Problem statement:-

The aim of the project is to predict fraudulent credit card transactions using machine learning models. This is crucial from the bank's as well as customer's perspective. The banks cannot afford to lose their customers' money to fraudsters. Every fraud is a loss to the bank as the bank is responsible for the fraud transactions.

The dataset contains transactions made over a period of two days in September 2013 by European credit cardholders. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. We need to take care of the data imbalance while building the model and come up with the best model by trying various algorithms.

Steps:-

The steps are broadly divided into below steps. The sub steps are also listed while we approach each of the steps.

1. Reading, understanding and visualising the data
2. Preparing the data for modelling
3. Building the model
4. Evaluate the model

```
# Importing the basic libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')

pd.set_option('display.max_columns', 500)
```

Reading and understanding the data

```
# Reading the dataset
df = pd.read_csv('./Datasets/creditcard.csv')
df.head()
```

```
   Time        V1        V2        V3        V4        V5        V6
V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388
0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -
0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499
0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203
```

```
0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921
0.592941

         V8        V9       V10       V11       V12       V13
V14  \
0  0.098698  0.363787  0.090794 -0.551600 -0.617801 -0.991390 -
0.311169
1  0.085102 -0.255425 -0.166974  1.612727  1.065235  0.489095 -
0.143772
2  0.247676 -1.514654  0.207643  0.624501  0.066084  0.717293 -
0.165946
3  0.377436 -1.387024 -0.054952 -0.226487  0.178228  0.507757 -
0.287924
4 -0.270533  0.817739  0.753074 -0.822843  0.538196  1.345852 -
1.119670

        V15       V16       V17       V18       V19       V20
V21  \
0  1.468177 -0.470401  0.207971  0.025791  0.403993  0.251412 -
0.018307
1  0.635558  0.463917 -0.114805 -0.183361 -0.145783 -0.069083 -
0.225775
2  2.345865 -2.890083  1.109969 -0.121359 -2.261857  0.524980
0.247998
3 -0.631418 -1.059647 -0.684093  1.965775 -1.232622 -0.208038 -
0.108300
4  0.175121 -0.451449 -0.237033 -0.038195  0.803487  0.408542 -
0.009431

        V22       V23       V24       V25       V26       V27
V28  \
0  0.277838 -0.110474  0.066928  0.128539 -0.189115  0.133558 -
0.021053
1 -0.638672  0.101288 -0.339846  0.167170  0.125895 -0.008983
0.014724
2  0.771679  0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -
0.059752
3  0.005274 -0.190321 -1.175575  0.647376 -0.221929  0.062723
0.061458
4  0.798278 -0.137458  0.141267 -0.206010  0.502292  0.219422
0.215153

   Amount  Class
0  149.62      0
1    2.69      0
2  378.66      0
3  123.50      0
4   69.99      0
```

```
df.shape
```

```
(284807, 31)
```

```
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
```

```
df.describe()
```

```
                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

                V10           V11           V12           V13
V14  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
2.848070e+05
mean   2.239053e-15  1.673327e-15 -1.247012e-15  8.190001e-16
1.207294e-15
std    1.088850e+00  1.020713e+00  9.992014e-01  9.952742e-01
9.585956e-01
min   -2.458826e+01 -4.797473e+00 -1.868371e+01 -5.791881e+00 -
1.921433e+01
25%   -5.354257e-01 -7.624942e-01 -4.055715e-01 -6.485393e-01 -
4.255740e-01
50%   -9.291738e-02 -3.275735e-02  1.400326e-01 -1.356806e-02
5.060132e-02
```

```
75%    4.539234e-01  7.395934e-01  6.182380e-01  6.625050e-01
4.931498e-01
max    2.374514e+01  1.201891e+01  7.848392e+00  7.126883e+00
1.052677e+01

                V15           V16           V17           V18
V19  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
2.848070e+05
mean   4.887456e-15  1.437716e-15 -3.772171e-16  9.564149e-16
1.039917e-15
std    9.153160e-01  8.762529e-01  8.493371e-01  8.381762e-01
8.140405e-01
min   -4.498945e+00 -1.412985e+01 -2.516280e+01 -9.498746e+00 -
7.213527e+00
25%   -5.828843e-01 -4.680368e-01 -4.837483e-01 -4.988498e-01 -
4.562989e-01
50%    4.807155e-02  6.641332e-02 -6.567575e-02 -3.636312e-03
3.734823e-03
75%    6.488208e-01  5.232963e-01  3.996750e-01  5.008067e-01
4.589494e-01
max    8.877742e+00  1.731511e+01  9.253526e+00  5.041069e+00
5.591971e+00

                V20           V21           V22           V23
V24  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
2.848070e+05
mean   6.406204e-16  1.654067e-16 -3.568593e-16  2.578648e-16
4.473266e-15
std    7.709250e-01  7.345240e-01  7.257016e-01  6.244603e-01
6.056471e-01
min   -5.449772e+01 -3.483038e+01 -1.093314e+01 -4.480774e+01 -
2.836627e+00
25%   -2.117214e-01 -2.283949e-01 -5.423504e-01 -1.618463e-01 -
3.545861e-01
50%   -6.248109e-02 -2.945017e-02  6.781943e-03 -1.119293e-02
4.097606e-02
75%    1.330408e-01  1.863772e-01  5.285536e-01  1.476421e-01
4.395266e-01
max    3.942090e+01  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
```

Handling missing values

Handling missing values in columns

```
# Cheking percent of missing values in columns
df_missing_columns =
(round(((df.isnull().sum()/len(df.index))*100),2).to_frame('null')).so
rt_values('null', ascending=False)
df_missing_columns
```

```
        null
Time     0.0
V16      0.0
Amount   0.0
V28      0.0
V27      0.0
V26      0.0
V25      0.0
V24      0.0
V23      0.0
V22      0.0
V21      0.0
V20      0.0
V19      0.0
V18      0.0
V17      0.0
V15      0.0
```

```
V1       0.0
V14      0.0
V13      0.0
V12      0.0
V11      0.0
V10      0.0
V9       0.0
V8       0.0
V7       0.0
V6       0.0
V5       0.0
V4       0.0
V3       0.0
V2       0.0
Class    0.0
```

We can see that there is no missing values in any of the columns. Hence, there is no problem with null values in the entire dataset.

Checking the distribution of the classes

```
classes = df['Class'].value_counts()
classes
```

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

```
normal_share = round((classes[0]/df['Class'].count()*100),2)
normal_share
```

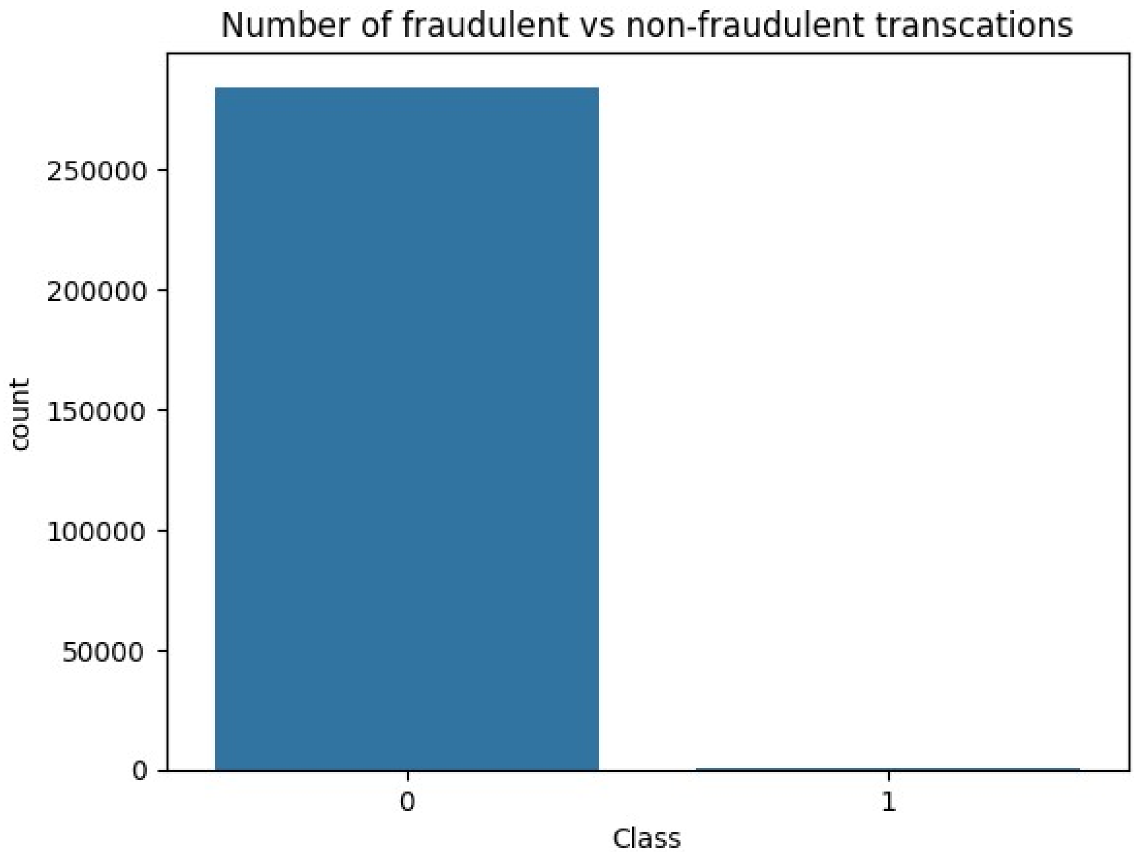
```
99.83
```

```
fraud_share = round((classes[1]/df['Class'].count()*100),2)
fraud_share
```

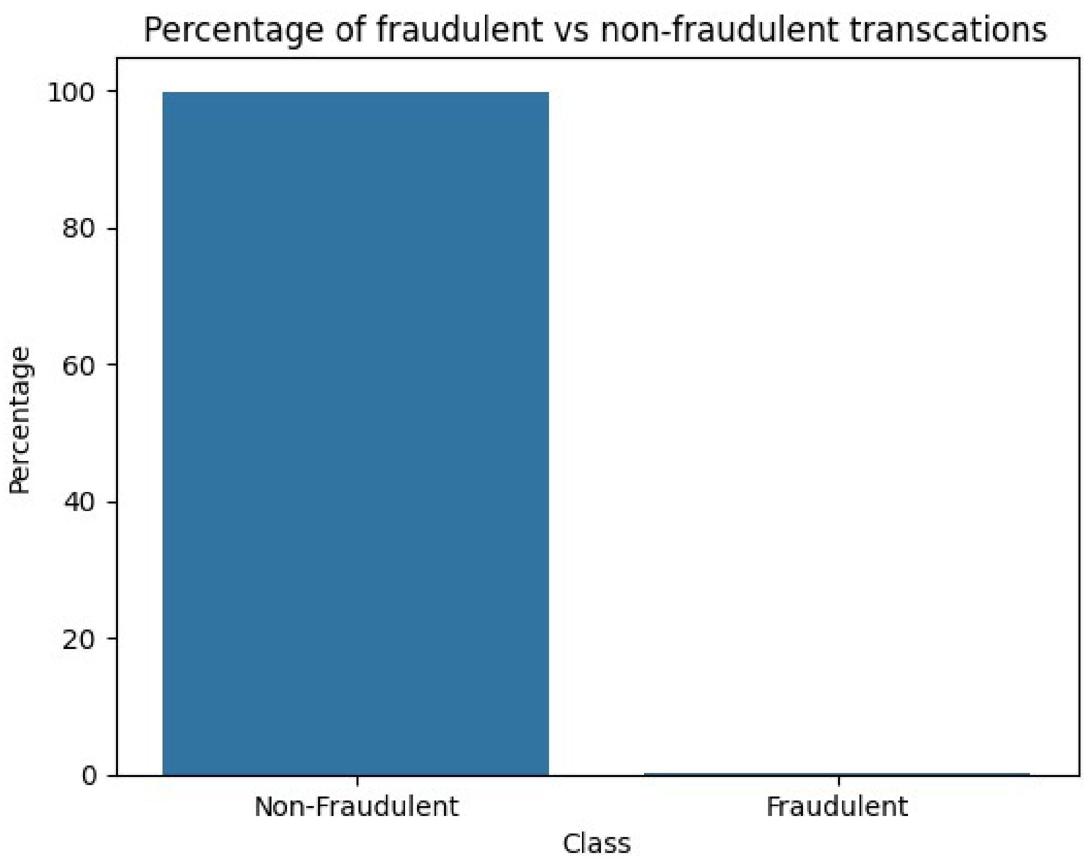
```
0.17
```

We can see that there is only 0.17% frauds. We will take care of the class imbalance later.

```
# Bar plot for the number of fraudulent vs non-fraudulent transcations
sns.countplot(x='Class', data=df)
plt.title('Number of fraudulent vs non-fraudulent transcations')
plt.show()
```




```
# Bar plot for the percentage of fraudulent vs non-fraudulent transcations
fraud_percentage = {'Class':['Non-Fraudulent', 'Fraudulent'], 'Percentage':[normal_share, fraud_share]}
df_fraud_percentage = pd.DataFrame(fraud_percentage)
sns.barplot(x='Class',y='Percentage', data=df_fraud_percentage)
plt.title('Percentage of fraudulent vs non-fraudulent transcations')
plt.show()
```

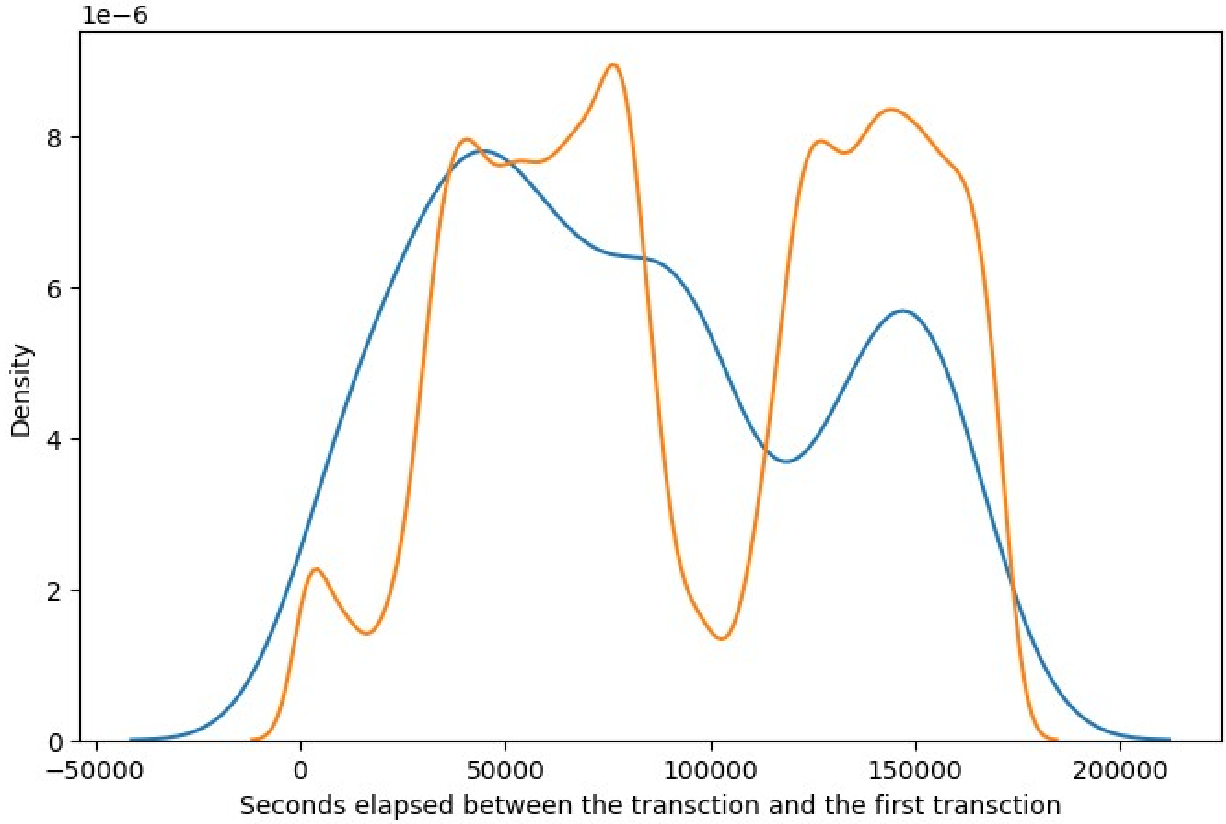


Outliers treatment

We are not performing any outliers treatment for this particular dataset. Because all the columns are already PCA transformed, which assumed that the outlier values are taken care while transforming the data.

