e-16	1.431581e-15	-2.712659e-16	7.926364e-16
V7	0.084714	1.991550e-15	3.966486e-16	2.168574e-15	1.556330e-16	-4.209851e-16
V8	-0.036949	-9.490675e-17	-4.413984e-17	3.433113e-16	5.195643e-16	7.589187e-16
V9	-0.008660	2.169581e-16	-5.728718e-17	-4.233770e-16	3.859585e-16	4.205206e-16
V10	0.030617	7.433820e-17	-4.782388e-16	6.289267e-16	6.055490e-16	-6.601716e-16
V11	-0.247689	2.438580e-16	9.468995e-16	-5.501758e-17	-2.083600e-16	7.342759e-16
V12	0.124348	2.422086e-16	-6.588252e-16	2.206522e-16	-5.657963e-16	3.761033e-16
V13	0.065902	-2.115458e-	3.854521e-16	-6.883375e-	-1.506129e-	-9.578659e

```
# Checking the correlation in heatmap
plt.figure(figsize=(24,18))

sns.heatmap(corr, cmap="coolwarm", annot=True)
plt.show()
```



Here we will observe the distribution of our classes

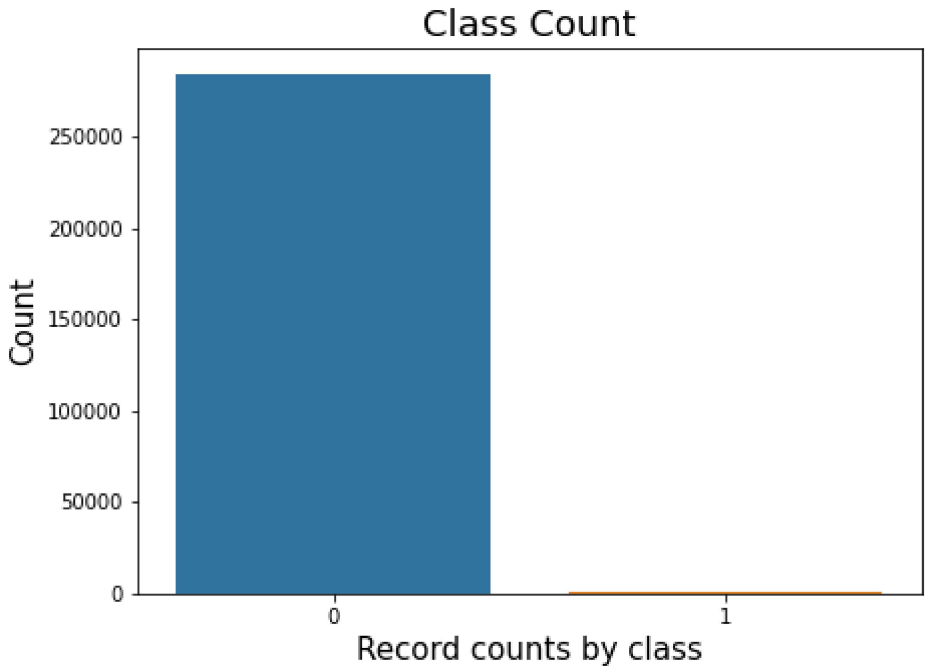
```
# Checking the % distribution of normal vs fraud
classes=df['Class'].value_counts()
normal_share=classes[0]/df['Class'].count()*100
```

```
fraud_share=classes[1]/df['Class'].count()*100

print(normal_share)
print(fraud_share)
```

```
99.82725143693798
0.1727485630620034
```

```
# Create a bar plot for the number and percentage of fraudulent vs non-fraudulent transcation
plt.figure(figsize=(7,5))
sns.countplot(df['Class'])
plt.title("Class Count", fontsize=18)
plt.xlabel("Record counts by class", fontsize=15)
plt.ylabel("Count", fontsize=15)
plt.show()
```




```
# As time is given in relative fashion, we are using pandas.Timedelta which Represents a dura
Delta_Time = pd.to_timedelta(df['Time'], unit='s')

#Create derived columns Mins and hours
df['Time_Day'] = (Delta_Time.dt.components.days).astype(int)
df['Time_Hour'] = (Delta_Time.dt.components.hours).astype(int)
df['Time_Min'] = (Delta_Time.dt.components.minutes).astype(int)
```

```
# Drop unnecessary columns
# We will drop Time,as we have derived the Day/Hour/Minutes from the time column
df.drop('Time', axis = 1, inplace= True)
# We will keep only derived column hour, as day/minutes might not be very useful
df.drop(['Time_Day', 'Time_Min'], axis = 1, inplace= True)
```

Splitting the data into train & test data

```
# Splitting the dataset into X and y
y= df['Class']
X = df.drop(['Class'], axis=1)
```

```
# Checking some rows of X
X.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8

```
# Checking some rows of y
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Class, dtype: int64
```

```
# Splitting the dataset using train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=100, test_size=0.20)
```

Preserve X_test & y_test to evaluate on the test data once you build the model

```
# Checking the spread of data post split
print(np.sum(y))
print(np.sum(y_train))
print(np.sum(y_test))
```

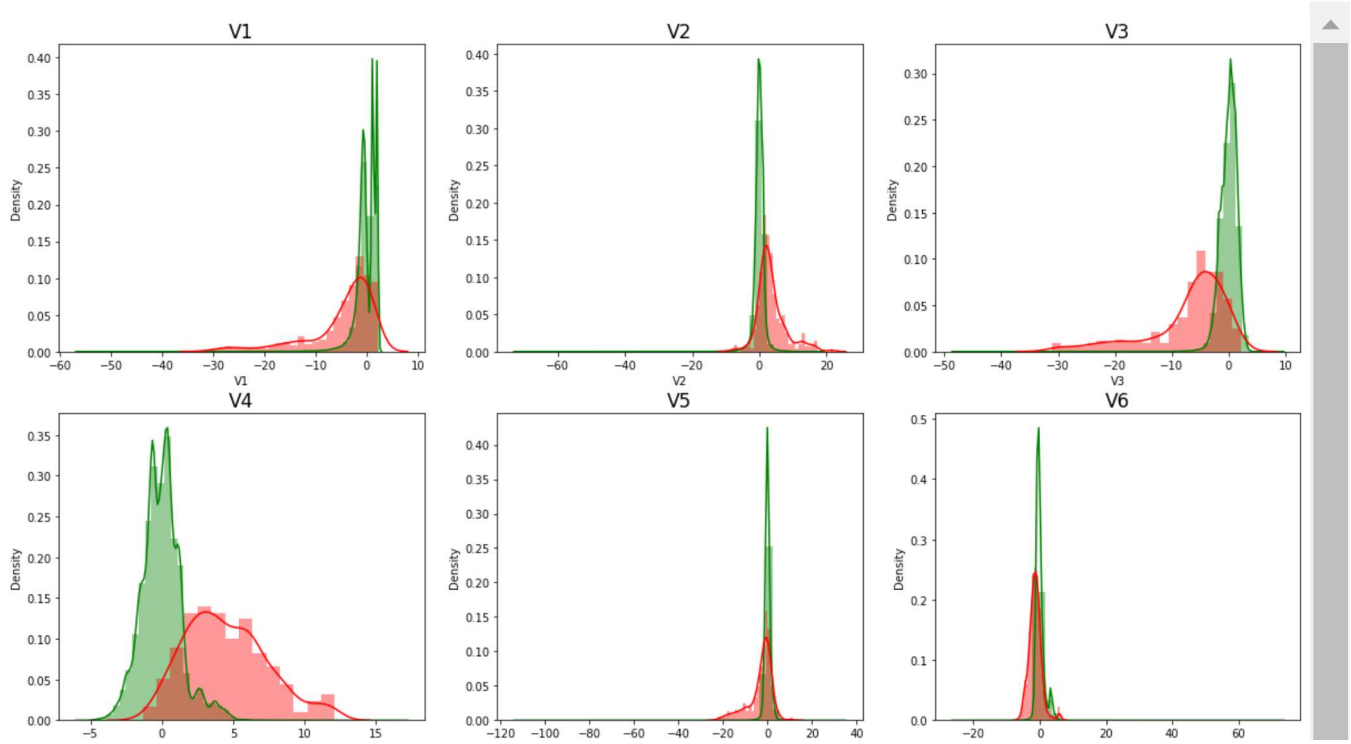
```
492
396
96
```

Plotting the distribution of a variable