Observe the distribution of classes with time

```
# Creating fraudulent dataframe
data_fraud = df[df['Class'] == 1]
# Creating non fraudulent dataframe
data_non_fraud = df[df['Class'] == 0]

# Distribution plot
plt.figure(figsize=(8,5))
ax = sns.distplot(data_fraud['Time'],label='fraudulent',hist=False)
ax = sns.distplot(data_non_fraud['Time'],label='non
fraudulent',hist=False)
ax.set(xlabel='Seconds elapsed between the transction and the first
transction')
plt.show()
```



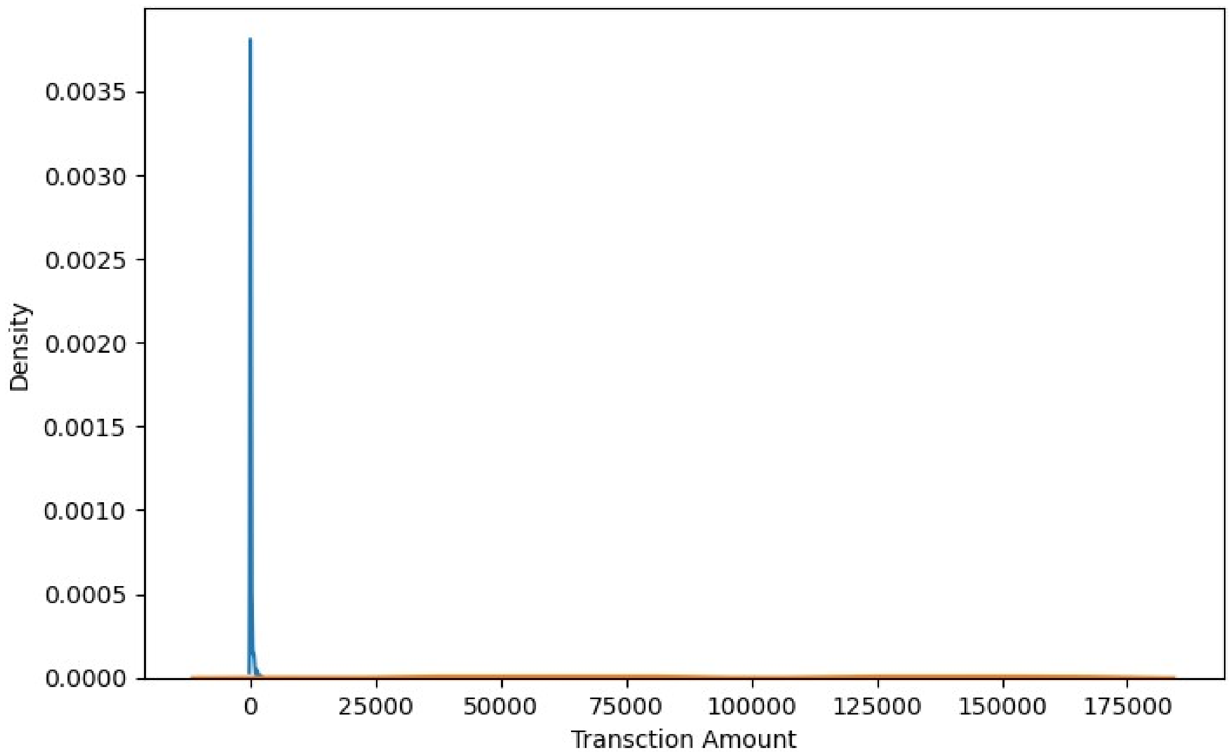
Analysis

We do not see any specific pattern for the fraudulent and non-fraudulent transctions with respect to Time. Hence, we can drop the `Time` column.

```
# Dropping the Time column
df.drop('Time', axis=1, inplace=True)
```

Observe the distribution of classes with amount

```
# Distribution plot
plt.figure(figsize=(8,5))
ax = sns.distplot(data_fraud['Amount'],label='fraudulent',hist=False)
ax = sns.distplot(data_non_fraud['Time'],label='non
fraudulent',hist=False)
ax.set(xlabel='Transction Amount')
plt.show()
```



Analysis

We can see that the fraudulent transctions are mostly densed in the lower range of amount, whereas the non-fraudulent transctions are spreaded throughout low to high range of amount.

Train-Test Split

```
# Import library
from sklearn.model_selection import train_test_split

# Putting feature variables into X
X = df.drop(['Class'], axis=1)

# Putting target variable to y
y = df['Class']

# Splitting data into train and test set 80:20
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.8, test_size=0.2, random_state=100)
```

Feature Scaling

We need to scale only the `Amount` column as all other columns are already scaled by the PCA transformation.

```
# Standardization method
from sklearn.preprocessing import StandardScaler

# Instantiate the Scaler
scaler = StandardScaler()

# Fit the data into scaler and transform
X_train['Amount'] = scaler.fit_transform(X_train[['Amount']])

X_train.head()
```

```
              V1        V2        V3        V4        V5        V6  
V7  \
201788  2.023734 -0.429219 -0.691061 -0.201461 -0.162486  0.283718 -
0.674694
179369 -0.145286  0.736735  0.543226  0.892662  0.350846  0.089253  
0.626708
73138  -3.015846 -1.920606  1.229574  0.721577  1.089918 -0.195727 -
0.462586
208679  1.851980 -1.007445 -1.499762 -0.220770 -0.568376 -1.232633  
0.248573
206534  2.237844 -0.551513 -1.426515 -0.924369 -0.401734 -1.438232 -
0.119942

              V8        V9       V10       V11       V12       V13  
V14  \
201788  0.192230  1.124319 -0.037763  0.308648  0.875063 -0.009562  
0.116038
179369 -0.049137 -0.732566  0.297692  0.519027  0.041275 -0.690783  
0.647121
73138   0.919341 -0.612193 -0.966197  1.106534  1.026421 -0.474229  
0.641488
208679 -0.539483 -0.813368  0.785431 -0.784316  0.673626  1.428269  
0.043937
206534 -0.449263 -0.717258  0.851668 -0.497634 -0.445482  0.324575  
0.125543

             V15       V16       V17       V18       V19       V20  
V21  \
201788  0.086537  0.628337 -0.997868  0.482547  0.576077 -0.171390 -
0.195207
179369  0.526333 -1.098558  0.511739  0.243984  3.349611  0.206709 -
0.124288
73138  -0.430684 -0.631257  0.634633 -0.718062 -0.039929  0.842838  
0.274911
208679 -0.309507 -1.805728 -0.012118  0.377096 -0.658353 -0.196551 -
0.406722
206534  0.266588  0.802640  0.225312 -1.865494  0.621879 -0.045417  
0.050447
```

```
             V22       V23       V24       V25       V26       V27
V28  \
201788 -0.477813  0.340513  0.059174 -0.431015 -0.297028 -0.000063 -
0.046947
179369 -0.263560 -0.110568 -0.434224 -0.509076  0.719784 -0.006357
0.146053
73138  -0.319550  0.212891 -0.268792  0.241190  0.318445 -0.100726 -
0.365257
208679 -0.899081  0.137370  0.075894 -0.244027  0.455618 -0.094066 -
0.031488
206534  0.125601  0.215531 -0.080485 -0.063975 -0.307176 -0.042838 -
0.063872

          Amount
201788 -0.345273
179369 -0.206439
73138   0.358043
208679  0.362400
206534 -0.316109
```

Scaling the test set

We don't fit scaler on the test set. We only transform the test set.

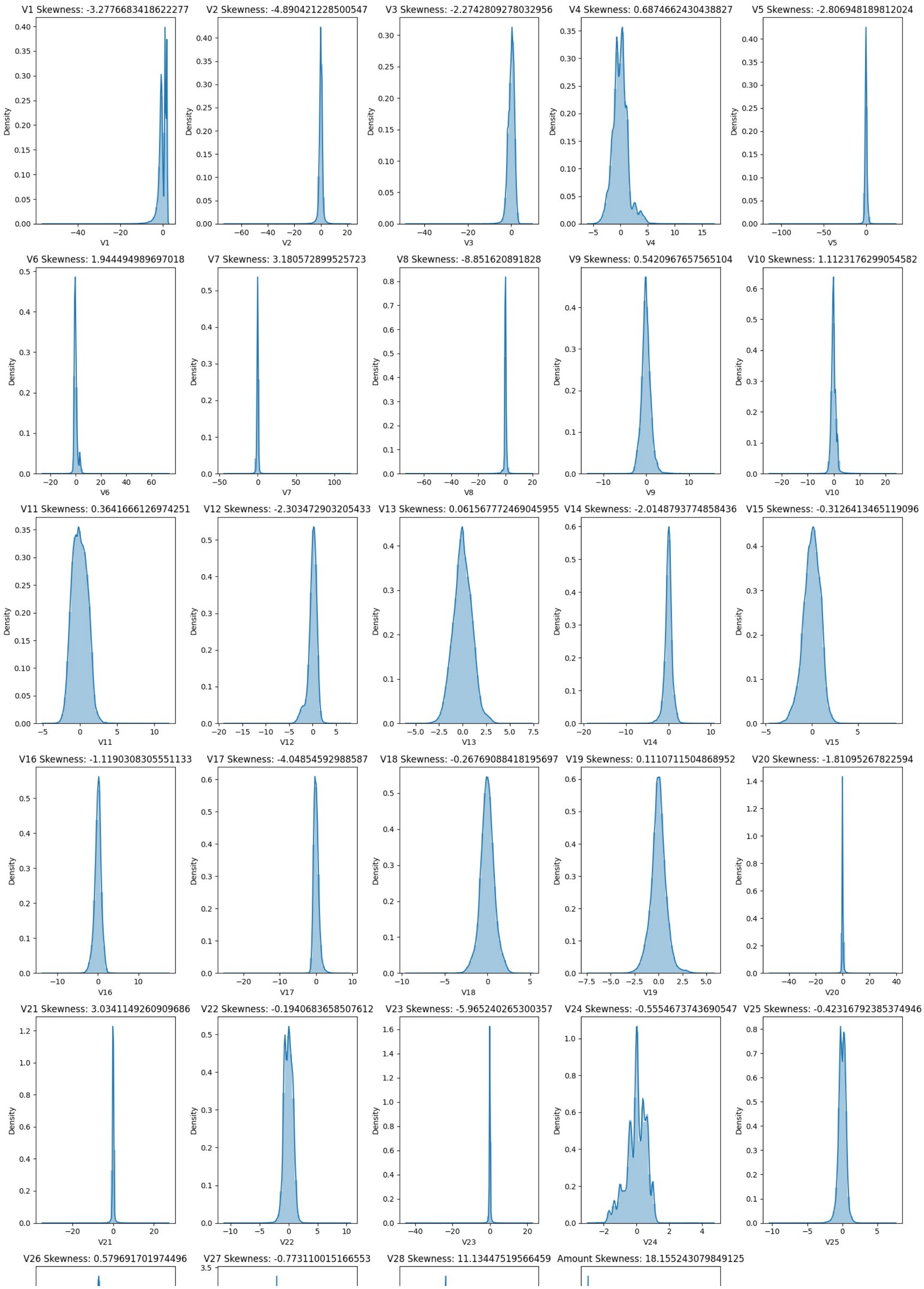
```
# Transform the test set
X_test['Amount'] = scaler.transform(X_test[['Amount']])
X_test.head()

              V1        V2        V3        V4        V5        V6
V7  \
49089   1.229452 -0.235478 -0.627166  0.419877  1.797014  4.069574 -
0.896223
154704  2.016893 -0.088751 -2.989257 -0.142575  2.675427  3.332289 -
0.652336
67247   0.535093 -1.469185  0.868279  0.385462 -1.439135  0.368118 -
0.499370
251657  2.128486 -0.117215 -1.513910  0.166456  0.359070 -0.540072
0.116023
201903  0.558593  1.587908 -2.368767  5.124413  2.171788 -0.500419
1.059829

              V8        V9       V10       V11       V12       V13
V14  \
49089   1.036103  0.745991 -0.147304 -0.850459  0.397845 -0.259849 -
0.277065
154704  0.752811  1.962566 -1.025024  1.126976 -2.418093  1.250341 -
0.056209
67247   0.303698  1.042073 -0.437209  1.145725  0.907573 -1.095634 -
0.055080
251657 -0.216140  0.680314  0.079977 -1.705327 -0.127579 -0.207945
```

```
0.307878
201903 -0.254233 -1.959060  0.948915 -0.288169 -1.007647  0.470316 -
2.771902

             V15       V16       V17       V18       V19       V20
V21  \
49089  -0.766810 -0.200946 -0.338122  0.006032  0.477431 -0.057922 -
0.170060
154704 -0.736695  0.014783  1.890249  0.333755 -0.450398 -0.147619 -
0.184153
67247  -0.621880 -0.191066  0.311988 -0.478635  0.231159  0.437685
0.028010
251657  0.213491  0.163032 -0.587029 -0.561292  0.472667 -0.227278 -
0.357993
201903  0.221958  0.354333  2.603189  1.092576  0.668084  0.249457 -
0.035049

             V22       V23       V24       V25       V26       V27
V28  \
49089  -0.288750 -0.130270  1.025935  0.847990 -0.271476  0.060052
0.018104
154704 -0.089661  0.087188  0.570679  0.101899  0.620842 -0.048958 -
0.042831
67247  -0.384708 -0.128376  0.286638 -0.136700  0.913904 -0.083364
0.052485
251657 -0.905085  0.223474 -1.075605 -0.188519  0.267672 -0.071733 -
0.072238
201903  0.271455  0.381606  0.332001 -0.334757  0.448890  0.168585
0.004955

          Amount
49089  -0.340485
154704 -0.320859
67247   0.853442
251657 -0.344410
201903 -0.229480
```

Checking the Skewness

```
# Plotting the distribution of the variables (skewness) of all the
columns
cols = X_train.columns

k = 0
plt.figure(figsize=(17, 28))

for col in cols:
    k = k + 1
    plt.subplot(6, 5, k)
```

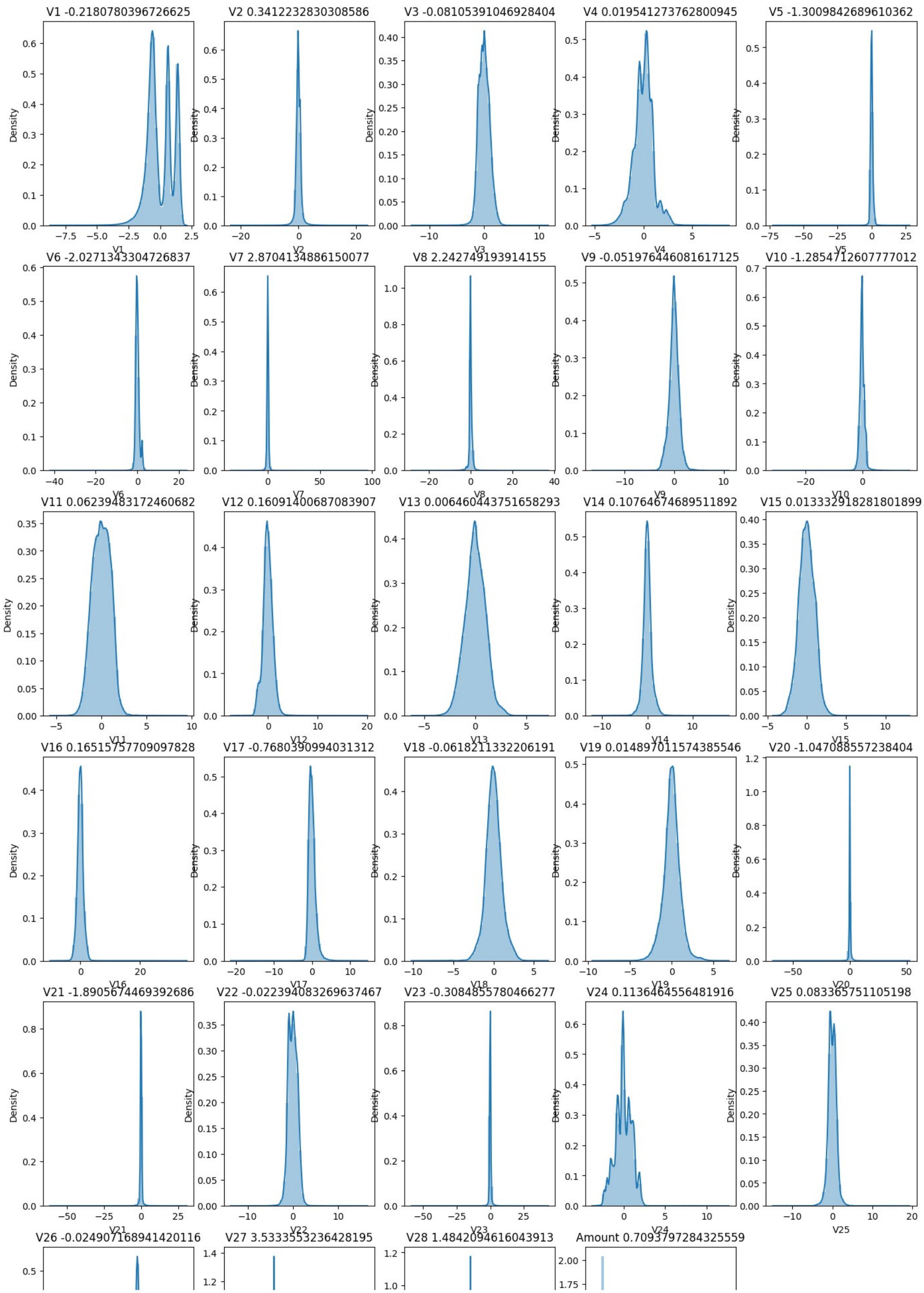
```
    sns.distplot(X_train[col])
    plt.title(col + ' Skewness: ' + str(X_train[col].skew()))

plt.tight_layout()
plt.show()
```


V1 Skewness: -3.2776683418622277
V2 Skewness: -4.890421228500547
V3 Skewness: -2.2742809278032956
V4 Skewness: 0.6874662430438827
V5 Skewness: -2.806948189812024
V6 Skewness: 1.944494989697018
V7 Skewness: 3.180572899525723
V8 Skewness: -8.851620891828
V9 Skewness: 0.5420967657565104
V10 Skewness: 1.1123176299054582
V11 Skewness: 0.3641666126974251
V12 Skewness: -2.303472903205433
V13 Skewness: 0.061567772469045955
V14 Skewness: -2.0148793774858436
V15 Skewness: -0.3126413465119096
V16 Skewness: -1.1190308305551133
V17 Skewness: -4.04854592988587
V18 Skewness: -0.26769088418195697
V19 Skewness: 0.1110711504868952
V20 Skewness: -1.81095267822594
V21 Skewness: 3.0341149260909686
V22 Skewness: -0.1940683658507612
V23 Skewness: -5.965240265300357
V24 Skewness: -0.5554673743690547
V25 Skewness: -0.42316792385374946
V26 Skewness: 0.579691701974496
V27 Skewness: -0.773110015166553
V28 Skewness: 11.13447519566459
Amount Skewness: 18.155243079849125

We see that there are many variables, which are heavily skewed. We will mitigate the skewness only for those variables for bringing them into normal distribution.

Mitigate skweness with PowerTransformer

```
# Importing PowerTransformer
from sklearn.preprocessing import PowerTransformer
# Instantiate the powertransformer
pt = PowerTransformer(method='yeo-johnson', standardize=True, 
copy=False)
# Fit and transform the PT on training data
X_train[cols] = pt.fit_transform(X_train)

# Transform the test set
X_test[cols] = pt.transform(X_test)

# Plotting the distribution of the variables (skewness) of all the 
columns
k=0
plt.figure(figsize=(17,28))
for col in cols :
    k=k+1
    plt.subplot(6, 5,k)
    sns.distplot(X_train[col])
    plt.title(col+' '+str(X_train[col].skew()))
```



V1 -0.2180780396726625
V2 0.3412232830308586
V3 -0.08105391046928404
V4 0.019541273762800945
V5 -1.3009842689610362
V6 -2.0271343304726837
V7 2.8704134886150077
V8 2.242749193914155
V9 -0.051976446081617125
V10 -1.2854712607777012
V11 0.06239483172460682
V12 0.16091400687083907
V13 0.006460443751658293
V14 0.10764674689511892
V15 0.013332918281801899
V16 0.16515757709097828
V17 -0.7680390994031312
V18 -0.0618211332206191
V19 0.014897011574385546
V20 -1.047088557238404
V21 -1.8905674469392686
V22 -0.022394083269637467
V23 -0.3084855780466277
V24 0.1136464556481916
V25 0.083365751105198
V26 -0.024907168941420116
V27 3.5333553236428195
V28 1.4842094616043913
Amount 0.7093797284325559
Density

Now we can see that all the variables are normally distributed after the transformation.

Model building on imbalanced data

Metric Selection for Heavily Imbalanced Data

Given the substantial class imbalance in the dataset, where only 0.17% of transactions are fraudulent, relying on accuracy as an evaluation metric is not prudent. Accuracy can be misleading in highly imbalanced scenarios, as a model could achieve high accuracy by simply predicting the majority class. In our case, even if the model predicts all instances as the majority class, it would still yield over 99% accuracy. To address this issue, the ROC-AUC score is a more suitable metric for fair evaluation. The ROC curve provides insights into the model's performance across various classification thresholds, offering a nuanced view of its discriminative power. By selecting an optimal threshold that balances true positive rate (TPR) and false positive rate (FPR), we can calculate the F1 score to assess precision and recall at the chosen threshold.

Reasons for Not Choosing SVM and Random Forest in Specific Cases

SVM

The decision to avoid SVM was based on the dataset's size, with 284,807 data points. When employing oversampling techniques, the number of data points increases further. SVM tends to be computationally demanding and resource-intensive, especially during cross-validation for hyperparameter tuning. Due to constraints in computational resources and time limitations, SVM was not explored in this context.

Random Forest

Similar resource constraints led to the decision to exclude Random Forest in specific hyperparameter tuning scenarios. The extensive computational requirements associated with oversampling techniques made the implementation of Random Forest impractical within the available constraints.

Exclusion of KNN in Model Building

K-Nearest Neighbors (KNN) was not considered for model building due to its inherent limitations in memory efficiency. As the dataset size grows, KNN becomes progressively slower, primarily because it needs to store all data points in memory. The computational burden arises when calculating distances for a single data point against the entire dataset to identify the nearest neighbors. This inefficiency renders KNN impractical for large datasets, prompting the exploration of alternative algorithms that offer better scalability and efficiency.

Logistic regression

```python
# Importing scikit logistic regression module
from sklearn.linear_model import LogisticRegression
```

```
# Impoting metrics
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import classification_report
```

Tuning hyperparameter C

C is the the inverse of regularization strength in Logistic Regression. Higher values of C correspond to less regularization.

```
# Importing libraries for cross validation
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as recall as we are more focused on acheiving the
higher sensitivity than the accuracy
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

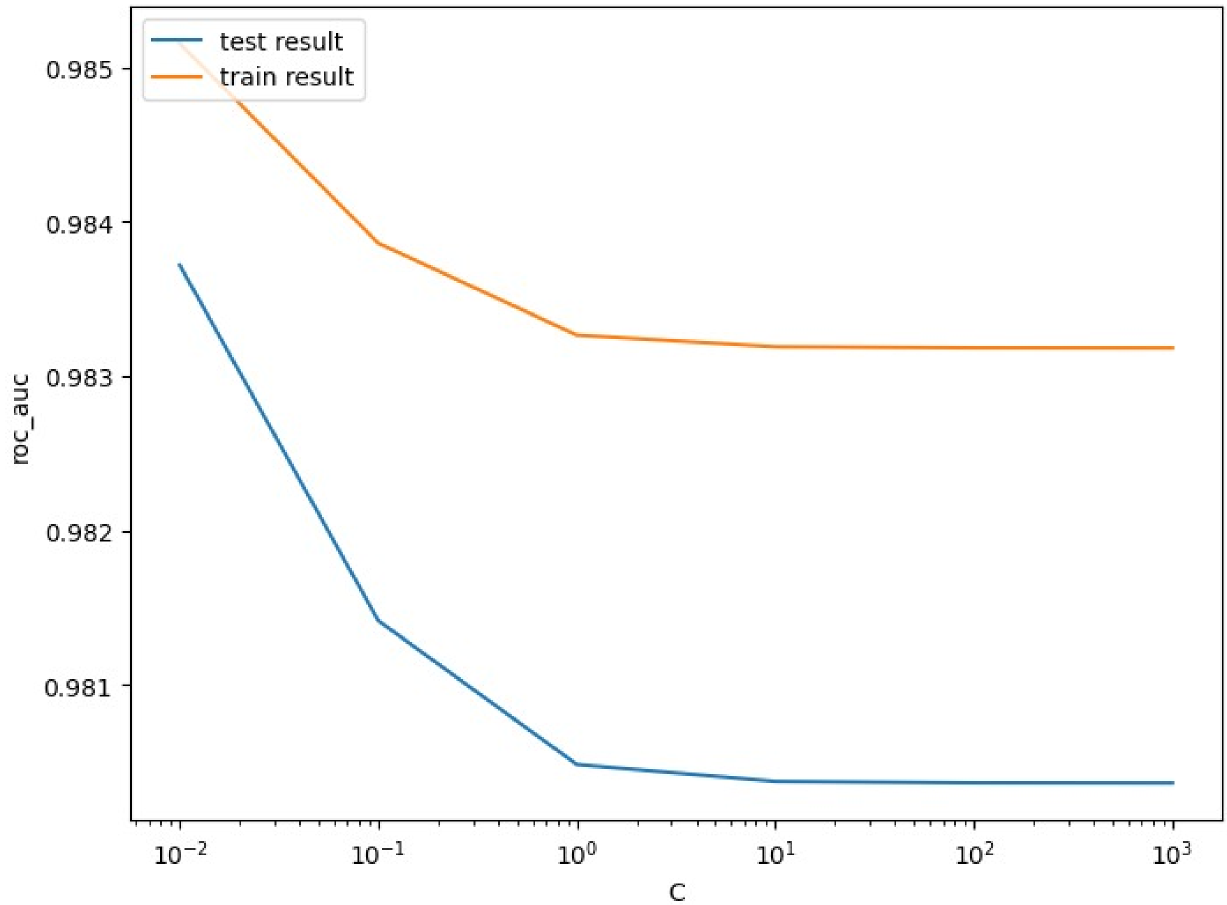
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
param_C  \
0       1.271986      0.379019         0.038552        0.019045
0.01
1       1.184194      0.135116         0.031305        0.012860
```

```
0.1
2       1.139086      0.123698         0.029236        0.012638
1
3       1.118935      0.054724         0.026788        0.013746
10
4       1.062211      0.118397         0.024069        0.005773
100
5       1.103135      0.100601         0.024923        0.010002
1000

        params  split0_test_score  split1_test_score
split2_test_score  \
0  {'C': 0.01}           0.986856           0.987234
0.968390
1   {'C': 0.1}           0.986104           0.987144
0.960929
2     {'C': 1}           0.985834           0.986806
0.958452
3    {'C': 10}           0.985798           0.986754
0.958181
4   {'C': 100}           0.985793           0.986748
0.958155
5  {'C': 1000}           0.985793           0.986747
0.958153

   split3_test_score  split4_test_score  mean_test_score
std_test_score  \
0           0.982373           0.993743         0.983719
0.008479
1           0.980620           0.992284         0.981416
0.010893
2           0.979781           0.991548         0.980484
0.011635
3           0.979674           0.991467         0.980375
0.011715
4           0.979666           0.991461         0.980365
0.011722
5           0.979663           0.991461         0.980363
0.011723

   rank_test_score  split0_train_score  split1_train_score  \
0                1            0.984043            0.984587
1                2            0.982402            0.983785
2                3            0.981722            0.983322
3                4            0.981632            0.983262
4                5            0.981625            0.983256
5                6            0.981623            0.983256

   split2_train_score  split3_train_score  split4_train_score  \
0            0.988474            0.985596            0.983075
```

1	0.987917	0.984018	0.981187
2	0.987492	0.983305	0.980489
3	0.987435	0.983216	0.980404
4	0.987429	0.983207	0.980396
5	0.987428	0.983206	0.980395

	mean_train_score	std_train_score
0	0.985155	0.001849
1	0.983862	0.002270
2	0.983266	0.002365
3	0.983190	0.002375
4	0.983182	0.002376
5	0.983182	0.002376

```
# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('roc_auc')
plt.legend(['test result', 'train result'], loc='upper left')
plt.xscale('log')
```




```
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test roc_auc is {0} at C = {1}".format(best_score,
best_C))

 The highest test roc_auc is 0.9837192853831933 at C = 0.01
```

Logistic regression with optimal C

```
# Instantiate the model with best C
logistic_imb = LogisticRegression(C=0.01)

# Fit the model on the train set
logistic_imb_model = logistic_imb.fit(X_train, y_train)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = logistic_imb_model.predict(X_train)
```

```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)
```

```
[[227427     22]
 [   135    261]]
```

```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_train, y_train_pred))
```

```
Accuracy:- 0.9993109350655051
Sensitivity:- 0.6590909090909091
Specificity:- 0.9999032750198946
F1-Score:- 0.7687776141384388
```

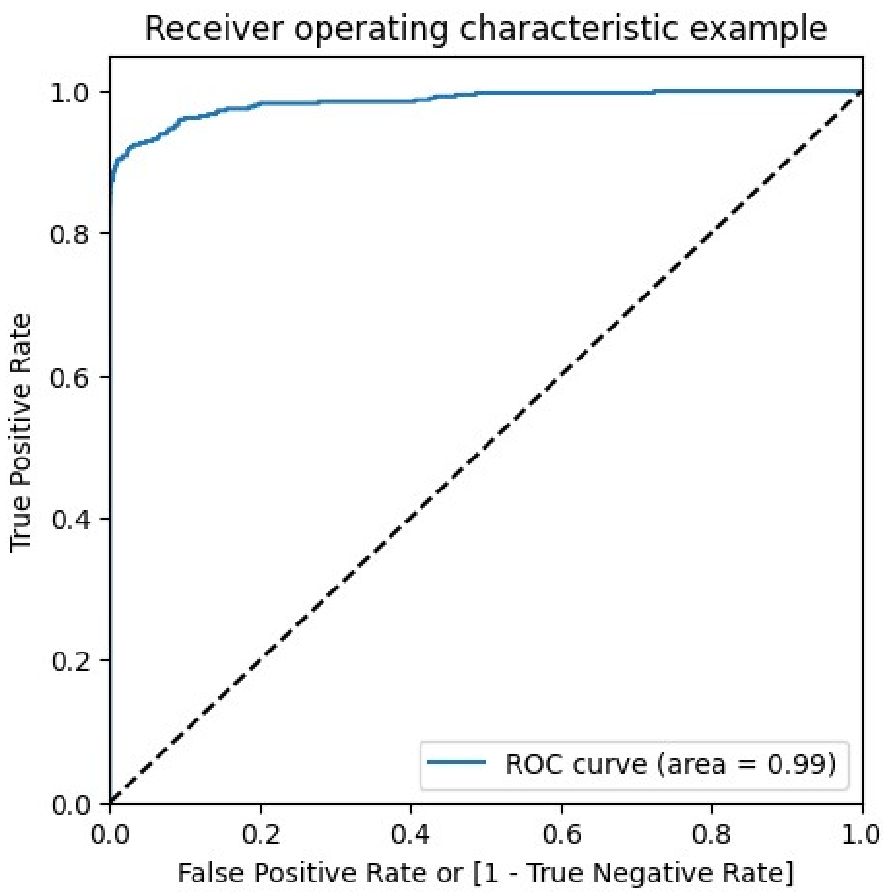
```python
# classification_report
print(classification_report(y_train, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       0.92      0.66      0.77       396

    accuracy                           1.00    227845
   macro avg       0.96      0.83      0.88    227845
weighted avg       1.00      1.00      1.00    227845
```

ROC on the train set

```python
# ROC Curve function

def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve( actual, probs,
                                              drop_intermediate =
False )
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(5, 5))
```

```
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return None

# Predicted probability
y_train_pred_proba = logistic_imb_model.predict_proba(X_train)[:,1]

# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba)
```



We acheived very good ROC 0.99 on the train set.

Prediction on the test set

```
# Prediction on the test set
y_test_pred = logistic_imb_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56850    16]
 [   42    54]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_test, y_test_pred))
```

```
Accuracy:- 0.9989817773252344
Sensitivity:- 0.5625
Specificity:- 0.9997186367952731
F1-Score:- 0.6506024096385543
```

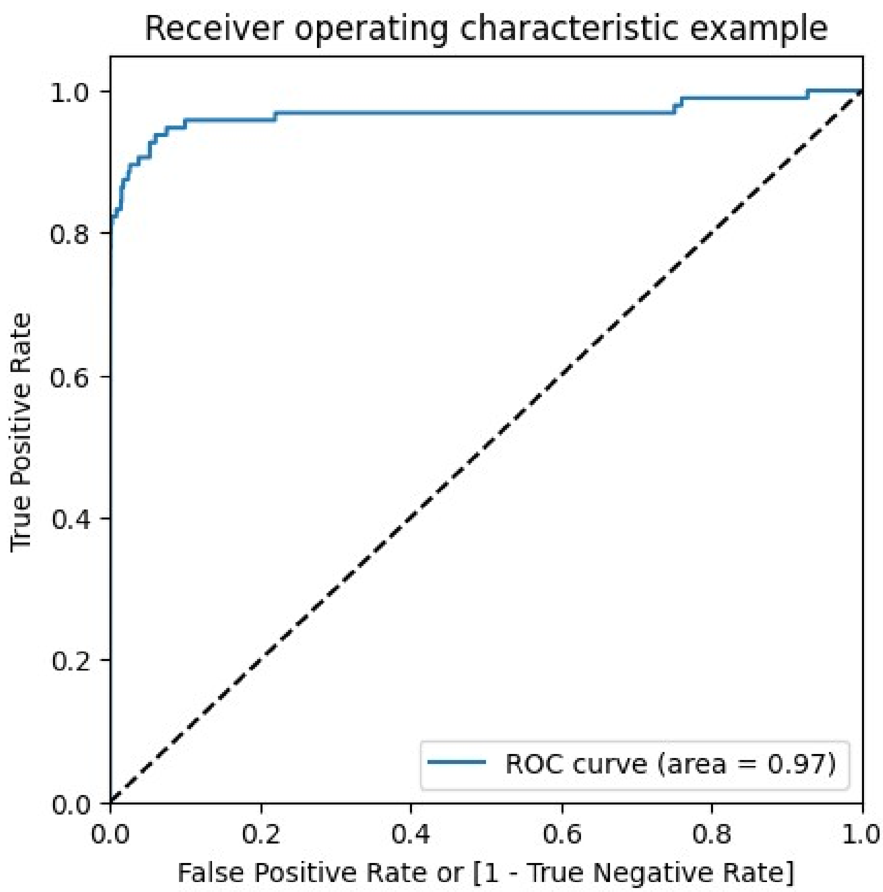
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.77      0.56      0.65        96

    accuracy                           1.00     56962
   macro avg       0.89      0.78      0.83     56962
weighted avg       1.00      1.00      1.00     56962
```

ROC on the test set

```
# Predicted probability
y_test_pred_proba = logistic_imb_model.predict_proba(X_test)[:,1]
```

```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



We can see that we have very good ROC on the test set 0.97, which is almost close to 1.

Model summary

- Train set
  - Accuracy = 0.99
  - Sensitivity = 0.70
  - Specificity = 0.99
  - F1-Score = 0.76
  - ROC = 0.99
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.77
  - Specificity = 0.99
  - F1-Score = 0.65
  - ROC = 0.97

Overall, the model is performing well in the test set, what it had learnt from the train set.