```
# Accumulating all the column names under one variable
cols = list(X.columns.values)


# plot the histogram of a variable from the dataset to see the skewness
normal_records = df.Class == 0
fraud_records = df.Class == 1

plt.figure(figsize=(20, 60))
for n, col in enumerate(cols):
  plt.subplot(10,3,n+1)
  sns.distplot(X[col][normal_records], color='green')
  sns.distplot(X[col][fraud_records], color='red')
  plt.title(col, fontsize=17)
plt.show()
```



Model Building

```
#Create a dataframe to store results
df_Results = pd.DataFrame(columns=['Methodology','Model','Accuracy','roc_value','threshold'])
```

```
# Created a common function to plot confusion matrix
def Plot_confusion_matrix(y_test, pred_test):
  cm = confusion_matrix(y_test, pred_test)
  plt.clf()
  plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Accent)
  categoryNames = ['Non-Fraudalent','Fraudalent']
  plt.title('Confusion Matrix - Test Data')
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  ticks = np.arange(len(categoryNames))
  plt.xticks(ticks, categoryNames, rotation=45)
  plt.yticks(ticks, categoryNames)
  s = [['TN','FP'], ['FN', 'TP']]

  for i in range(2):
      for j in range(2):
          plt.text(j,i, str(s[i][j])+" = "+str(cm[i][j]),fontsize=12)
  plt.show()
```

```
# # Created a common function to fit and predict on a Logistic Regression model for both L1 a
def buildAndRunLogisticModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):

  # Logistic Regression
```

```
from sklearn import linear_model
from sklearn.model_selection import KFold

num_C = list(np.power(10.0, np.arange(-10, 10)))
cv_num = KFold(n_splits=10, shuffle=True, random_state=42)

searchCV_l2 = linear_model.LogisticRegressionCV(
        Cs= num_C
        ,penalty='l2'
        ,scoring='roc_auc'
        ,cv=cv_num
        ,random_state=42
        ,max_iter=10000
        ,fit_intercept=True
        ,solver='newton-cg'
        ,tol=10
    )

searchCV_l1 = linear_model.LogisticRegressionCV(
        Cs=num_C
        ,penalty='l1'
        ,scoring='roc_auc'
        ,cv=cv_num
        ,random_state=42
        ,max_iter=10000
        ,fit_intercept=True
        ,solver='liblinear'
        ,tol=10
    )

searchCV_l1.fit(X_train, y_train)
searchCV_l2.fit(X_train, y_train)
print ('Max auc_roc for l1:', searchCV_l1.scores_[1].mean(axis=0).max())
print ('Max auc_roc for l2:', searchCV_l2.scores_[1].mean(axis=0).max())

print("Parameters for l1 regularisations")
print(searchCV_l1.coef_)
print(searchCV_l1.intercept_)
print(searchCV_l1.scores_)

print("Parameters for l2 regularisations")
print(searchCV_l2.coef_)
print(searchCV_l2.intercept_)
print(searchCV_l2.scores_)


#find predicted vallues
y_pred_l1 = searchCV_l1.predict(X_test)
y_pred_l2 = searchCV_l2.predict(X_test)
```

```
    #Find predicted probabilities
    y_pred_probs_l1 = searchCV_l1.predict_proba(X_test)[:,1]
    y_pred_probs_l2 = searchCV_l2.predict_proba(X_test)[:,1]

    # Accuaracy of L2/L1 models
    Accuracy_l2 = metrics.accuracy_score(y_pred=y_pred_l2, y_true=y_test)
    Accuracy_l1 = metrics.accuracy_score(y_pred=y_pred_l1, y_true=y_test)

    print("Accuarcy of Logistic model with l2 regularisation : {0}".format(Accuracy_l2))
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred_l2)
    print("classification Report")
    print(classification_report(y_test, y_pred_l2))

    print("Accuarcy of Logistic model with l1 regularisation : {0}".format(Accuracy_l1))
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred_l1)
    print("classification Report")
    print(classification_report(y_test, y_pred_l1))

    l2_roc_value = roc_auc_score(y_test, y_pred_probs_l2)
    print("l2 roc_value: {0}" .format(l2_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l2)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("l2 threshold: {0}".format(threshold))

    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logistic

    l1_roc_value = roc_auc_score(y_test, y_pred_probs_l1)
    print("l1 roc_value: {0}" .format(l1_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l1)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("l1 threshold: {0}".format(threshold))

    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logistic
    return df_Results


# Created a common function to fit and predict on a KNN model
```

```
def buildAndRunKNNModels(df_Results,Methodology, X_train,y_train, X_test, y_test ):

  #create KNN model and fit the model with train dataset
  knn = KNeighborsClassifier(n_neighbors = 5,n_jobs=16)
  knn.fit(X_train,y_train)
  score = knn.score(X_test,y_test)
  print("model score")
  print(score)

  #Accuracy
  y_pred = knn.predict(X_test)
  KNN_Accuracy = metrics.accuracy_score(y_pred=y_pred, y_true=y_test)
  print("Confusion Matrix")
  Plot_confusion_matrix(y_test, y_pred)
  print("classification Report")
  print(classification_report(y_test, y_pred))


  knn_probs = knn.predict_proba(X_test)[:, 1]