XGBoost

```
# Importing XGBoost
from xgboost import XGBClassifier
```

Tuning the hyperparameters

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
             'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train, y_train)
```

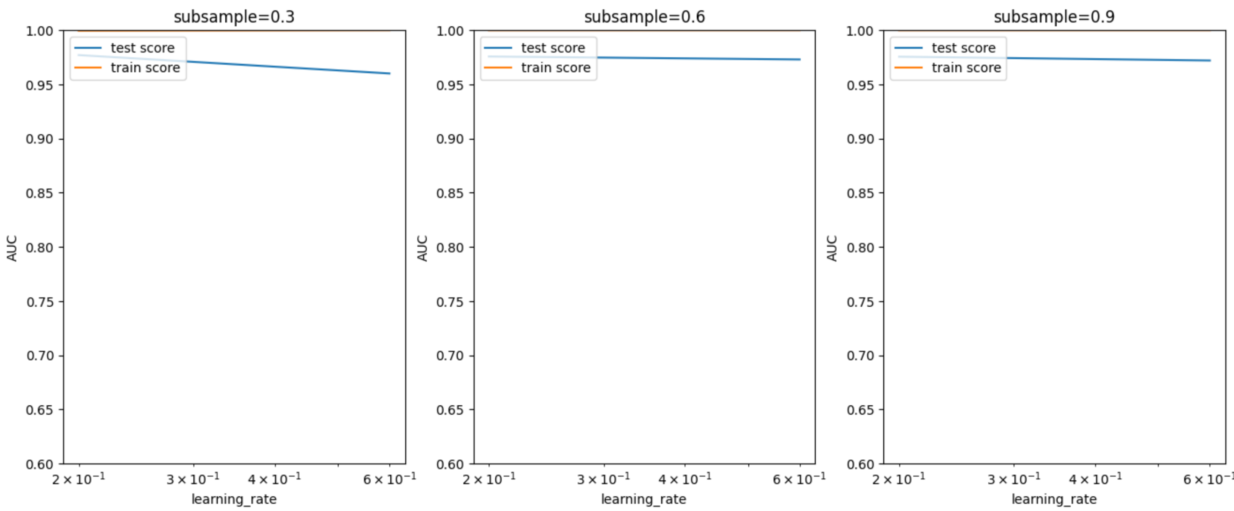
```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None,
colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False,
eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None,
importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_cat_threshold=None,
                                     max_cat_to_onehot=None,
                                     max_delta_step=None, max_depth=2,
```

```
                                             max_leaves=None,
min_child_weight=None,
                                             missing=nan,
monotone_constraints=None,
                                             multi_strategy=None,
n_estimators=200,
                                             n_jobs=None,
num_parallel_tree=None,
                                             random_state=None, ...),
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             return_train_score=True, scoring='roc_auc', verbose=1)

# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results

   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       2.535768      0.737668         0.067588        0.005965
1       1.793594      0.134591         0.061041        0.004689
2       1.675877      0.025984         0.056897        0.003275
3       1.776437      0.021462         0.059674        0.001658
4       1.748587      0.081171         0.057439        0.000894
5       1.858781      0.060790         0.063512        0.007455

  param_learning_rate param_subsample  \
0                 0.2             0.3
1                 0.2             0.6
2                 0.2             0.9
3                 0.6             0.3
4                 0.6             0.6
5                 0.6             0.9

                                     params  split0_test_score  \
0  {'learning_rate': 0.2, 'subsample': 0.3}           0.975585
1  {'learning_rate': 0.2, 'subsample': 0.6}           0.972484
2  {'learning_rate': 0.2, 'subsample': 0.9}           0.974963
3  {'learning_rate': 0.6, 'subsample': 0.3}           0.955029
4  {'learning_rate': 0.6, 'subsample': 0.6}           0.974179
5  {'learning_rate': 0.6, 'subsample': 0.9}           0.968630

   split1_test_score  split2_test_score  mean_test_score
std_test_score  \
0           0.974595           0.980946         0.977042
0.002790
1           0.976596           0.978129         0.975736
0.002383
2           0.972600           0.979148         0.975570
0.002707
3           0.953863           0.971355         0.960083
```

```
0.007985
4            0.970199           0.974645         0.973008
0.001995
5            0.972430           0.975082         0.972047
0.002648

   rank_test_score  split0_train_score  split1_train_score  \
0                1            0.999865            0.999600
1                2            0.999963            0.999952
2                3            0.999963            0.999971
3                6            0.999998            0.999997
4                4            1.000000            1.000000
5                5            1.000000            1.000000

   split2_train_score  mean_train_score  std_train_score
0            0.999272          0.999579         0.000243
1            0.999955          0.999957         0.000005
2            0.999945          0.999960         0.000011
3            0.999995          0.999997         0.000001
4            1.000000          1.000000         0.000000
5            1.000000          1.000000         0.000000
```

```python
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
              'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



Model with optimal hyperparameters

We see that the train score almost touches to 1. Among the hyperparameters, we can choose the best parameters as learning_rate : 0.2 and subsample: 0.3

```
model_cv.best_params_

{'learning_rate': 0.2, 'subsample': 0.3}

# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label,
which we need for calculating auc
params = {'learning_rate': 0.2,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.9,
         'objective':'binary:logistic'}

# fit model on training data
xgb_imb_model = XGBClassifier(params = params)
xgb_imb_model.fit(X_train, y_train)

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
```

```
                  multi_strategy=None, n_estimators=None, n_jobs=None,
                  num_parallel_tree=None,
                  params={'learning_rate': 0.2, 'max_depth': 2,
'n_estimators': 200,
                          'objective': 'binary:logistic', 'subsample':
0.9}, ...)
```

Prediction on the train set

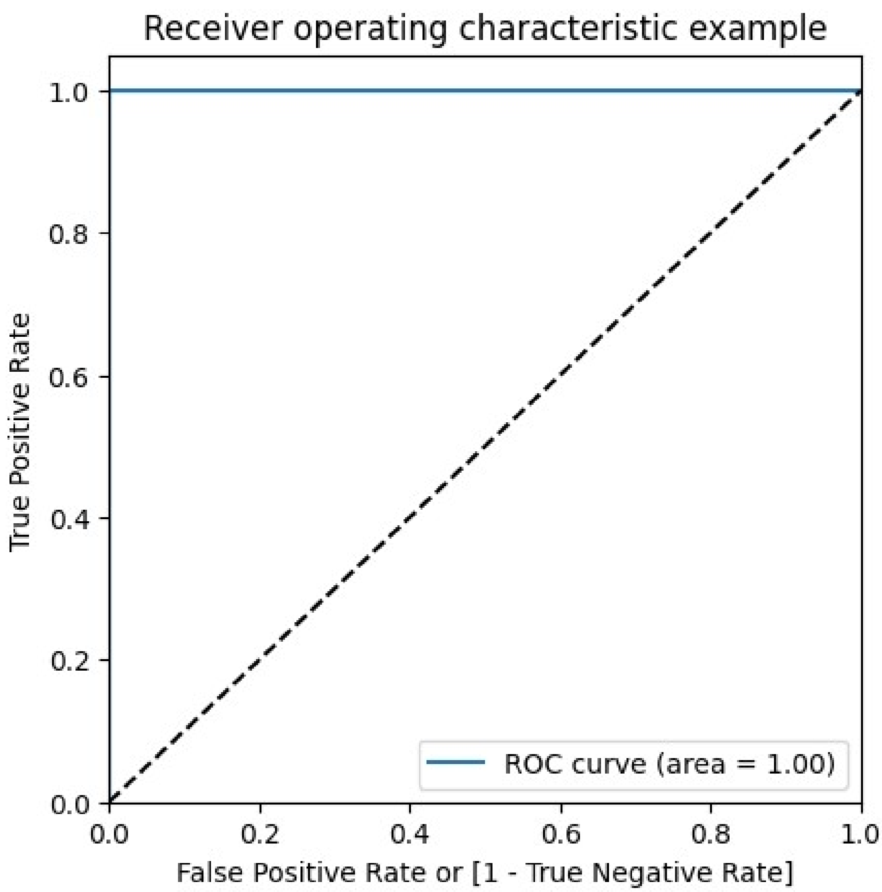
```
# Predictions on the train set
y_train_pred = xgb_imb_model.predict(X_train)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)

[[227449      0]
 [     0    396]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_train, y_train_pred))

Accuracy:- 1.0
Sensitivity:- 1.0
Specificity:- 1.0
F1-Score:- 1.0

# classification_report
print(classification_report(y_train, y_train_pred))

              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00       396

    accuracy                           1.00    227845
   macro avg       1.00      1.00      1.00    227845
```

```
weighted avg       1.00      1.00      1.00    227845
```

```
# Predicted probability
y_train_pred_proba_imb_xgb = xgb_imb_model.predict_proba(X_train)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train, y_train_pred_proba_imb_xgb)
auc
```

```
1.0
```

```
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba_imb_xgb)
```



Prediction on the test set

```
# Predictions on the test set
y_test_pred = xgb_imb_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56858     8]
 [   25    71]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_test, y_test_pred))
```

```
Accuracy:- 0.999420666409185
Sensitivity:- 0.7395833333333334
Specificity:- 0.9998593183976365
F1-Score:- 0.8114285714285714
```

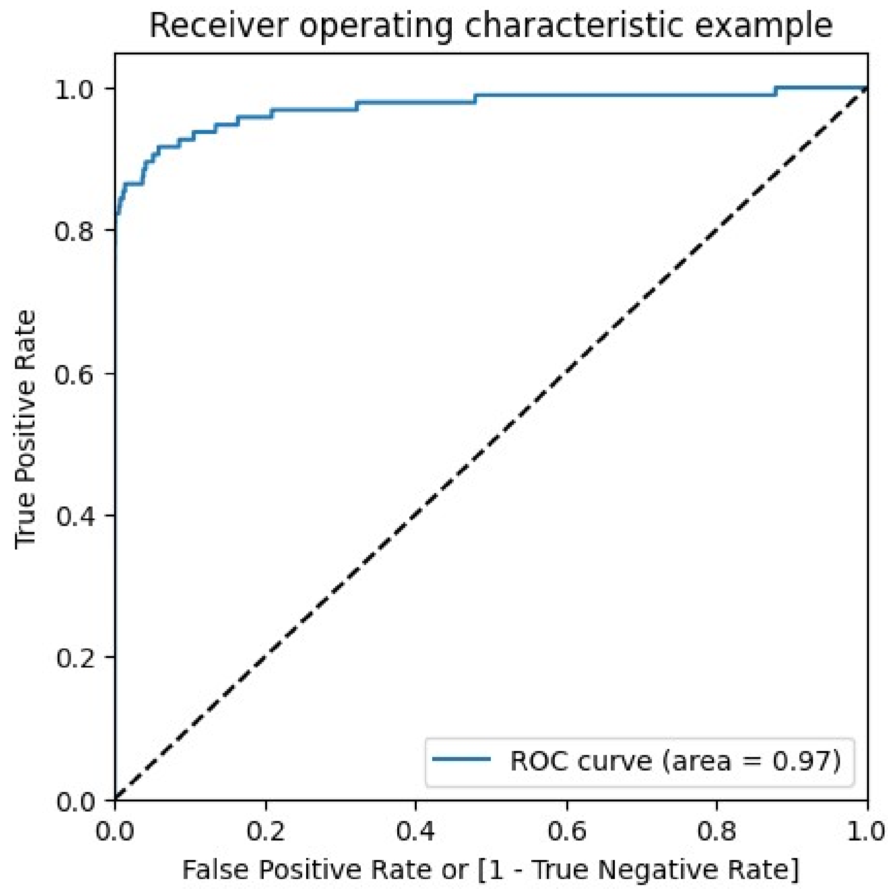
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.90      0.74      0.81        96

    accuracy                           1.00     56962
   macro avg       0.95      0.87      0.91     56962
weighted avg       1.00      1.00      1.00     56962
```

```
# Predicted probability
y_test_pred_proba = xgb_imb_model.predict_proba(X_test)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9723599118981465
```

```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.99
  - Sensitivity = 0.85
  - Specificity = 0.99
  - ROC-AUC = 0.99
  - F1-Score = 0.90
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.75
  - Specificity = 0.99
  - ROC-AUC = 0.98
  - F-Score = 0.79

Overall, the model is performing well in the test set, what it had learnt from the train set.

Choosing best model on the imbalanced data

We can see that among all the models we tried (Logistic, XGBoost, Decision Tree, and Random Forest), almost all of them have performed well. More specifically Logistic regression and XGBoost performed best in terms of ROC-AUC score.

But as we have to choose one of them, we can go for the best as `XGBoost`, which gives us ROC score of 1.0 on the train data and 0.98 on the test data.

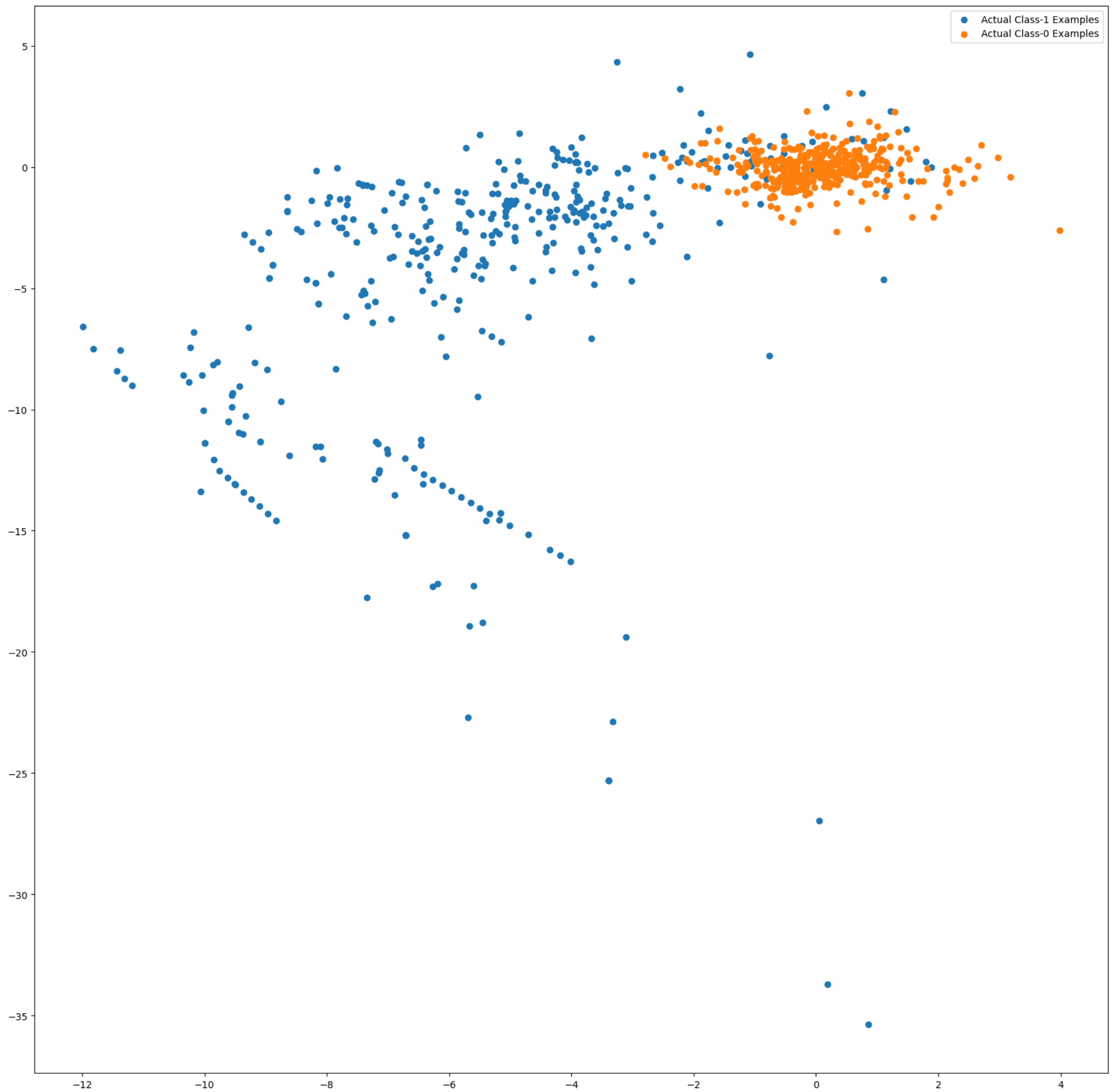
Keep in mind that XGBoost requires more resource utilization than Logistic model. Hence building XGBoost model is more costlier than the Logistic model. But XGBoost having ROC score 0.98, which is 0.01 more than the Logistic model. The 0.01 increase of score may convert into huge amount of saving for the bank.

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

```
# Features of XGBoost model

var_imp = []
for i in xgb_imb_model.feature_importances_:
    var_imp.append(i)
print('Top var =',
var_imp.index(np.sort(xgb_imb_model.feature_importances_)[-1])+1)
print('2nd Top var =',
var_imp.index(np.sort(xgb_imb_model.feature_importances_)[-2])+1)
print('3rd Top var =',
var_imp.index(np.sort(xgb_imb_model.feature_importances_)[-3])+1)
# Variable on Index-16 and Index-13 seems to be the top 2 variables
top_var_index =
var_imp.index(np.sort(xgb_imb_model.feature_importances_)[-1])
second_top_var_index =
var_imp.index(np.sort(xgb_imb_model.feature_importances_)[-2])

X_train_1 = X_train.to_numpy()[np.where(y_train==1.0)]
X_train_0 = X_train.to_numpy()[np.where(y_train==0.0)]

np.random.shuffle(X_train_0)

import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = [20, 20]

plt.scatter(X_train_1[:, top_var_index], X_train_1[:,
second_top_var_index], label='Actual Class-1 Examples')
plt.scatter(X_train_0[:X_train_1.shape[0], top_var_index],
X_train_0[:X_train_1.shape[0], second_top_var_index],
            label='Actual Class-0 Examples')
plt.legend()
```

```
Top var = 14
2nd Top var = 7
3rd Top var = 10

<matplotlib.legend.Legend at 0x161d3791990>
```



Print the FPR,TPR & select the best threshold from the roc curve for the best model

```
print('Train auc =', metrics.roc_auc_score(y_train,
y_train_pred_proba_imb_xgb))
fpr, tpr, thresholds = metrics.roc_curve(y_train,
y_train_pred_proba_imb_xgb)
```

```
threshold = thresholds[np.argmax(tpr-fpr)]
print("Threshold=",threshold)

Train auc = 1.0
Threshold= 0.82052475
```

We can see that the threshold is 0.85, for which the TPR is the highest and FPR is the lowest and we got the best ROC score.

Handling data imbalance

As we see that the data is heavily imbalanced, We will try several approaches for handling data imbalance.

- Undersampling :- Here for balancing the class distribution, the non-fraudulent transctions count will be reduced to 396 (similar count of fraudulent transctions)
- Oversampling :- Here we will make the same count of non-fraudulent transctions as fraudulent transctions.
- SMOTE :- Synthetic minority oversampling technique. It is another oversampling technique, which uses nearest neighbor algorithm to create synthetic data.
- Adasyn:- This is similar to SMOTE with minor changes that the new synthetic data is generated on the region of low density of imbalanced data points.