  # Calculate roc auc
  knn_roc_value = roc_auc_score(y_test, knn_probs)
  print("KNN roc_value: {0}" .format(knn_roc_value))
  fpr, tpr, thresholds = metrics.roc_curve(y_test, knn_probs)
  threshold = thresholds[np.argmax(tpr-fpr)]
  print("KNN threshold: {0}".format(threshold))

  roc_auc = metrics.auc(fpr, tpr)
  print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
  plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
  plt.legend(loc=4)
  plt.show()

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'KNN','Acc

  return df_Results


# Created a common function to fit and predict on a Tree models for both gini and entropy cri
def buildAndRunTreeModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  #Evaluate Decision Tree model with 'gini' & 'entropy'
  criteria = ['gini', 'entropy']
  scores = {}

  for c in criteria:
      dt = DecisionTreeClassifier(criterion = c, random_state=42)
      dt.fit(X_train, y_train)
      y_pred = dt.predict(X_test)
      test_score = dt.score(X_test, y_test)
      tree_preds = dt.predict_proba(X_test)[:, 1]
      tree_roc_value = roc_auc_score(y_test, tree_preds)
```

```
        scores = test_score
        print(c + " score: {0}" .format(test_score))
        print("Confusion Matrix")
        Plot_confusion_matrix(y_test, y_pred)
        print("classification Report")
        print(classification_report(y_test, y_pred))
        print(c + " tree_roc_value: {0}" .format(tree_roc_value))
        fpr, tpr, thresholds = metrics.roc_curve(y_test, tree_preds)
        threshold = thresholds[np.argmax(tpr-fpr)]
        print("Tree threshold: {0}".format(threshold))
        roc_auc = metrics.auc(fpr, tpr)
        print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
        plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
        plt.legend(loc=4)
        plt.show()

        df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Tree

    return df_Results


# Created a common function to fit and predict on a Random Forest model
def buildAndRunRandomForestModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
    #Evaluate Random Forest model

    # Create the model with 100 trees
    RF_model = RandomForestClassifier(n_estimators=100,
                                bootstrap = True,
                                max_features = 'sqrt', random_state=42)
    # Fit on training data
    RF_model.fit(X_train, y_train)
    RF_test_score = RF_model.score(X_test, y_test)
    RF_model.predict(X_test)

    print('Model Accuracy: {0}'.format(RF_test_score))


    # Actual class predictions
    rf_predictions = RF_model.predict(X_test)

    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, rf_predictions)
    print("classification Report")
    print(classification_report(y_test, rf_predictions))

    # Probabilities for each class
    rf_probs = RF_model.predict_proba(X_test)[:, 1]

    # Calculate roc auc
    roc_value = roc_auc_score(y_test, rf_probs)
```

```
    print("Random Forest roc_value: {0}" .format(roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, rf_probs)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("Random Forest threshold: {0}".format(threshold))
    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Random Fo

    return df_Results


  # Created a common function to fit and predict on a XGBoost model
  def buildAndRunXGBoostModels(df_Results, Methodology,X_train,y_train, X_test, y_test ):
    #Evaluate XGboost model
    XGBmodel = XGBClassifier(random_state=42)
    XGBmodel.fit(X_train, y_train)
    y_pred = XGBmodel.predict(X_test)

    XGB_test_score = XGBmodel.score(X_test, y_test)
    print('Model Accuracy: {0}'.format(XGB_test_score))

    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred)
    print("classification Report")
    print(classification_report(y_test, y_pred))
    # Probabilities for each class
    XGB_probs = XGBmodel.predict_proba(X_test)[:, 1]

    # Calculate roc auc
    XGB_roc_value = roc_auc_score(y_test, XGB_probs)

    print("XGboost roc_value: {0}" .format(XGB_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("XGBoost threshold: {0}".format(threshold))
    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'XGBoost',

    return df_Results
```

```
# Created a common function to fit and predict on a SVM model
def buildAndRunSVMModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  #Evaluate SVM model with sigmoid kernel  model
  from sklearn.svm import SVC
  from sklearn.metrics import accuracy_score
  from sklearn.metrics import roc_auc_score

  clf = SVC(kernel='sigmoid', random_state=42)
  clf.fit(X_train,y_train)
  y_pred_SVM = clf.predict(X_test)
  SVM_Score = accuracy_score(y_test,y_pred_SVM)
  print("accuracy_score : {0}".format(SVM_Score))
  print("Confusion Matrix")
  Plot_confusion_matrix(y_test, y_pred_SVM)
  print("classification Report")
  print(classification_report(y_test, y_pred_SVM))

  # Run classifier
  classifier = SVC(kernel='sigmoid' , probability=True)
  svm_probs = classifier.fit(X_train, y_train).predict_proba(X_test)[:, 1]

  # Calculate roc auc
  roc_value = roc_auc_score(y_test, svm_probs)

  print("SVM roc_value: {0}" .format(roc_value))
  fpr, tpr, thresholds = metrics.roc_curve(y_test, svm_probs)
  threshold = thresholds[np.argmax(tpr-fpr)]
  print("SVM threshold: {0}".format(threshold))
  roc_auc = metrics.auc(fpr, tpr)
  print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
  plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
  plt.legend(loc=4)
  plt.show()

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'SVM','Acc

  return df_Results
```

- Build different models on the imbalanced dataset and see the result

Perform cross validation with RepeatedKFold

```
#Lets perfrom RepeatedKFold and check the results
from sklearn.model_selection import RepeatedKFold
rkf = RepeatedKFold(n_splits=5, n_repeats=10, random_state=None)
# X is the feature set and y is the target
for train_index, test_index in rkf.split(X,y):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
X_train_cv, X_test_cv = X.iloc[train_index], X.iloc[test_index]
y_train_cv, y_test_cv = y.iloc[train_index], y.iloc[test_index]
```