Undersampling

```
# Importing undersampler library
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter

# instantiating the random undersampler
rus = RandomUnderSampler()
# resampling X, y
X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)

# Befor sampling class distribution
print('Before sampling class distribution:-',Counter(y_train))
# new class distribution
print('New class distribution:-',Counter(y_train_rus))

Before sampling class distribution:- Counter({0: 227449, 1: 396})
New class distribution:- Counter({0: 396, 1: 396})
```

Model building on balanced data with Undersampling

Logistic Regression

```
# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as roc-auc
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train_rus, y_train_rus)
```

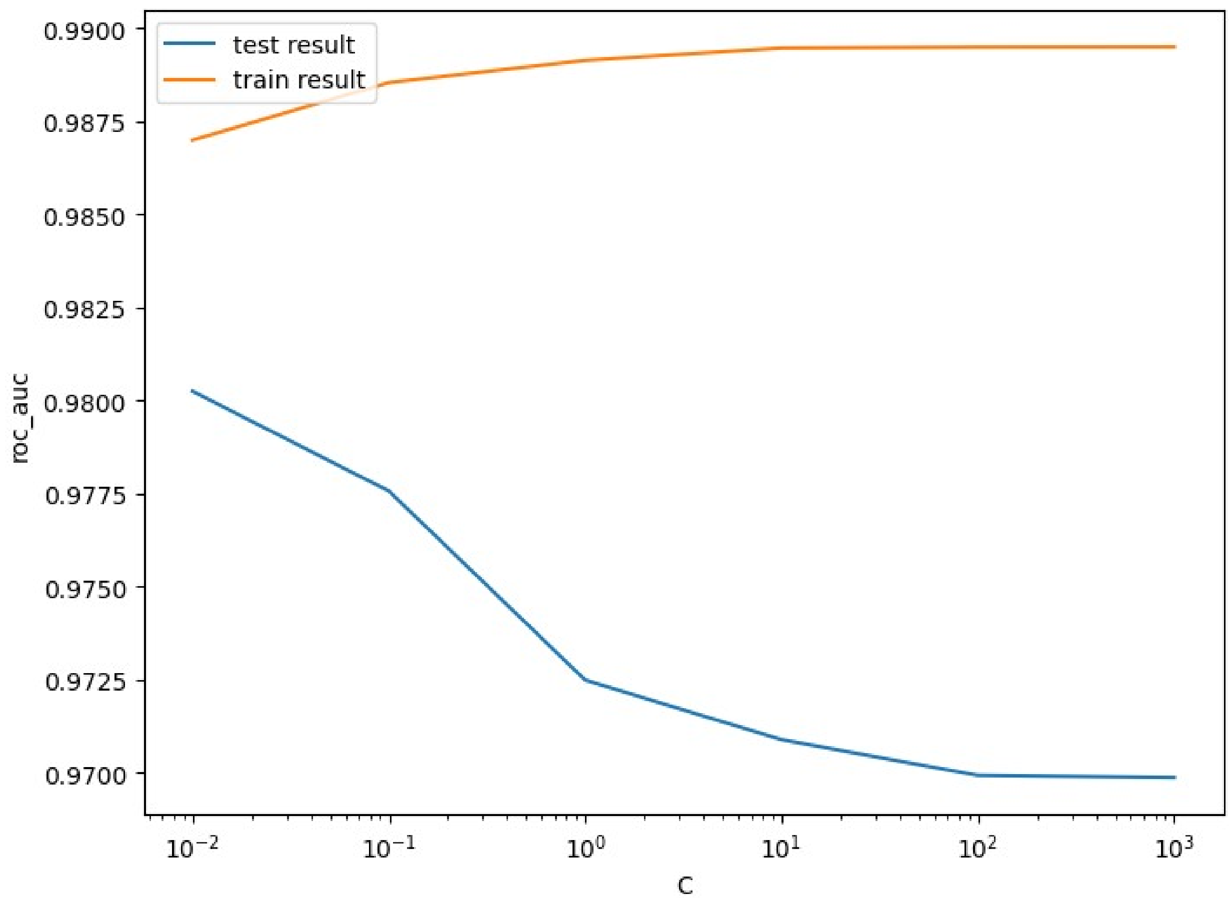
```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
param_C  \
0       0.012725      0.001290         0.005466        0.001567
0.01
1       0.014703      0.004255         0.005208        0.000510
0.1
2       0.017486      0.003684         0.003707        0.000864
1
3       0.018295      0.004838         0.003241        0.003386
10
4       0.020883      0.003173         0.001938        0.003326
100
5       0.017613      0.004038         0.007724        0.001553
1000

        params  split0_test_score  split1_test_score
split2_test_score  \
0  {'C': 0.01}           0.982671           0.991928
0.968750
```

```
1   {'C': 0.1}           0.975994          0.990345
0.971154
2     {'C': 1}           0.969157          0.985755
0.964263
3    {'C': 10}           0.967250          0.985122
0.958173
4   {'C': 100}           0.965660          0.984647
0.955449
5  {'C': 1000}           0.965501          0.984647
0.954968

   split3_test_score  split4_test_score  mean_test_score
std_test_score  \
0           0.977714           0.980177         0.980248
0.007496
1           0.970980           0.979371         0.977569
0.007121
2           0.964727           0.978566         0.972494
0.008390
3           0.964887           0.979049         0.970896
0.009799
4           0.965047           0.978888         0.969938
0.010475
5           0.964887           0.979371         0.969875
0.010720

   rank_test_score  split0_train_score  split1_train_score  \
0                1            0.986709            0.985794
1                2            0.988776            0.987052
2                3            0.989595            0.987172
3                4            0.990444            0.987322
4                5            0.990414            0.987352
5                6            0.990394            0.987361

   split2_train_score  split3_train_score  split4_train_score  \
0            0.988566            0.987023            0.986869
1            0.989850            0.988386            0.988622
2            0.990546            0.989182            0.989159
3            0.990894            0.989362            0.989269
4            0.990865            0.989362            0.989438
5            0.990904            0.989372            0.989428

   mean_train_score  std_train_score
0          0.986992         0.000895
1          0.988537         0.000896
2          0.989131         0.001101
3          0.989458         0.001236
4          0.989486         0.001211
5          0.989492         0.001214
```

```python
# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('roc_auc')
plt.legend(['test result', 'train result'], loc='upper left')
plt.xscale('log')
```




```python
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test roc_auc is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test roc_auc is 0.9802479304463592 at C = 0.01
```

Logistic regression with optimal C

```
# Instantiate the model with best C
logistic_bal_rus = LogisticRegression(C=0.1)

# Fit the model on the train set
logistic_bal_rus_model = logistic_bal_rus.fit(X_train_rus,
y_train_rus)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = logistic_bal_rus_model.predict(X_train_rus)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_pred)
print(confusion)
```

```
[[389   7]
 [ 31 365]]
```

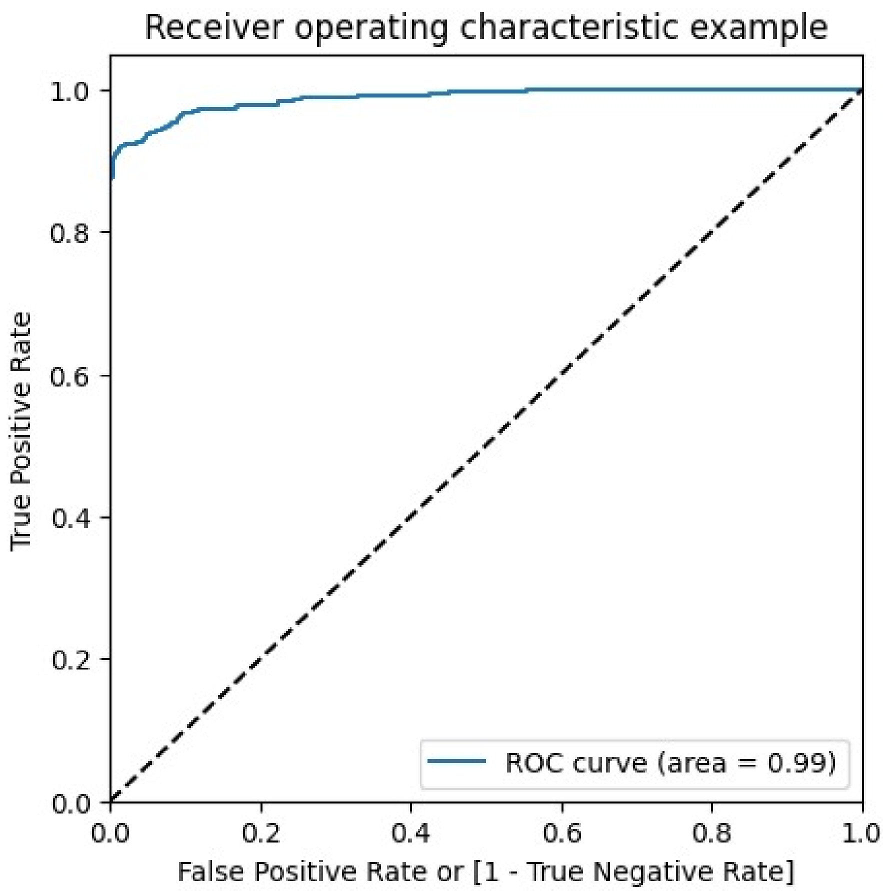
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_rus, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_train_rus, y_train_pred))
```

```
Accuracy:- 0.952020202020202
Sensitivity:- 0.9217171717171717
Specificity:- 0.9823232323232324
F1-Score:- 0.9505208333333334
```

```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95       396
           1       0.98      0.92      0.95       396

    accuracy                           0.95       792
   macro avg       0.95      0.95      0.95       792
```

```
weighted avg       0.95      0.95      0.95       792


# Predicted probability
y_train_pred_proba = logistic_bal_rus_model.predict_proba(X_train_rus)
[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc

0.9878264972961943

# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

```
# Prediction on the test set
y_test_pred = logistic_bal_rus_model.predict(X_test)
```

```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55742  1124]
 [   14    82]]
```

```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.980021768898564
Sensitivity:- 0.8541666666666666
Specificity:- 0.9802342348679352
```

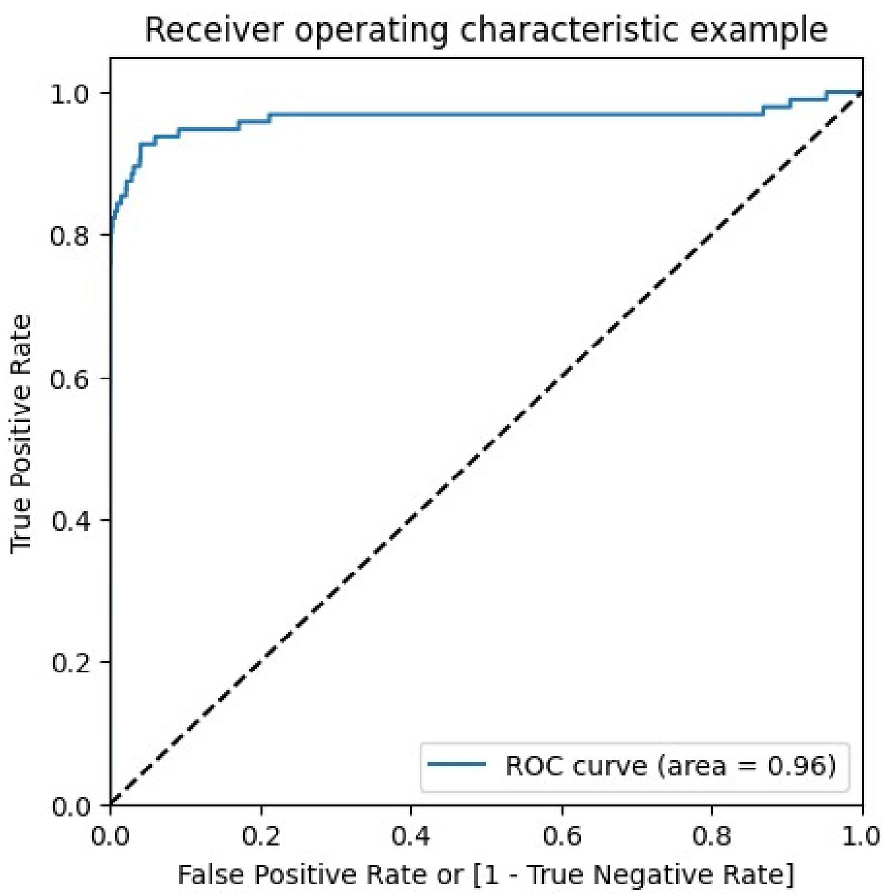
```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.07      0.85      0.13        96

    accuracy                           0.98     56962
   macro avg       0.53      0.92      0.56     56962
weighted avg       1.00      0.98      0.99     56962
```

```python
# Predicted probability
y_test_pred_proba = logistic_bal_rus_model.predict_proba(X_test)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9630049516993165
```

```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.95
  - Sensitivity = 0.92
  - Specificity = 0.98
  - ROC = 0.99
- Test set
  - Accuracy = 0.97
  - Sensitivity = 0.86
  - Specificity = 0.97
  - ROC = 0.96

XGBoost

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
```

```
              'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train_rus, y_train_rus)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

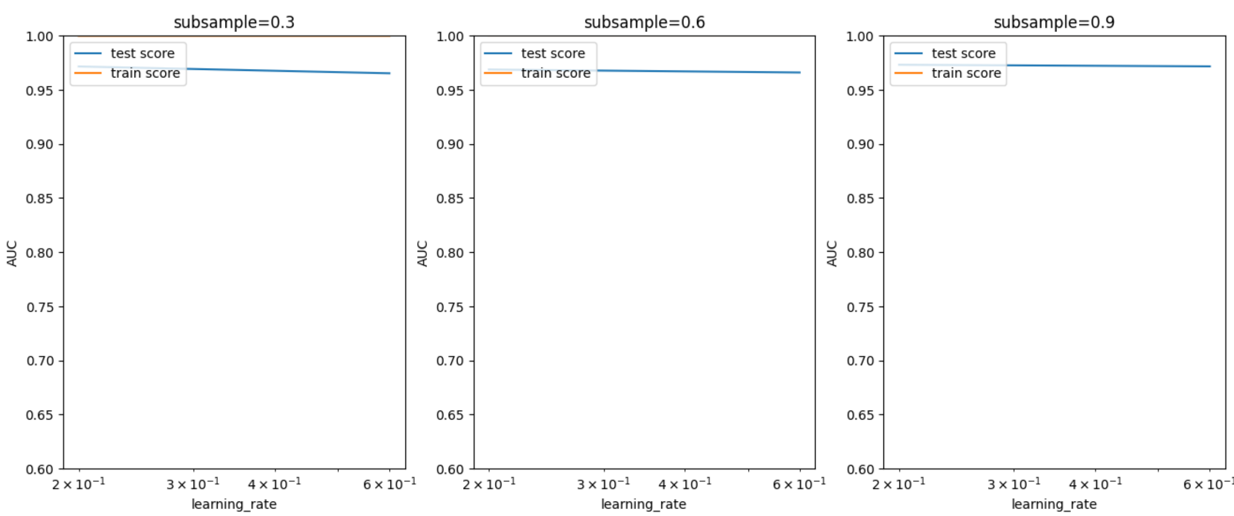
```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None,
colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False,
eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None,
importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_cat_threshold=None,
                                     max_cat_to_onehot=None,
                                     max_delta_step=None, max_depth=2,
                                     max_leaves=None,
min_child_weight=None,
                                     missing=nan,
monotone_constraints=None,
                                     multi_strategy=None,
n_estimators=200,
                                     n_jobs=None,
num_parallel_tree=None,
                                     random_state=None, ...),
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```python
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.150507      0.035023         0.012531        0.001830   
1       0.121757      0.009973         0.008923        0.000888   
2       0.118611      0.005923         0.010269        0.001908   
3       0.098031      0.007062         0.009510        0.001451   
4       0.101163      0.000077         0.010325        0.000427   
5       0.132967      0.049887         0.013083        0.003599   

  param_learning_rate param_subsample  \
0                 0.2             0.3   
1                 0.2             0.6   
2                 0.2             0.9   
3                 0.6             0.3   
4                 0.6             0.6   
5                 0.6             0.9   

                                      params  split0_test_score  \
0  {'learning_rate': 0.2, 'subsample': 0.3}           0.969984   
1  {'learning_rate': 0.2, 'subsample': 0.6}           0.966081   
2  {'learning_rate': 0.2, 'subsample': 0.9}           0.973887   
3  {'learning_rate': 0.6, 'subsample': 0.3}           0.963958   
4  {'learning_rate': 0.6, 'subsample': 0.6}           0.961490   
5  {'learning_rate': 0.6, 'subsample': 0.9}           0.971017   

   split1_test_score  split2_test_score  mean_test_score
std_test_score  \
0           0.964474           0.980200         0.971553
0.006515   
1           0.962580           0.977560         0.968740
0.006398   
2           0.965737           0.979798         0.973140
0.005765   
3           0.961892           0.969927         0.965259
0.003407   
4           0.961346           0.975149         0.965995
0.006473   
5           0.968262           0.975666         0.971648
0.003055   

   rank_test_score  split0_train_score  split1_train_score  \
0                3            0.999900            0.999842   
1                4            1.000000            1.000000   
2                1            1.000000            1.000000   
3                6            0.999699            0.999928   
4                5            1.000000            1.000000   
5                2            1.000000            1.000000   
```

	split2_train_score	mean_train_score	std_train_score
0	0.9999	0.999880	0.000027
1	1.0000	1.000000	0.000000
2	1.0000	1.000000	0.000000
3	1.0000	0.999876	0.000129
4	1.0000	1.000000	0.000000
5	1.0000	1.000000	0.000000

```python
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
              'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



Model with optimal hyperparameters

We see that the train score almost touches to 1. Among the hyperparameters, we can choose the best parameters as learning_rate : 0.2 and subsample: 0.3

```
model_cv.best_params_
```

```
{'learning_rate': 0.2, 'subsample': 0.9}
```

```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': 0.2,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.6,
         'objective':'binary:logistic'}

# fit model on training data
xgb_bal_rus_model = XGBClassifier(params = params)
xgb_bal_rus_model.fit(X_train_rus, y_train_rus)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None,
              params={'learning_rate': 0.2, 'max_depth': 2,
'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample':
0.6}, ...)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = xgb_bal_rus_model.predict(X_train_rus)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_rus)
print(confusion)
```

```
[[396   0]
 [  0 396]]
```

```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_rus, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 1.0
Sensitivity:- 1.0
Specificity:- 1.0
```

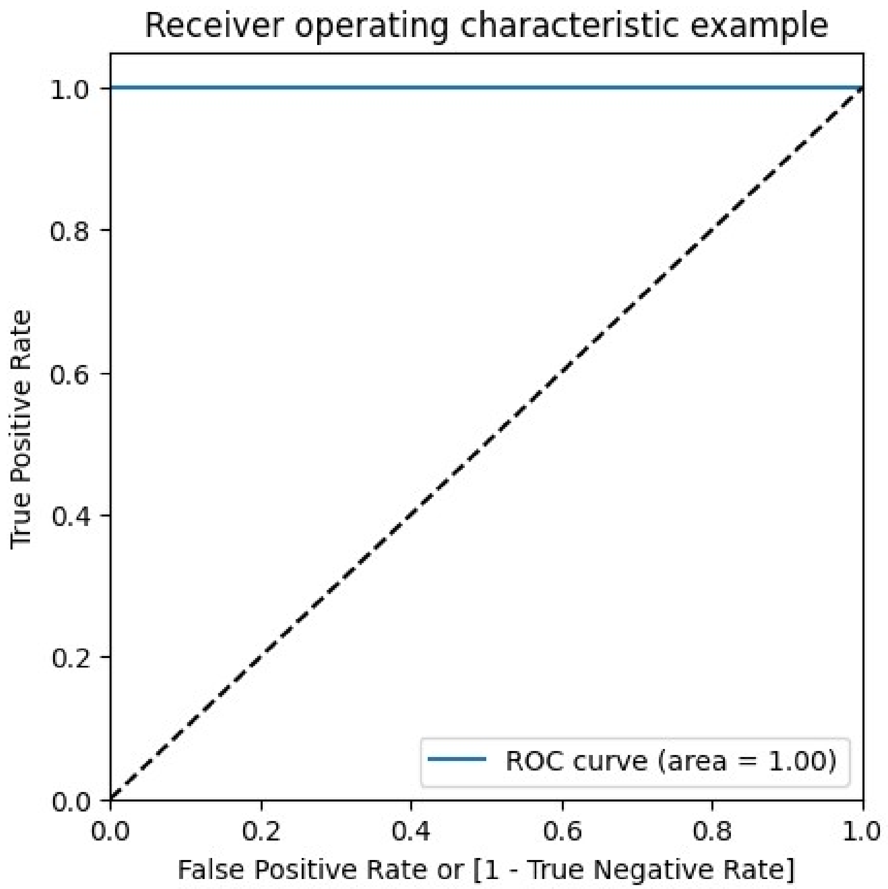
```python
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       396
           1       1.00      1.00      1.00       396

    accuracy                           1.00       792
   macro avg       1.00      1.00      1.00       792
weighted avg       1.00      1.00      1.00       792
```

```python
# Predicted probability
y_train_pred_proba = xgb_bal_rus_model.predict_proba(X_train_rus)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

```
1.0
```

```python
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

```
# Predictions on the test set
y_test_pred = xgb_bal_rus_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)

[[55312  1554]
 [   14    82]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))
```

```python
# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9724728766546118
Sensitivity:- 0.8541666666666666
Specificity:- 0.9726725987408996
```

```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

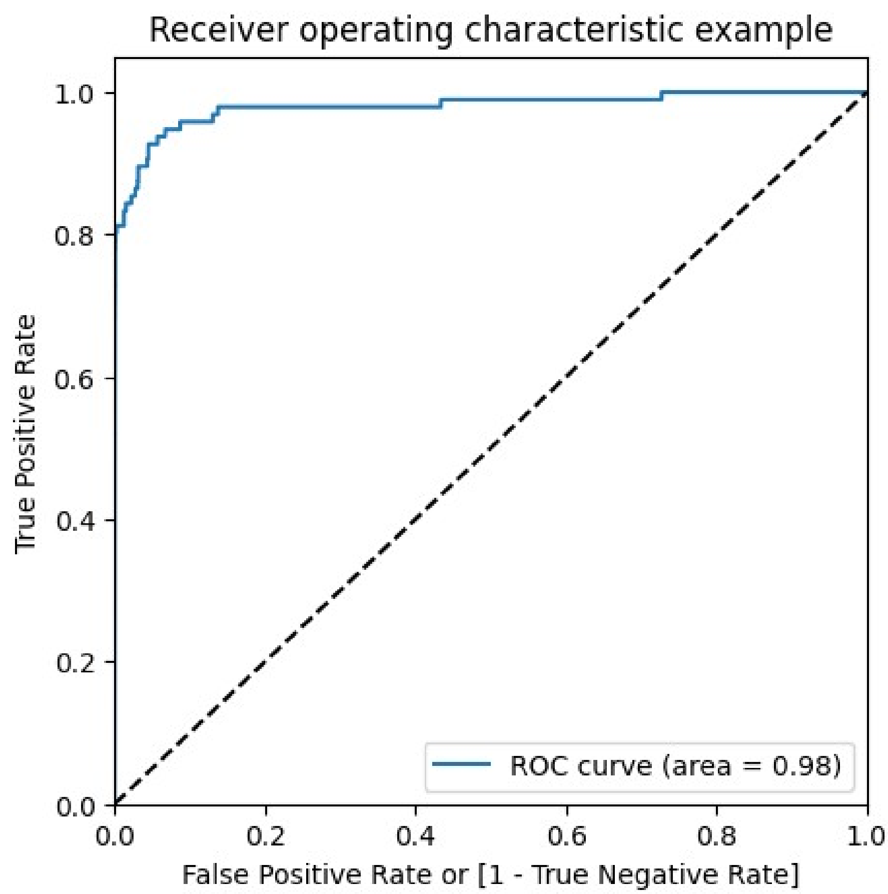
```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     56866
           1       0.05      0.85      0.09        96

    accuracy                           0.97     56962
   macro avg       0.52      0.91      0.54     56962
weighted avg       1.00      0.97      0.98     56962
```

```python
# Predicted probability
y_test_pred_proba = xgb_bal_rus_model.predict_proba(X_test)[:,1]
```

```python
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9792754750934947
```

```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 1.0
  - Sensitivity = 1.0
  - Specificity = 1.0
  - ROC-AUC = 1.0
- Test set
  - Accuracy = 0.96
  - Sensitivity = 0.92
  - Specificity = 0.96
  - ROC-AUC = 0.98

Oversampling

```
# Importing oversampler library
from imblearn.over_sampling import RandomOverSampler

# instantiating the random oversampler
ros = RandomOverSampler()
```

```
# resampling X, y
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)

# Befor sampling class distribution
print('Before sampling class distribution:-',Counter(y_train))
# new class distribution
print('New class distribution:-',Counter(y_train_ros))
```

```
Before sampling class distribution:- Counter({0: 227449, 1: 396})
New class distribution:- Counter({0: 227449, 1: 227449})
```

Logistic Regression

```
# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as roc-auc
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train_ros, y_train_ros)
```

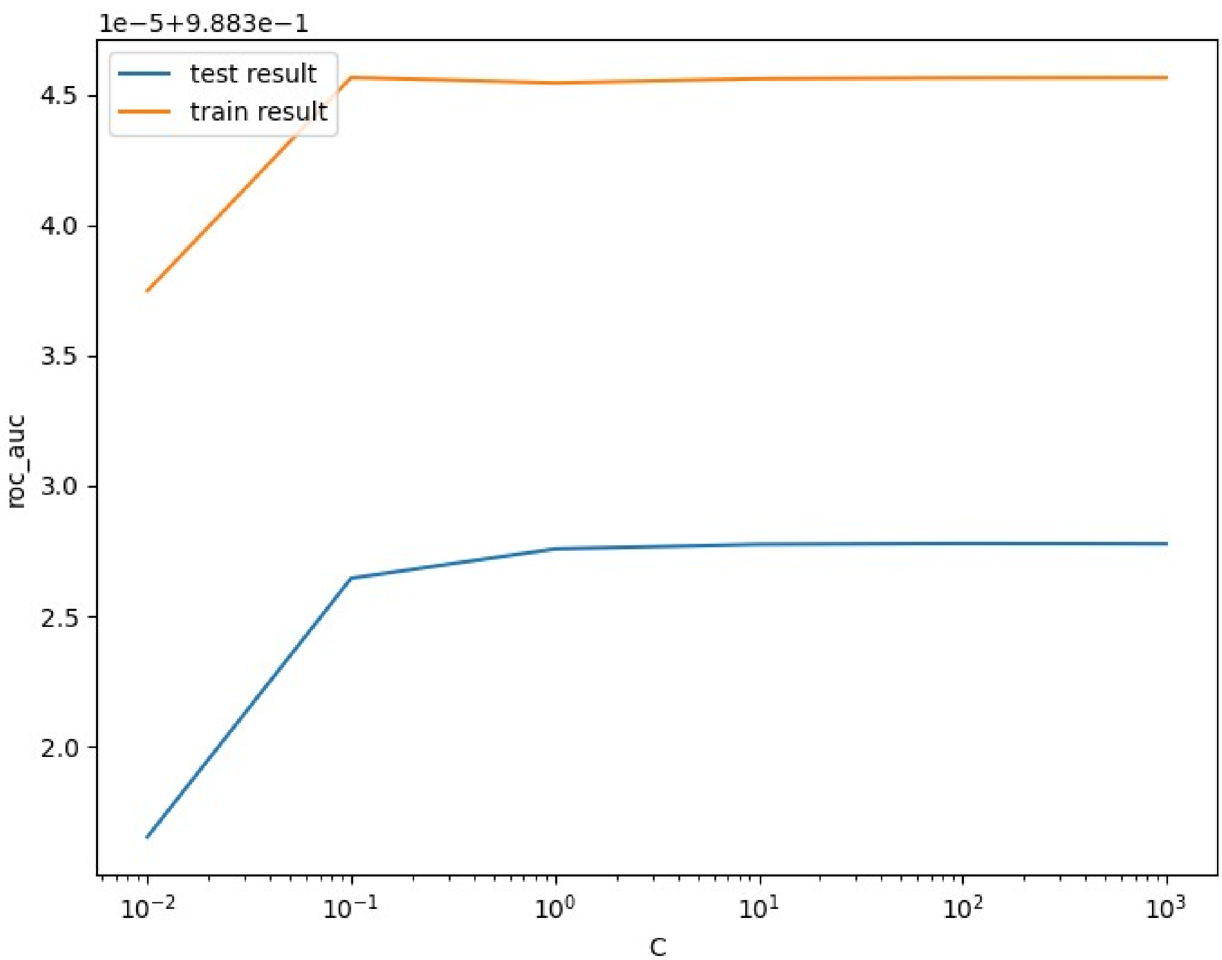
```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
param_C  \
0       3.019965      0.728350         0.055811        0.010448
0.01
1       2.162980      0.074955         0.036827        0.002772
0.1
2       2.068724      0.116851         0.038328        0.002808
1
3       2.268915      0.160516         0.045153        0.005852
10
```

```
4       2.186963      0.119889         0.041423        0.002915
100
5       2.184774      0.096105         0.039226        0.002758
1000

        params  split0_test_score  split1_test_score
split2_test_score  \
0  {'C': 0.01}           0.988044           0.988452
0.988304
1   {'C': 0.1}           0.988055           0.988465
0.988316
2     {'C': 1}           0.988053           0.988467
0.988318
3    {'C': 10}           0.988052           0.988467
0.988319
4   {'C': 100}           0.988053           0.988467
0.988319
5  {'C': 1000}           0.988053           0.988467
0.988319

   split3_test_score  split4_test_score  mean_test_score
std_test_score  \
0           0.988370           0.988414         0.988317
0.000145
1           0.988373           0.988423         0.988326
0.000145
2           0.988372           0.988428         0.988328
0.000146
3           0.988372           0.988428         0.988328
0.000147
4           0.988372           0.988428         0.988328
0.000146
5           0.988372           0.988428         0.988328
0.000146

   rank_test_score  split0_train_score  split1_train_score  \
0                6            0.988367            0.988261
1                5            0.988378            0.988271
2                4            0.988378            0.988271
3                3            0.988379            0.988272
4                1            0.988379            0.988272
5                2            0.988379            0.988272

   split2_train_score  split3_train_score  split4_train_score  \
0            0.988339            0.988415            0.988305
1            0.988345            0.988423            0.988312
2            0.988344            0.988421            0.988312
3            0.988344            0.988421            0.988312
4            0.988344            0.988421            0.988312
5            0.988344            0.988421            0.988312
```

```
   mean_train_score  std_train_score
0          0.988337         0.000053
1          0.988346         0.000052
2          0.988345         0.000052
3          0.988346         0.000052
4          0.988346         0.000052
5          0.988346         0.000052

# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('roc_auc')
plt.legend(['test result', 'train result'], loc='upper left')
plt.xscale('log')
```



```
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test roc_auc is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test roc_auc is 0.988327779649056 at C = 100
```

Logistic regression with optimal C

```
# Instantiate the model with best C
logistic_bal_ros = LogisticRegression(C=0.1)

# Fit the model on the train set
logistic_bal_ros_model = logistic_bal_ros.fit(X_train_ros, y_train_ros)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = logistic_bal_ros_model.predict(X_train_ros)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_pred)
print(confusion)
```

```
[[222237   5212]
 [ 17959 209490]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_ros, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_train_ros, y_train_pred))
```

```
Accuracy:- 0.949063306499479
Sensitivity:- 0.9210416401039354
Specificity:- 0.9770849728950226
F1-Score:- 0.9475948262019084
```

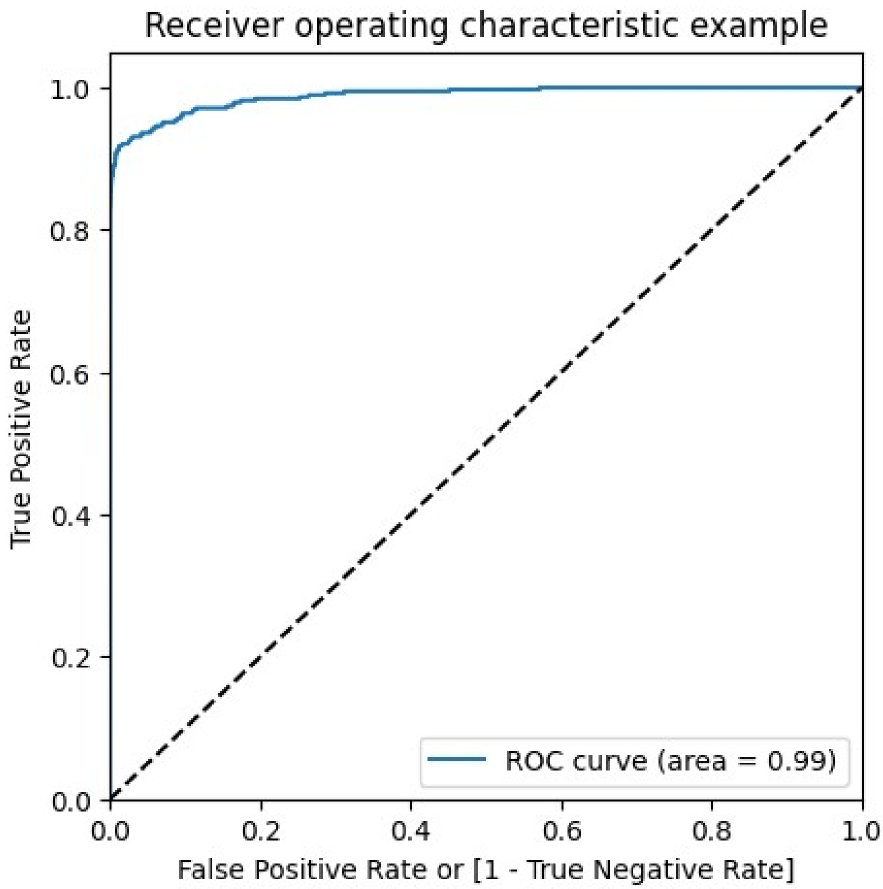
```python
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95    227449
           1       0.98      0.92      0.95    227449

    accuracy                           0.95    454898
   macro avg       0.95      0.95      0.95    454898
weighted avg       0.95      0.95      0.95    454898
```

```python
# Predicted probability
y_train_pred_proba = logistic_bal_ros_model.predict_proba(X_train_ros)
[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

```
0.9883382343883345
```

```python
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

```
# Prediction on the test set
y_test_pred = logistic_bal_ros_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)

[[55531  1335]
 [   11    85]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))
```

```python
# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9763702117200941
Sensitivity:- 0.8854166666666666
Specificity:- 0.9765237576055992
```

```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

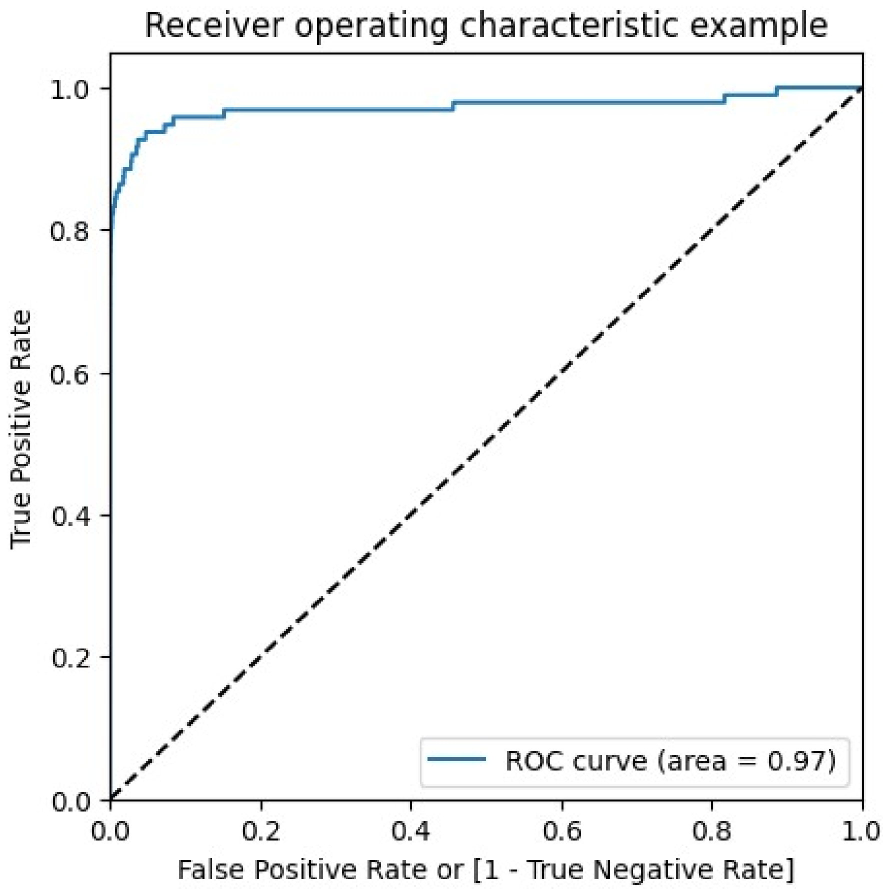
```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.06      0.89      0.11        96

    accuracy                           0.98     56962
   macro avg       0.53      0.93      0.55     56962
weighted avg       1.00      0.98      0.99     56962
```

```python
# Predicted probability
y_test_pred_proba = logistic_bal_ros_model.predict_proba(X_test)[:,1]
```

```python
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9714092120071747
```

```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.95
  - Sensitivity = 0.92
  - Specificity = 0.97
  - ROC = 0.98
- Test set
  - Accuracy = 0.97
  - Sensitivity = 0.89
  - Specificity = 0.97
  - ROC = 0.97

XGBoost

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
```

```
             'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train_ros, y_train_ros)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