```
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     5      6     12 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    10     11     15 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     4     18     24 ... 2
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3      7     13 ... 2
TRAIN: [     3      4      5 ... 284801 284803 284804] TEST: [     0      1      2 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     7      8      9 ... 2
TRAIN: [     0      1      2 ... 284802 284803 284804] TEST: [     4      6     15 ... 2
TRAIN: [     0      1      3 ... 284804 284805 284806] TEST: [     2     10     11 ... 2
TRAIN: [     2      3      4 ... 284804 284805 284806] TEST: [     0      1     17 ... 2
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3      5     12 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     3      7      8 ... 2
TRAIN: [     0      2      3 ... 284803 284804 284805] TEST: [     1     12     19 ... 2
TRAIN: [     0      1      3 ... 284803 284805 284806] TEST: [     2      4      9 ... 2
TRAIN: [     0      1      2 ... 284802 284804 284806] TEST: [     5      6     11 ... 2
TRAIN: [     1      2      3 ... 284804 284805 284806] TEST: [     0     17     26 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     3      9     28 ... 2
TRAIN: [     1      3      4 ... 284804 284805 284806] TEST: [     0      2      7 ... 2
TRAIN: [     0      2      3 ... 284803 284804 284806] TEST: [     1      6      8 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     4      5     16 ... 2
TRAIN: [     0      1      2 ... 284801 284802 284805] TEST: [    11     13     14 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    12     13     14 ... 2
TRAIN: [     0      2      3 ... 284804 284805 284806] TEST: [     1      5      6 ... 2
TRAIN: [     0      1      2 ... 284803 284804 284806] TEST: [     4      7      8 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    10     11     15 ... 2
TRAIN: [     1      4      5 ... 284801 284802 284805] TEST: [     0      2      3 ... 2
TRAIN: [     0      1      3 ... 284804 284805 284806] TEST: [     2      4      5 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     8     13     15 ... 2
TRAIN: [     0      1      2 ... 284802 284804 284806] TEST: [     9     14     21 ... 2
TRAIN: [     0      2      4 ... 284803 284805 284806] TEST: [     1      3      6 ... 2
TRAIN: [     1      2      3 ... 284803 284804 284805] TEST: [     0     11     12 ... 2
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3     12     15 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     5     22     25 ... 2
TRAIN: [     1      2      3 ... 284804 284805 284806] TEST: [     0     13     18 ... 2
TRAIN: [     0      3      5 ... 284802 284804 284806] TEST: [     1      2      4 ... 2
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     7      8      9 ... 2
TRAIN: [     1      2      3 ... 284803 284804 284806] TEST: [     0      8     24 ... 2
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     9     10     13 ... 2
TRAIN: [     0      3      4 ... 284804 284805 284806] TEST: [     1      2      6 ... 2
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3      5      7 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     4     11     12 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     6      8     20 ... 2
TRAIN: [     0      2      3 ... 284804 284805 284806] TEST: [     1      4      9 ... 2
TRAIN: [     1      4      6 ... 284803 284805 284806] TEST: [     0      2      3 ... 2
TRAIN: [     0      1      2 ... 284803 284804 284806] TEST: [     7     13     14 ... 2
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [    17     30     35 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     6     17     20 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     5     13     15 ... 2
TRAIN: [     0      4      5 ... 284803 284805 284806] TEST: [     1      2      3 ... 2
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    16     26     27 ... 2
TRAIN: [     1      2      3 ... 284802 284803 284804] TEST: [     0      4     11 ... 2
```

```
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results,"RepeatedKFold Cross Validation", X_train_c
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,y_tr
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,y_t
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results,"RepeatedKFold Cross Validation",X_trai
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,y_tr
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.9757741295537155
Max auc_roc for l2: 0.9865741449266722
Parameters for l1 regularisations
[[-0.05664497 -0.12515701 -0.18527504  0.0659936  -0.26440918  0.09351725
   0.1218119  -0.03176011 -0.1478769  -0.14916947 -0.02962538  0.00313734
  -0.13632793 -0.22700063  0.01235601 -0.12010628 -0.23319794  0.03095384
   0.01223133  0.24200061  0.09337734 -0.02531852  0.00942889 -0.01410184
  -0.02038237  0.00027867 -0.06441265 -0.00383413 -0.006582   -0.1164491 ]]
[-2.09682918]
{1: array([[0.5       , 0.5       , 0.5       , 0.5       , 0.60309428,
        0.6149433 , 0.62232097, 0.92291065, 0.93588395, 0.93713444,
        0.92093358, 0.92534087, 0.93549294, 0.93159234, 0.92892502,
        0.92978354, 0.92453112, 0.93841169, 0.9420121 , 0.95249686],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.53975808,
        0.53338235, 0.53864862, 0.87791416, 0.8932588 , 0.91288678,
        0.88595075, 0.88708285, 0.91274826, 0.90356226, 0.90861809,
        0.91879369, 0.89222393, 0.90362352, 0.89662714, 0.96436544],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.48210007,
        0.508047  , 0.54688312, 0.90017981, 0.90894655, 0.90105052,
        0.88396462, 0.88522427, 0.92445031, 0.90128408, 0.92462035,
        0.9370644 , 0.88340369, 0.90887521, 0.9338884 , 0.97205609],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.51246758,
        0.5114712 , 0.53957378, 0.84786022, 0.88933579, 0.89470233,
        0.86801251, 0.87519232, 0.92150327, 0.90097977, 0.9208036 ,
        0.92174992, 0.87490048, 0.91223057, 0.90423877, 0.9843937 ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.52434603,
        0.54919298, 0.5865437 , 0.96401216, 0.96629969, 0.95448441,
        0.94240104, 0.9466059 , 0.96671387, 0.95069871, 0.96742685,
        0.98314652, 0.94989442, 0.96167899, 0.97443219, 0.9906839 ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.5971116 ,
        0.55194576, 0.5509632 , 0.83269809, 0.9053757 , 0.93636863,
        0.88678423, 0.89433939, 0.93330213, 0.92832634, 0.91696567,
        0.92915683, 0.89579125, 0.92146265, 0.90749053, 0.95705473],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.39011846,
        0.43218037, 0.50253858, 0.88560003, 0.92952565, 0.94573802,
        0.93024917, 0.92833601, 0.94012568, 0.94398338, 0.93512857,
        0.93351832, 0.93328761, 0.94986566, 0.91587148, 0.96005135],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.61050167,
        0.61596227, 0.62009918, 0.98031355, 0.98882736, 0.98398104,
        0.97385817, 0.97679426, 0.99323867, 0.98515605, 0.99271092,
        0.99443185, 0.97541082, 0.98967635, 0.99378459, 0.99811463],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.5457621 ,
        0.54152505, 0.53853312, 0.86137613, 0.94791529, 0.9512736 ,
        0.91430407, 0.93248897, 0.94763233, 0.95088918, 0.93643348,
        0.95103911, 0.92464503, 0.95240543, 0.92928398, 0.98735479],
```

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998964	0.969011	0.001423
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998929	0.869289	0.047230
2	RepeatedKFold Cross Validation	KNN	0.999263	0.864827	0.200000

Results for cross validation with RepeatedKFold:

Looking at Accuracy and ROC value we have "Logistic Regression with L2 Regularisation" which has provided best results for cross validation with RepeatedKFold technique

Perform cross validation with StratifiedKFold

```
#Lets perfrom StratifiedKFold and check the results
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, random_state=None)
# X is the feature set and y is the target
for train_index, test_index in skf.split(X,y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train_SKF_cv, X_test_SKF_cv = X.iloc[train_index], X.iloc[test_index]
    y_train_SKF_cv, y_test_SKF_cv = y.iloc[train_index], y.iloc[test_index]
```

```
TRAIN: [ 30473  30496  31002 ... 284804 284805 284806] TEST: [    0     1     2 ... 5701
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [ 30473  30496  31002 ... 1
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [ 81609  82400  83053 ... 1
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [150654 150660 150661 ... 2
TRAIN: [     0      1      2 ... 227866 227867 227868] TEST: [212516 212644 213092 ... 2
```

```
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results,"StratifiedKFold Cross Validation", X_train
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results,"StratifiedKFold Cross Validation",X_train_SKF_c
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )
```

```
#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results,"StratifiedKFold Cross Validation",X_train_SKF_
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results,"StratifiedKFold Cross Validation",X_tr
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"StratifiedKFold Cross Validation",X_train_S
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,"StratifiedKFold Cross Validation",X_train_SKF_c
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.9652106893977332
Max auc_roc for l2: 0.9825816914072499
Parameters for l1 regularisations
[[-0.03660539 -0.1358827  -0.20474594  0.04333662 -0.24034598  0.08416903
   0.12654905 -0.04815306 -0.14258132 -0.1384162  -0.11668372  0.05505155
  -0.15637116 -0.24216184 -0.03137093 -0.09904364 -0.23133976  0.05582481
   0.02122044  0.20503748  0.09013502  0.00148754 -0.03796378 -0.00392139
  -0.14797782 -0.00701001 -0.06594801 -0.00660368 -0.00656465 -0.12981985]]
[-2.0409044]
{1: array([[0.5       , 0.5       , 0.5       , 0.5       , 0.48341269,
        0.44979055, 0.48500892, 0.78098262, 0.9497821 , 0.94525947,
        0.91827811, 0.91446679, 0.94989144, 0.94933682, 0.94178407,
        0.94900653, 0.92481293, 0.94421561, 0.9164553 , 0.97793244],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.54612019,
        0.57709187, 0.60597219, 0.84651057, 0.93343227, 0.93465714,
        0.92629828, 0.92557122, 0.94481541, 0.93298593, 0.95187361,
        0.93810717, 0.92664917, 0.94492021, 0.92500884, 0.95841435],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.58868006,
        0.61887535, 0.63372622, 0.89640565, 0.952319  , 0.95115981,
        0.93861224, 0.94358681, 0.96139027, 0.94896452, 0.9635843 ,
        0.96250675, 0.94059027, 0.96353909, 0.93360628, 0.98182732],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.59753909,
        0.56683014, 0.55944301, 0.80696138, 0.84768477, 0.86696389,
        0.82045338, 0.82698901, 0.88712716, 0.86862794, 0.89226625,
        0.89140345, 0.82803893, 0.89201256, 0.85123391, 0.93335636],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.46424579,
        0.46709088, 0.53359722, 0.85076556, 0.91321611, 0.92052559,
        0.88358815, 0.90359788, 0.93212983, 0.91699858, 0.91795313,
        0.94482596, 0.90084275, 0.93494127, 0.90607695, 0.9475907 ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.48969022,
        0.49824625, 0.57490495, 0.81629523, 0.89704532, 0.90870084,
```

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998964	0.969011	0.001423
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998929	0.869289	0.047230
2	RepeatedKFold Cross Validation	KNN	0.999263	0.864827	0.200000

Results for cross validation with StratifiedKFold:

Looking at the ROC value we have Logistic Regression with L2 Regularisation has provided best results for cross validation with StratifiedKFold technique

	Validation				

Conclusion :

- As the results show Logistic Regression with L2 Regularisation for StratifiedKFold cross validation provided best results

8	StratifiedKFold Cross	Logistic Regression with L2	0.998771	0.983329	0.001572

Proceed with the model which shows the best result

- Apply the best hyperparameter on the model
- Predict on the test dataset

```
# Logistic Regression
from sklearn import linear_model #import the package
from sklearn.model_selection import KFold

num_C = list(np.power(10.0, np.arange(-10, 10)))
cv_num = KFold(n_splits=10, shuffle=True, random_state=42)

clf = linear_model.LogisticRegressionCV(
          Cs= num_C
          ,penalty='l2'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
          ,max_iter=10000
          ,fit_intercept=True
          ,solver='newton-cg'
          ,tol=10
      )

clf.fit(X_train_SKF_cv, y_train_SKF_cv)
```

```
print ('Max auc_roc for l2:', clf.scores_[1].mean(axis=0).max())


print("Parameters for l2 regularisations")
print(clf.coef_)
print(clf.intercept_)
print(clf.scores_)


#find predicted vallues
y_pred_l2 = clf.predict(X_test)


#Find predicted probabilities
y_pred_probs_l2 = clf.predict_proba(X_test)[:,1]


# Accuaracy of L2/L1 models
Accuracy_l2 = metrics.accuracy_score(y_pred=y_pred_l2, y_true=y_test)


print("Accuarcy of Logistic model with l2 regularisation : {0}".format(Accuracy_l2))


from sklearn.metrics import roc_auc_score
l2_roc_value = roc_auc_score(y_test, y_pred_probs_l2)
print("l2 roc_value: {0}" .format(l2_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l2)
threshold = thresholds[np.argmax(tpr-fpr)]
print("l2 threshold: {0}".format(threshold))
```

```
Max auc_roc for l2: 0.9865741449266722
Parameters for l2 regularisations
[[ 8.81955672e-03  4.12852966e-02 -8.73597273e-02  2.31593930e-01
   8.23465655e-02 -5.16810590e-02 -4.00236018e-02 -1.21892419e-01
  -8.37934902e-02 -1.88393098e-01  1.47808325e-01 -2.13768201e-01
  -3.73003674e-02 -3.80938168e-01 -4.79315972e-03 -1.05538339e-01
  -9.33039347e-02 -4.43804397e-03  1.14866801e-02 -7.35149625e-03
   4.44308093e-02  3.04505742e-02 -8.51460452e-03 -1.50983112e-02
  -6.56644541e-03  5.37855175e-03 -8.33077400e-03 -6.17909130e-05
   3.22049047e-04  9.96568363e-03]]
[-7.6233989]
{1: array([[0.53980902, 0.54182146, 0.56634264, 0.74371077, 0.91438765,
        0.95390987, 0.96803901, 0.97816813, 0.96897688, 0.95944719,
        0.95944719, 0.95485347, 0.95485347, 0.95485347, 0.95485347,
        0.95485347, 0.95485347, 0.95485347, 0.95485347, 0.95485347],
       [0.58448142, 0.58599444, 0.60685041, 0.76199495, 0.92335672,
        0.94993447, 0.96479963, 0.96984747, 0.97499254, 0.9766534 ,
        0.97621921, 0.97621921, 0.97621921, 0.97621921, 0.97621921,
        0.97621921, 0.97621921, 0.97621921, 0.97621921, 0.97621921],
       [0.61346624, 0.61476302, 0.63158702, 0.75831428, 0.8961028 ,
        0.93985439, 0.97944884, 0.98794782, 0.98007622, 0.97403205,
        0.97371054, 0.97371054, 0.97371054, 0.97371054, 0.97371054,
```