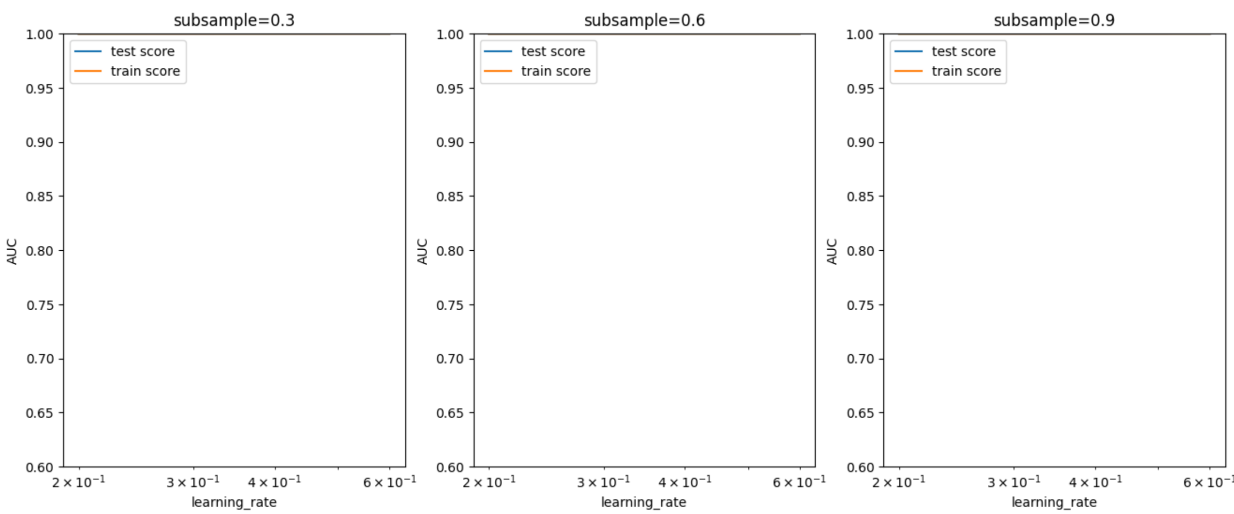
```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None,
colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False,
eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None,
importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_cat_threshold=None,
                                     max_cat_to_onehot=None,
                                     max_delta_step=None, max_depth=2,
                                     max_leaves=None,
min_child_weight=None,
                                     missing=nan,
monotone_constraints=None,
                                     multi_strategy=None,
n_estimators=200,
                                     n_jobs=None,
num_parallel_tree=None,
                                     random_state=None, ...),
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```python
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       3.700946      0.591655         0.144920        0.034687   
1       5.044792      0.300841         0.141428        0.022726   
2       4.751309      1.015213         0.111906        0.001667   
3       4.852920      0.447064         0.168766        0.043294   
4       5.497935      0.152418         0.150439        0.026223   
5       5.396589      0.574716         0.149871        0.035196   

  param_learning_rate param_subsample  \
0                 0.2             0.3   
1                 0.2             0.6   
2                 0.2             0.9   
3                 0.6             0.3   
4                 0.6             0.6   
5                 0.6             0.9   

                                     params  split0_test_score  \
0  {'learning_rate': 0.2, 'subsample': 0.3}           0.999912   
1  {'learning_rate': 0.2, 'subsample': 0.6}           0.999897   
2  {'learning_rate': 0.2, 'subsample': 0.9}           0.999897   
3  {'learning_rate': 0.6, 'subsample': 0.3}           0.999981   
4  {'learning_rate': 0.6, 'subsample': 0.6}           0.999987   
5  {'learning_rate': 0.6, 'subsample': 0.9}           0.999994   

   split1_test_score  split2_test_score  mean_test_score
std_test_score  \
0           0.999911           0.999893         0.999905    8.906213e-
06   
1           0.999914           0.999899         0.999903    7.858527e-
06   
2           0.999918           0.999902         0.999906    8.867462e-
06   
3           0.999983           0.999983         0.999982    8.882476e-
07   
4           0.999979           0.999975         0.999981    5.147556e-
06   
5           0.999977           0.999967         0.999979    1.112796e-
05   

   rank_test_score  split0_train_score  split1_train_score  \
0                5            0.999921            0.999917   
1                6            0.999906            0.999917   
2                4            0.999913            0.999919   
3                1            0.999992            0.999993   
4                2            0.999996            0.999997   
5                3            1.000000            0.999993   
```

```
   split2_train_score  mean_train_score  std_train_score
0            0.999912          0.999917     3.778849e-06
1            0.999924          0.999916     7.007515e-06
2            0.999922          0.999918     3.807108e-06
3            0.999998          0.999995     2.745049e-06
4            0.999997          0.999997     4.470902e-07
5            1.000000          0.999998     3.013075e-06
```

```python
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
             'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



Model with optimal hyperparameters

We see that the train score almost touches to 1. Among the hyperparameters, we can choose the best parameters as learning_rate : 0.2 and subsample: 0.3

```
model_cv.best_params_
```

```
{'learning_rate': 0.6, 'subsample': 0.3}
```

```
# chosen hyperparameters
params = {'learning_rate': 0.6,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.9,
         'objective':'binary:logistic'}

# fit model on training data
xgb_bal_ros_model = XGBClassifier(params = params)
xgb_bal_ros_model.fit(X_train_ros, y_train_ros)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None,
              params={'learning_rate': 0.6, 'max_depth': 2,
'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample':
0.9}, ...)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = xgb_bal_ros_model.predict(X_train_ros)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_ros)
print(confusion)
```

```
[[227449      0]
 [     0 227449]]
```

```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_ros, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 1.0
Sensitivity:- 1.0
Specificity:- 1.0
```

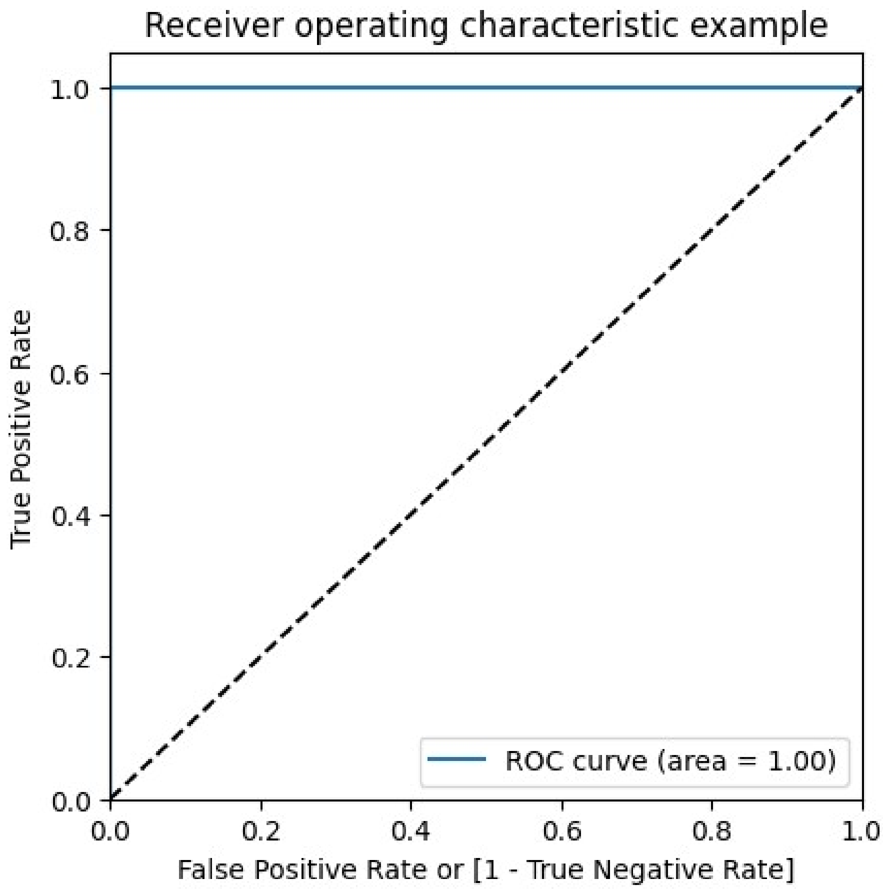
```python
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227449

    accuracy                           1.00    454898
   macro avg       1.00      1.00      1.00    454898
weighted avg       1.00      1.00      1.00    454898
```

```python
# Predicted probability
y_train_pred_proba = xgb_bal_ros_model.predict_proba(X_train_ros)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

```
1.0
```

```python
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

```
# Predictions on the test set
y_test_pred = xgb_bal_ros_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56854    12]
 [   21    75]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))
```

```
# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.999420666409185
Sensitivity:- 0.78125
Specificity:- 0.9997889775964548
```

```
# classification_report
print(classification_report(y_test, y_test_pred))
```

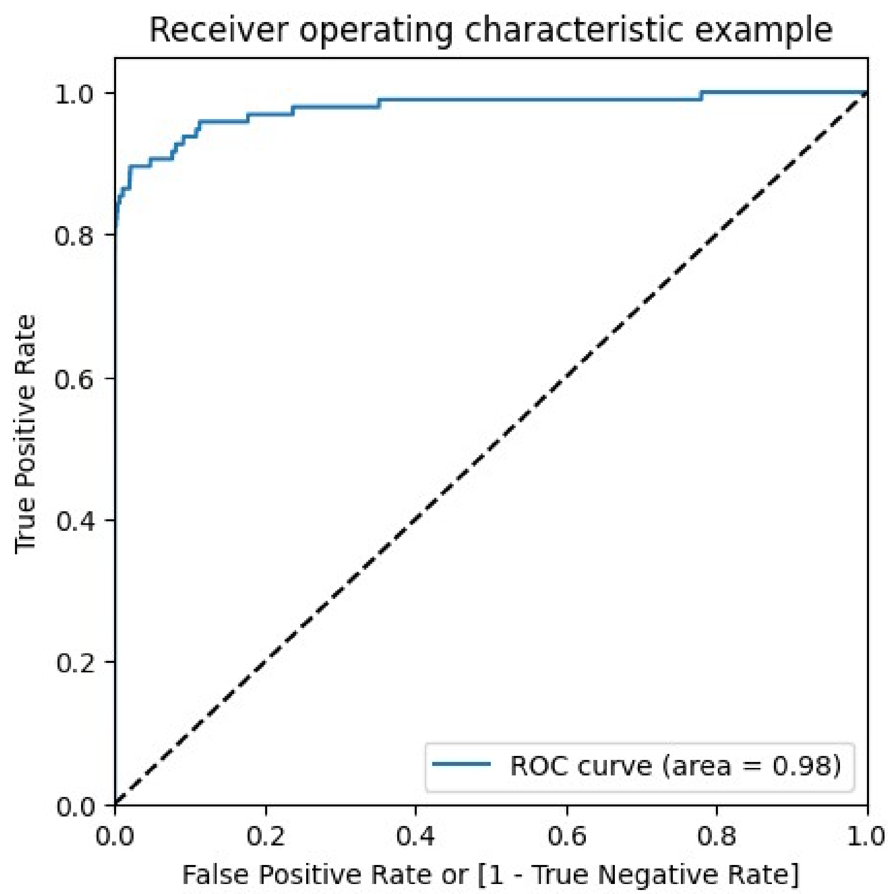
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.86      0.78      0.82        96

    accuracy                           1.00     56962
   macro avg       0.93      0.89      0.91     56962
weighted avg       1.00      1.00      1.00     56962
```

```
# Predicted probability
y_test_pred_proba = xgb_bal_ros_model.predict_proba(X_test)[:,1]
```

```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9775024839095418
```

```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 1.0
  - Sensitivity = 1.0
  - Specificity = 1.0
  - ROC-AUC = 1.0
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.80
  - Specificity = 0.99
  - ROC-AUC = 0.97

SMOTE (Synthetic Minority Oversampling Technique)

We are creating synthetic samples by doing upsampling using SMOTE(Synthetic Minority Oversampling Technique).

```
# Importing SMOTE
from imblearn.over_sampling import SMOTE
```

```
# Instantiate SMOTE
sm = SMOTE(random_state=27)
# Fitting SMOTE to the train set
X_train_smote, y_train_smote = sm.fit_resample(X_train, y_train)

print('Before SMOTE oversampling X_train shape=',X_train.shape)
print('After SMOTE oversampling X_train shape=',X_train_smote.shape)
```

```
Before SMOTE oversampling X_train shape= (227845, 29)
After SMOTE oversampling X_train shape= (454898, 29)
```

Logistic Regression

```
# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as roc-auc
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train_smote, y_train_smote)
```

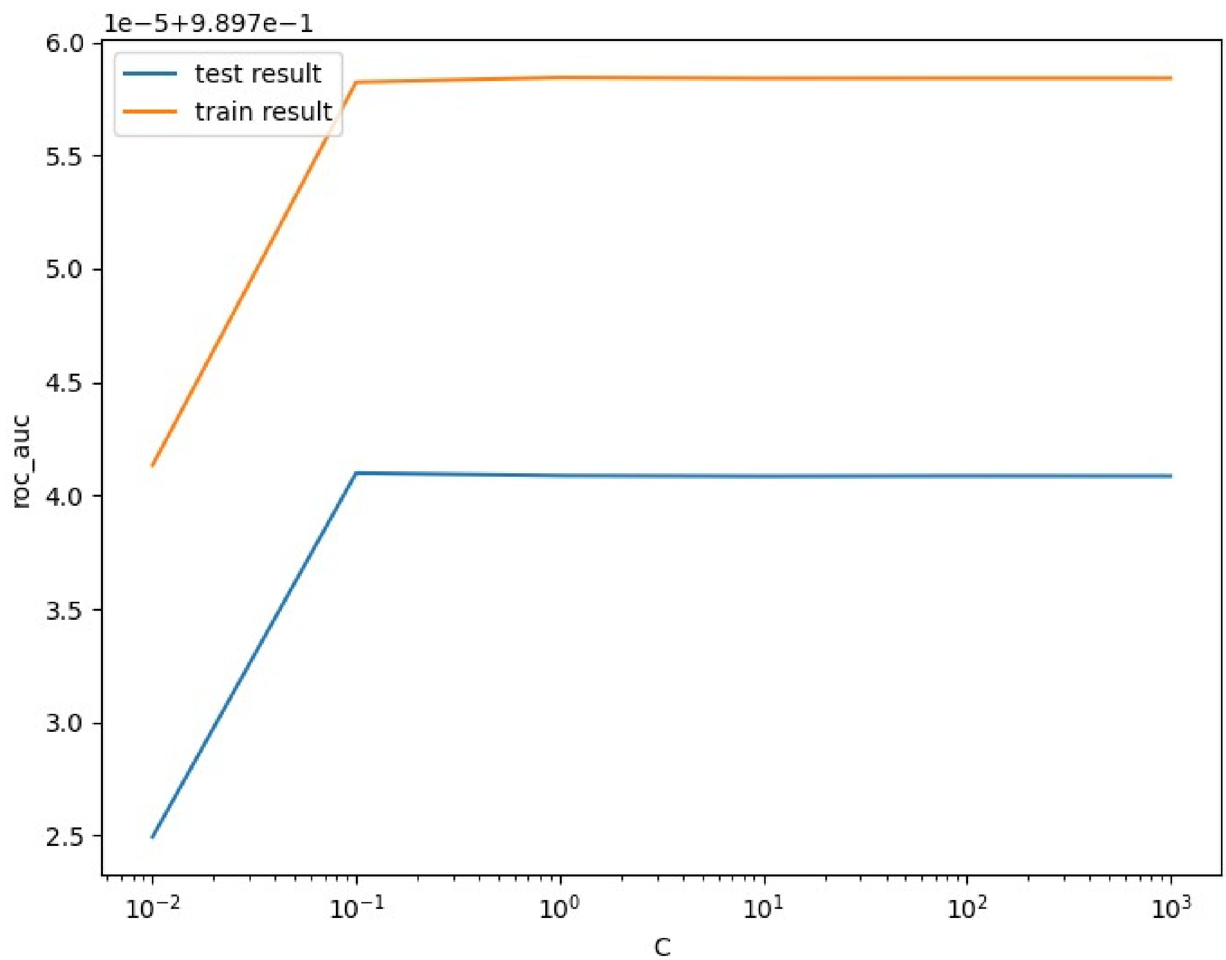
```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
param_C  \
0       4.350599      0.614689         0.101291        0.034046
0.01
1       3.001161      0.427049         0.083746        0.058739
0.1
2       1.915668      0.158935         0.050117        0.023027
1
3       2.146231      0.321787         0.070252        0.049219
10
```

```
4       2.051609      0.502301         0.046772        0.013744
100
5       2.154828      0.204071         0.062755        0.044003
1000

        params  split0_test_score  split1_test_score
split2_test_score  \
0  {'C': 0.01}           0.989805           0.989796
0.989484
1   {'C': 0.1}           0.989834           0.989807
0.989488
2     {'C': 1}           0.989836           0.989807
0.989486
3    {'C': 10}           0.989836           0.989807
0.989486
4   {'C': 100}           0.989836           0.989807
0.989486
5  {'C': 1000}           0.989836           0.989807
0.989486

   split3_test_score  split4_test_score  mean_test_score
std_test_score  \
0           0.989631           0.989910         0.989725
0.000150
1           0.989632           0.989942         0.989741
0.000161
2           0.989630           0.989944         0.989741
0.000162
3           0.989630           0.989945         0.989741
0.000163
4           0.989630           0.989945         0.989741
0.000163
5           0.989630           0.989945         0.989741
0.000163

   rank_test_score  split0_train_score  split1_train_score  \
0                6            0.989758            0.989666
1                1            0.989780            0.989686
2                2            0.989781            0.989687
3                5            0.989781            0.989687
4                3            0.989781            0.989687
5                4            0.989781            0.989687

   split2_train_score  split3_train_score  split4_train_score  \
0            0.989760            0.989841            0.989682
1            0.989772            0.989853            0.989700
2            0.989772            0.989852            0.989701
3            0.989772            0.989852            0.989701
4            0.989772            0.989852            0.989701
5            0.989772            0.989852            0.989701
```

```
   mean_train_score  std_train_score
0          0.989741         0.000063
1          0.989758         0.000060
2          0.989758         0.000060
3          0.989758         0.000060
4          0.989758         0.000060
5          0.989758         0.000060

# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('roc_auc')
plt.legend(['test result', 'train result'], loc='upper left')
plt.xscale('log')
```



```
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test roc_auc is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test roc_auc is 0.9897409900830768 at C = 0.1
```

Logistic regression with optimal C

```
# Instantiate the model with best C
logistic_bal_smote = LogisticRegression(C=0.1)