```
               0.97371054, 0.97371054, 0.97371054, 0.97371054, 0.97371054],
              [0.61268627, 0.61429074, 0.63028167, 0.76311901, 0.9009578 ,
               0.92148495, 0.96233168, 0.98011512, 0.97796851, 0.98070367,
               0.98070367, 0.98070367, 0.98070367, 0.98070367, 0.98070367,
               0.98070367, 0.98070367, 0.98070367, 0.98070367, 0.98070367],
              [0.58405655, 0.58578797, 0.6052037 , 0.7573307 , 0.92823764,
               0.97402797, 0.99266764, 0.99176292, 0.9868733 , 0.98227916,
               0.98227916, 0.98227916, 0.98227916, 0.98227916, 0.98227916,
               0.98227916, 0.98227916, 0.98227916, 0.98227916, 0.98227916],
              [0.59340887, 0.59567696, 0.621504  , 0.78234736, 0.92556162,
               0.95741592, 0.98566669, 0.98682866, 0.98212257, 0.97861291,
               0.97802242, 0.97802242, 0.97802242, 0.97802242, 0.97802242,
               0.97802242, 0.97802242, 0.97802242, 0.97802242, 0.97802242],
              [0.73007433, 0.7318635 , 0.75021658, 0.8662425 , 0.96029462,
               0.9708664 , 0.98480934, 0.98428829, 0.98343451, 0.98508243,
               0.98508243, 0.98508243, 0.98508243, 0.98508243, 0.98508243,
               0.98508243, 0.98508243, 0.98508243, 0.98508243, 0.98508243],
              [0.53912527, 0.54116075, 0.56759131, 0.76047758, 0.94354308,
               0.98492946, 0.99690329, 0.99822171, 0.99787561, 0.99667096,
               0.99667096, 0.99667096, 0.99667096, 0.99667096, 0.99667096,
               0.99667096, 0.99667096, 0.99667096, 0.99667096, 0.99667096],
              [0.60207428, 0.6013821 , 0.62200089, 0.76699866, 0.92923325,
               0.97203443, 0.99246834, 0.99463957, 0.99135228, 0.98242386,
               0.98242386, 0.98242386, 0.98242386, 0.98242386, 0.98242386,
               0.98242386, 0.98242386, 0.98242386, 0.98242386, 0.98242386],
              [0.58410202, 0.58567802, 0.60373999, 0.74236053, 0.89363762,
               0.94870851, 0.98802385, 0.99392176, 0.99312315, 0.99088946,
               0.99088946, 0.99088946, 0.99088946, 0.99088946, 0.99088946,
               0.99088946, 0.99088946, 0.99088946, 0.99088946, 0.99088946]])}
     Accuarcy of Logistic model with l2 regularisation : 0.9988764439450862
     l2 roc_value: 0.9765679037855075
     l2 threshold: 0.0010284453371876342
```

```
# Checking for the coefficient values
clf.coef_
```

```
     array([[ 8.81955672e-03,  4.12852966e-02, -8.73597273e-02,
              2.31593930e-01,  8.23465655e-02, -5.16810590e-02,
             -4.00236018e-02, -1.21892419e-01, -8.37934902e-02,
             -1.88393098e-01,  1.47808325e-01, -2.13768201e-01,
             -3.73003674e-02, -3.80938168e-01, -4.79315972e-03,
             -1.05538339e-01, -9.33039347e-02, -4.43804397e-03,
              1.14866801e-02, -7.35149625e-03,  4.44308093e-02,
              3.04505742e-02, -8.51460452e-03, -1.50983112e-02,
             -6.56644541e-03,  5.37855175e-03, -8.33077400e-03,
             -6.17909130e-05,  3.22049047e-04,  9.96568363e-03]])
```

```
# Creating a dataframe with the coefficient values
coefficients = pd.concat([pd.DataFrame(X.columns),pd.DataFrame(np.transpose(clf.coef_))], axi
coefficients.columns = ['Feature','Importance Coefficient']
```

```
coefficients
```

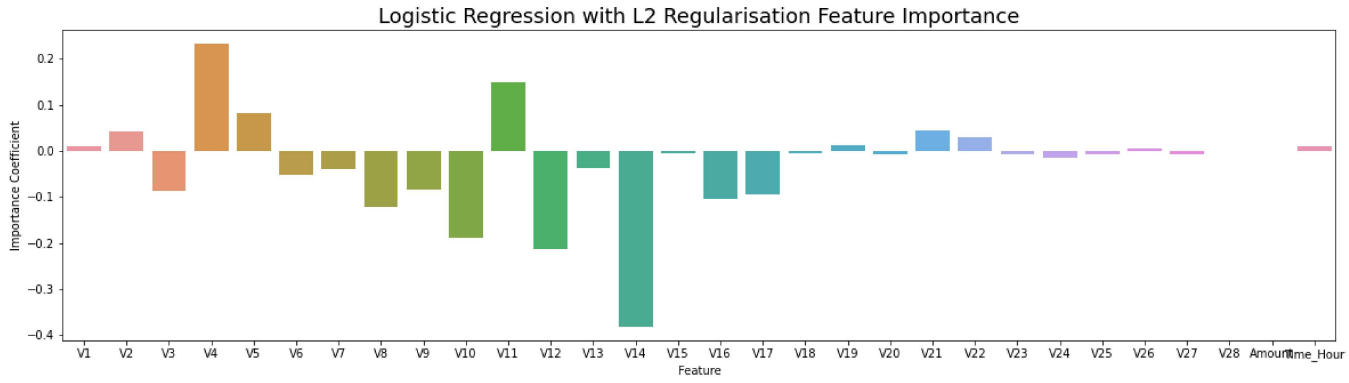
	Feature	Importance Coefficient
0	V1	0.008820
1	V2	0.041285
2	V3	-0.087360
3	V4	0.231594
4	V5	0.082347
5	V6	-0.051681
6	V7	-0.040024
7	V8	-0.121892
8	V9	-0.083793
9	V10	-0.188393
10	V11	0.147808
11	V12	-0.213768
12	V13	-0.037300
13	V14	-0.380938
14	V15	-0.004793
15	V16	-0.105538
16	V17	-0.093304
17	V18	-0.004438
18	V19	0.011487
19	V20	-0.007351
20	V21	0.044431
21	V22	0.030451
22	V23	-0.008515
23	V24	-0.015098
24	V25	-0.006566
25	V26	0.005379
26	V27	-0.008331
27	V28	-0.000062
28	Amount	0.000322
29	Time_Hour	0.009966

Print the important features of the best model to understand the dataset

- This will not give much explanation on the already transformed dataset
- But it will help us in understanding if the dataset is not PCA transformed

```
# Plotting the coefficient values
plt.figure(figsize=(20,5))
sns.barplot(x='Feature', y='Importance Coefficient', data=coefficients)
plt.title("Logistic Regression with L2 Regularisation Feature Importance", fontsize=18)

plt.show()
```



Hence it implies that V4, v5,V11 has + ve importance whereas V10, V12, V14 seems to have -ve impact on the predictaions

Model building with balancing Classes

Perform class balancing with :

- Random Oversampling
- SMOTE
- ADASYN

Oversampling with RandomOverSampler with StratifiedKFold Cross Validation

- We will use Random Oversampling method to handle the class imbalance

```
# Creating the dataset with RandomOverSampler and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn.over_sampling import RandomOverSampler

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    ROS = RandomOverSampler(sampling_strategy=0.5)
    X_over, y_over= ROS.fit_resample(X_train, y_train)

X_over = pd.DataFrame(data=X_over, columns=cols)


Data_Imbalance_Handiling     = "Random Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results , Data_Imbalance_Handiling , X_over, y_over
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results , Data_Imbalance_Handiling,X_over, y_over, X_tes
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results , Data_Imbalance_Handiling,X_over, y_over, X_te
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results , Data_Imbalance_Handiling,X_over, y_ov
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
print('-'*60 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results , Data_Imbalance_Handiling,X_over, y_over, X
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.9865694580981696
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
```

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

	Methodology	Model	Accuracy	roc_value	threshold
	RepeatedKFold Cross	Logistic Regression with L2			

Results for Random Oversampling with StratifiedKFold technique:

Looking at the Accuracy and ROC value we have XGBoost which has provided best results for Random Oversampling and StratifiedKFold technique

validation

Oversampling with SMOTE Oversampling

- We will use SMOTE Oversampling method to handle the class imbalance

```
# Creating dataframe with Smote and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    SMOTE = over_sampling.SMOTE(random_state=0)
    X_train_Smote, y_train_Smote= SMOTE.fit_resample(X_train, y_train)

X_train_Smote = pd.DataFrame(data=X_train_Smote,   columns=cols)
```

11	StratifiedKFold Cross	Tree Model with gini criteria	0.998841	0.826249	1.000000

```
Data_Imbalance_Handiling     = "SMOTE Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y_
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y_trai
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y_tra
```

```
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results, Data_Imbalance_Handiling, X_train_Smot
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y_
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.9930150238550091
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
```

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998964	0.969011	0.001423
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998929	0.869289	0.047230
2	RepeatedKFold Cross Validation	KNN	0.999263	0.864827	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999245	0.874842	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999087	0.874763	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.999491	0.932816	0.010000
6	RepeatedKFold Cross Validation	XGBoost	0.999438	0.970107	0.001447
7	RepeatedKFold Cross Validation	SVM	0.998086	0.421606	0.002741

Results for SMOTE Oversampling with StratifiedKFold:

Looking at Accuracy and ROC value we have XGBoost which has provided best results for SMOTE Oversampling with StratifiedKFold technique

10	StratifiedKFold Cross	KNN	0.999102	0.805746	0.200000

Oversampling with ADASYN Oversampling

- We will use ADASYN Oversampling method to handle the class imbalance

12	Validation	criteria	0.999017	0.821244	1.000000

```
# Creating dataframe with ADASYN and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    ADASYN = over_sampling.ADASYN(random_state=0)
    X_train_ADASYN, y_train_ADASYN= ADASYN.fit_resample(X_train, y_train)

X_train_ADASYN = pd.DataFrame(data=X_train_ADASYN,    columns=cols)
```

```
Data_Imbalance_Handiling     = "ADASYN Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results, Data_Imbalance_Handiling, X_train_ADASYN,
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN, y_trai
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN, y_tra
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results, Data_Imbalance_Handiling,X_train_ADASY
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN, y_
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.9835151915234407
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]])}
Parameters for l2 regularisations
[[ 0.99441334  1.06118409  0.87069584  0.95027684  0.76812921 -1.20275354
  -0.94772947 -0.58397921 -0.36278991 -0.88778254  0.79288369 -1.45342517
  -0.16512685 -1.99287039  0.14389941 -1.0017768  -1.14779789 -0.52970203
   0.3849057  -1.41837451 -0.20546233  0.71955209  0.65930352 -1.1149254
   0.71585263  0.20920088  0.14882922  1.32851471  0.0128743   0.02228753]]
[-5.11988896]
{1: array([[0.610404  , 0.63191849, 0.72150467, 0.87081578, 0.95798817,
        0.97416661, 0.97844939, 0.98148374, 0.98277345, 0.98315652,
        0.98321827, 0.98322068, 0.98322068, 0.98322068, 0.98322068,
        0.98322068, 0.98322068, 0.98322068, 0.98322068, 0.98322068],
       [0.61072194, 0.63192858, 0.72000917, 0.86805623, 0.95677094,
        0.9739103 , 0.97856566, 0.98179915, 0.98313269, 0.98349933,
        0.98355755, 0.98355755, 0.98355755, 0.98355755, 0.98355755,
```

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998964	0.969011	0.001423
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998929	0.869289	0.047230
2	RepeatedKFold Cross Validation	KNN	0.999263	0.864827	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999245	0.874842	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999087	0.874763	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.999491	0.932816	0.010000
6	RepeatedKFold Cross Validation	XGBoost	0.999438	0.970107	0.001447
7	RepeatedKFold Cross Validation	SVM	0.998086	0.421606	0.002741
8	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983329	0.001572
9	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998754	0.889076	0.021087
10	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
11	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998841	0.826249	1.000000
12	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999017	0.821244	1.000000
13	StratifiedKFold Cross Validation	Random Forest	0.999438	0.946472	0.010000
14	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
15	StratifiedKFold Cross	SVM	0.998280	0.598230	0.001694

Results for ADASYN Oversampling with StratifiedKFold:

Looking at Accuracy and ROC value we have XGBoost which has provided best results for ADASYN Oversampling with StratifiedKFold technique

Random Oversampling with

Overall conclusion after running the models on Oversampled data :

Looking at above results it seems XGBOOST model with Random Oversampling with StratifiedKFold CV has provided the best results under the category of all oversampling techniques. So we will try to tune the hyperparameters of this model to get best results.

Hyperparameter Tuning

23 SMOTE Oversampling with Logistic Regression with 0.983533 0.974582 0.394077

HPT - Xgboost Regression

```
# Performing Hyperparameter tuning
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
param_test = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2),
 'n_estimators':range(60,130,150),
 'learning_rate':[0.05,0.1,0.125,0.15,0.2],
 'gamma':[i/10.0 for i in range(0,5)],
 'subsample':[i/10.0 for i in range(7,10)],
 'colsample_bytree':[i/10.0 for i in range(7,10)]
}

gsearch1 = RandomizedSearchCV(estimator = XGBClassifier(base_score=0.5, booster='gbtree', col
              colsample_bynode=1,max_delta_step=0,
              missing=None, n_jobs=-1,
              nthread=None, objective='binary:logistic', random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, verbosity=1),
              param_distributions = param_test, n_iter=5,scoring='roc_auc',n_jobs=-1, cv=5)

gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
     ({'mean_fit_time': array([ 78.0790154 ,  89.69091663, 113.30138698,  41.1210391 ,
              124.47327914]),
       'mean_score_time': array([0.33167305, 0.39689088, 0.43635864, 0.25277925,
     0.39153495]),
       'mean_test_score': array([0.99737639, 0.9998794 , 0.99985979, 0.99902151,
     0.99986951]),
       'param_colsample_bytree': masked_array(data=[0.9, 0.7, 0.7, 0.7, 0.9],
                    mask=[False, False, False, False, False],
              fill_value='?',
                   dtype=object),
       'param_gamma': masked_array(data=[0.2, 0.2, 0.0, 0.4, 0.0],
                    mask=[False, False, False, False, False],
              fill_value='?',
```

```
            dtype=object),
 'param_learning_rate': masked_array(data=[0.15, 0.125, 0.1, 0.2, 0.125],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'param_max_depth': masked_array(data=[5, 7, 9, 3, 9],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'param_min_child_weight': masked_array(data=[3, 5, 3, 3, 1],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'param_n_estimators': masked_array(data=[60, 60, 60, 60, 60],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'param_subsample': masked_array(data=[0.7, 0.8, 0.9, 0.8, 0.7],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'colsample_bytree': 0.9,
   'gamma': 0.2,
   'learning_rate': 0.15,
   'max_depth': 5,
   'min_child_weight': 3,
   'n_estimators': 60,
   'subsample': 0.7},
  {'colsample_bytree': 0.7,
   'gamma': 0.2,
   'learning_rate': 0.125,
   'max_depth': 7,
   'min_child_weight': 5,
   'n_estimators': 60,
   'subsample': 0.8},
  {'colsample_bytree': 0.7,
   'gamma': 0.0,
   'learning_rate': 0.1,
   'max_depth': 9,
   'min_child_weight': 3,
   'n_estimators': 60,
   'subsample': 0.9},
  {'colsample_bytree': 0.7,
   'gamma': 0.4,
   'learning_rate': 0.2,
```

Please note that the hyperparameters found above using RandomizedSearchCV and the hyperparameters used below in creating the final model might be different, the reason being, I have executed the RandomizedSearchCV multiple times to find which set of hyperparameters gives the optimum result and finally used the one below which gave me the best performance.

```
0.0       0.2       0.4       0.6       0.8       1.0
```

```
# Creating XGBoost model with selected hyperparameters
from xgboost import XGBClassifier
```

```
clf = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.7, gamma=0.2,
              learning_rate=0.125, max_delta_step=0, max_depth=7,
              min_child_weight=5, missing=None, n_estimators=60, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=0.8, verbosity=1)

# fit on the dataset
clf.fit(X_over, y_over )
XGB_test_score = clf.score(X_test, y_test)
print('Model Accuracy: {0}'.format(XGB_test_score))

# Probabilities for each class
XGB_probs = clf.predict_proba(X_test)[:, 1]

# Calculate roc auc
XGB_roc_value = roc_auc_score(y_test, XGB_probs)

print("XGboost roc_value: {0}" .format(XGB_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
threshold = thresholds[np.argmax(tpr-fpr)]
print("XGBoost threshold: {0}".format(threshold))
```

```
Model Accuracy: 0.9993328768806727
XGboost roc_value: 0.9815403079438694
XGBoost threshold: 0.01721232570707798
Time taken by model:     117.78114692749501 seconds
```

Print the important features of the best model to understand the dataset

```
imp_var = []
for i in clf.feature_importances_:
    imp_var.append(i)
print('Top var =', imp_var.index(np.sort(clf.feature_importances_)[-1])+1)
print('2nd Top var =', imp_var.index(np.sort(clf.feature_importances_)[-2])+1)
print('3rd Top var =', imp_var.index(np.sort(clf.feature_importances_)[-3])+1)
```

```
Top var = 14
2nd Top var = 17
3rd Top var = 10
```

```
# Calculate roc auc
XGB_roc_value = roc_auc_score(y_test, XGB_probs)

print("XGboost roc_value: {0}" .format(XGB_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
threshold = thresholds[np.argmax(tpr-fpr)]
print("XGBoost threshold: {0}".format(threshold))
```

```
XGboost roc_value: 0.9815403079438694
XGBoost threshold: 0.01721232570707798
```

Conclusion

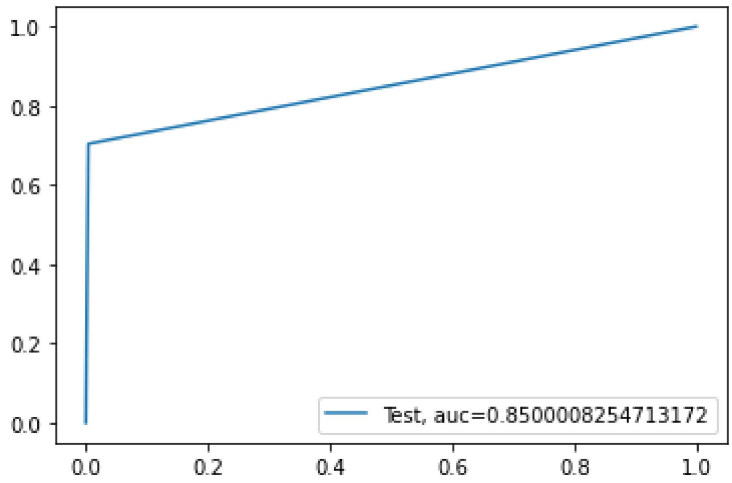
In the oversample cases, of all the models we build found that the XGBOOST model with Random Oversampling with StratifiedKFold CV gave us the best accuracy and ROC on oversampled data. Post that we performed hyperparameter tuning and got the below metrices :

XGboost roc_value: 0.9815403079438694 XGBoost threshold: 0.01721232570707798

However, of all the models we created we found Logistic Regression with L2 Regularisation for StratifiedKFold cross validation (without any oversampling or undersampling) gave us the best result.

```
    accuracy                           1.00     56961
   macro avg       0.61      0.85      0.67     56961
weighted avg       1.00      1.00      1.00     56961

entropy tree_roc_value: 0.8500008254713172
Tree threshold: 1.0
ROC for the test dataset 85.0%
```




```
Time Taken by Model: --- 83.75130772590637 seconds ---
--------------------------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9995259914678464
Confusion Matrix
```