# Fit the model on the train set
logistic_bal_smote_model = logistic_bal_smote.fit(X_train_smote, y_train_smote)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = logistic_bal_smote_model.predict(X_train_smote)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
print(confusion)
```

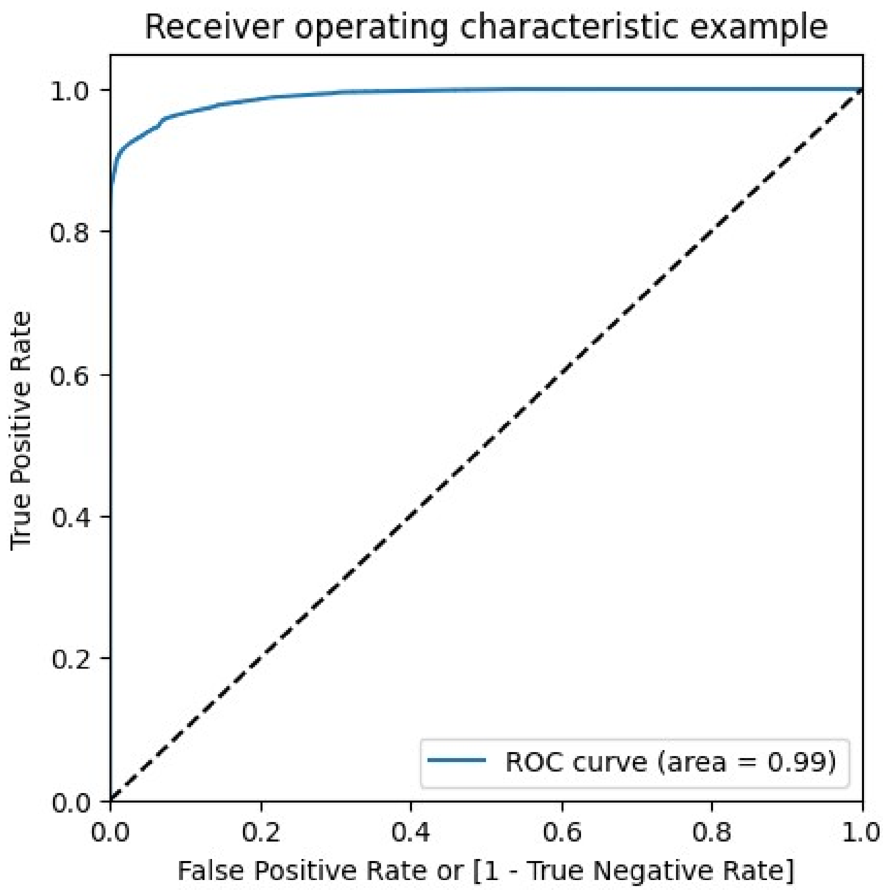
```
[[221911   5538]
 [ 17693 209756]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_smote, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9489314087993352
Sensitivity:- 0.9222111330452102
Specificity:- 0.9756516845534603
```

```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95    227449
           1       0.97      0.92      0.95    227449

    accuracy                           0.95    454898
   macro avg       0.95      0.95      0.95    454898
weighted avg       0.95      0.95      0.95    454898


# Predicted probability
y_train_pred_proba_log_bal_smote =
logistic_bal_smote_model.predict_proba(X_train_smote)[:,1]

# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba_log_bal_smote)
```



Prediction on the test set

```
# Prediction on the test set
y_test_pred = logistic_bal_smote_model.predict(X_test)
```

```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55416  1450]
 [   10    86]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

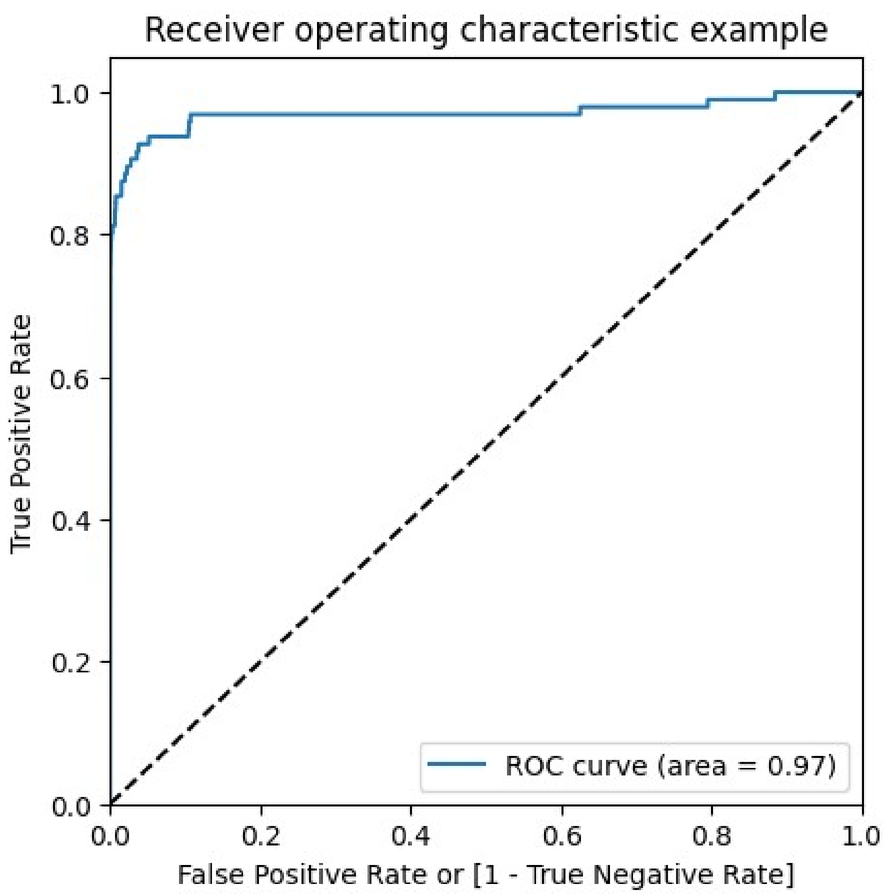
```
Accuracy:- 0.9743688774972789
Sensitivity:- 0.8958333333333334
Specificity:- 0.9745014595716245
```

```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     56866
           1       0.06      0.90      0.11        96

    accuracy                           0.97     56962
   macro avg       0.53      0.94      0.55     56962
weighted avg       1.00      0.97      0.99     56962
```

ROC on the test set

```
# Predicted probability
y_test_pred_proba = logistic_bal_smote_model.predict_proba(X_test)
[:,1]

# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.95
  - Sensitivity = 0.92
  - Specificity = 0.98
  - ROC = 0.99
- Test set
  - Accuracy = 0.97
  - Sensitivity = 0.90
  - Specificity = 0.99
  - ROC = 0.97

XGBoost

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
```

```
             'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train_smote, y_train_smote)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

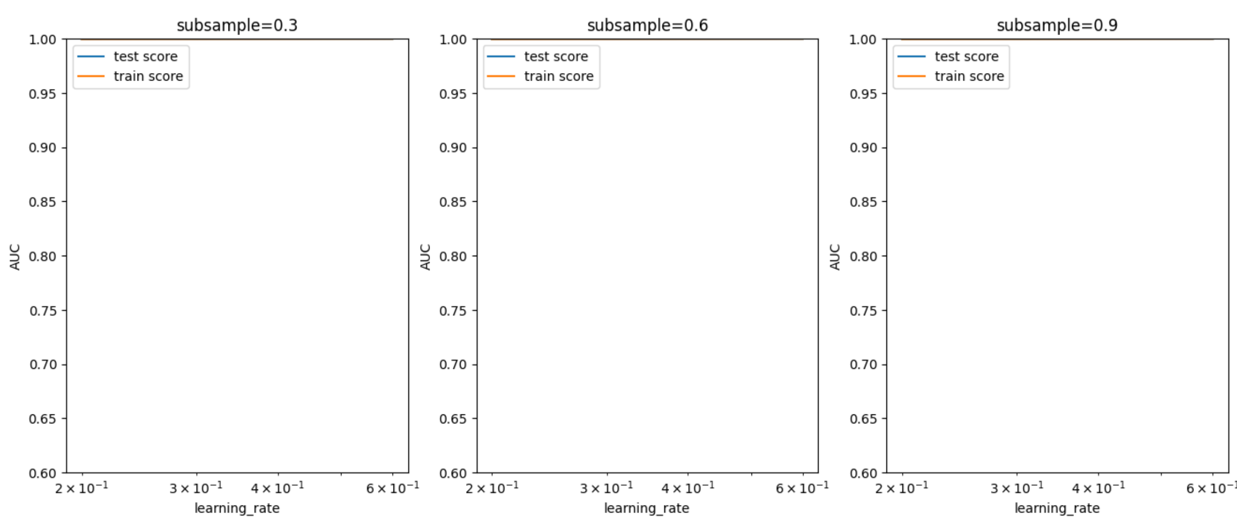
```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None,
colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False,
eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None,
importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_cat_threshold=None,
                                     max_cat_to_onehot=None,
                                     max_delta_step=None, max_depth=2,
                                     max_leaves=None,
min_child_weight=None,
                                     missing=nan,
monotone_constraints=None,
                                     multi_strategy=None,
n_estimators=200,
                                     n_jobs=None,
num_parallel_tree=None,
                                     random_state=None, ...),
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```python
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       4.830210      0.723539         0.153594        0.018271   
1       5.653752      0.740742         0.153280        0.030025   
2       5.075618      0.739584         0.200497        0.054708   
3       6.559366      1.417073         0.250432        0.058185   
4       5.646797      0.922618         0.180053        0.011094   
5       5.549089      1.385095         0.179097        0.053972   

  param_learning_rate param_subsample  \
0                 0.2             0.3   
1                 0.2             0.6   
2                 0.2             0.9   
3                 0.6             0.3   
4                 0.6             0.6   
5                 0.6             0.9   

                                     params  split0_test_score  \
0  {'learning_rate': 0.2, 'subsample': 0.3}           0.999675   
1  {'learning_rate': 0.2, 'subsample': 0.6}           0.999648   
2  {'learning_rate': 0.2, 'subsample': 0.9}           0.999657   
3  {'learning_rate': 0.6, 'subsample': 0.3}           0.999932   
4  {'learning_rate': 0.6, 'subsample': 0.6}           0.999964   
5  {'learning_rate': 0.6, 'subsample': 0.9}           0.999963   

   split1_test_score  split2_test_score  mean_test_score  
std_test_score  \
0           0.999729           0.999679         0.999694  
0.000024   
1           0.999719           0.999656         0.999674  
0.000032   
2           0.999730           0.999654         0.999680  
0.000035   
3           0.999958           0.999948         0.999946  
0.000011   
4           0.999953           0.999957         0.999958  
0.000005   
5           0.999949           0.999958         0.999957  
0.000006   

   rank_test_score  split0_train_score  split1_train_score  \
0                4            0.999725            0.999712   
1                6            0.999702            0.999709   
2                5            0.999712            0.999714   
3                3            0.999967            0.999968   
4                1            0.999977            0.999979   
5                2            0.999977            0.999976   
```

```
   split2_train_score  mean_train_score  std_train_score
0            0.999720          0.999719         0.000005
1            0.999721          0.999711         0.000008
2            0.999703          0.999710         0.000005
3            0.999977          0.999971         0.000005
4            0.999979          0.999978         0.000001
5            0.999981          0.999978         0.000002
```

```
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
             'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



Model with optimal hyperparameters

We see that the train score almost touches to 1. Among the hyperparameters, we can choose the best parameters as learning_rate : 0.2 and subsample: 0.3

```
model_cv.best_params_

{'learning_rate': 0.6, 'subsample': 0.6}

# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label,
which we need for calculating auc
params = {'learning_rate': 0.6,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.9,
         'objective':'binary:logistic'}

# fit model on training data
xgb_bal_smote_model = XGBClassifier(params = params)
xgb_bal_smote_model.fit(X_train_smote, y_train_smote)

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None,
              params={'learning_rate': 0.6, 'max_depth': 2,
'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample':
0.9}, ...)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = xgb_bal_smote_model.predict(X_train_smote)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
print(confusion)
```

```
[[227448      1]
 [     0 227449]]
```

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_smote,
y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9999978017049976
Sensitivity:- 1.0
Specificity:- 0.9999956034099952
```

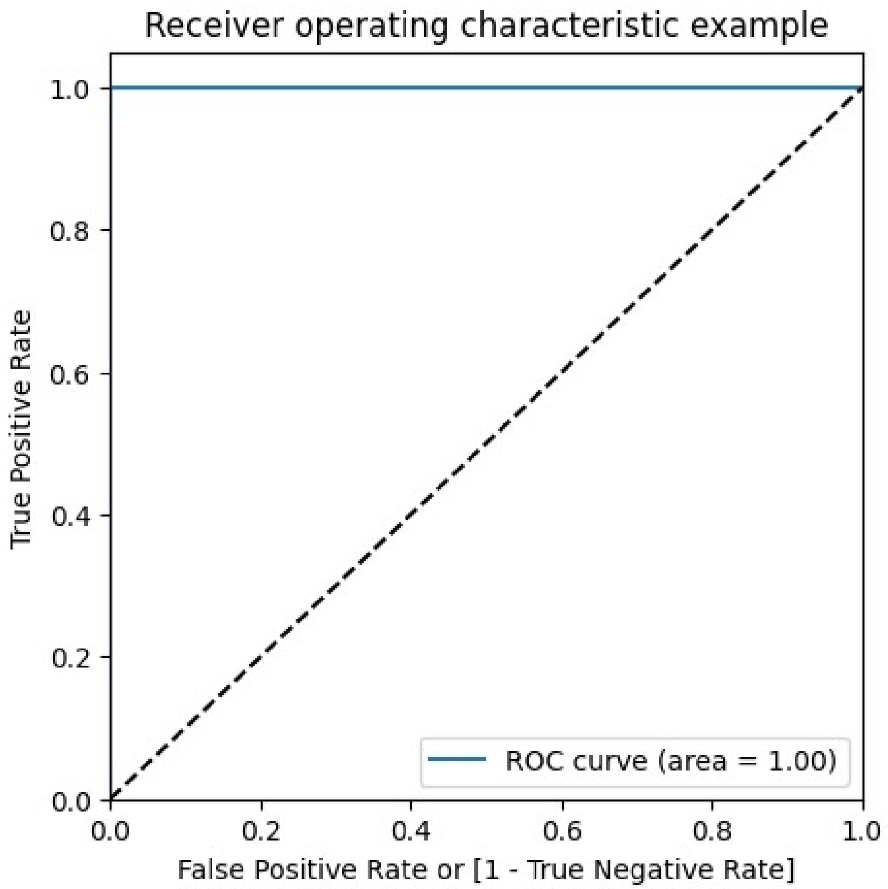
```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227449

    accuracy                           1.00    454898
   macro avg       1.00      1.00      1.00    454898
weighted avg       1.00      1.00      1.00    454898
```

```
# Predicted probability
y_train_pred_proba = xgb_bal_smote_model.predict_proba(X_train_smote)
[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_smote, y_train_pred_proba)
auc
```

```
0.9999999890785479
```

```
# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba)
```



Prediction on the test set

```
# Predictions on the test set
y_test_pred = xgb_bal_smote_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)

[[56833    33]
 [   20    76]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))
```

```python
# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9990695551420246
Sensitivity:- 0.7916666666666666
Specificity:- 0.9994196883902507
```

```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

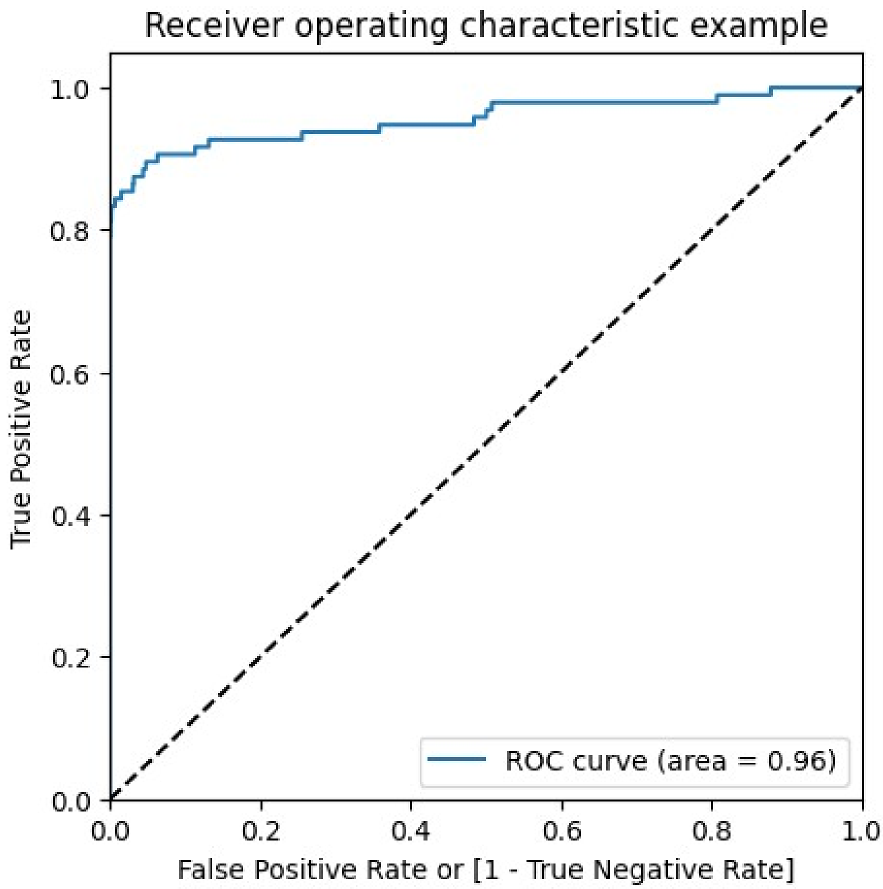
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.70      0.79      0.74        96

    accuracy                           1.00     56962
   macro avg       0.85      0.90      0.87     56962
weighted avg       1.00      1.00      1.00     56962
```

```python
# Predicted probability
y_test_pred_proba = xgb_bal_smote_model.predict_proba(X_test)[:,1]
```

```python
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9553290117703608
```

```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.99
  - Sensitivity = 1.0
  - Specificity = 0.99
  - ROC-AUC = 1.0
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.79
  - Specificity = 0.99
  - ROC-AUC = 0.96

Overall, the model is performing well in the test set, what it had learnt from the train set.

AdaSyn (Adaptive Synthetic Sampling)

```
# Importing adasyn
from imblearn.over_sampling import ADASYN
```

```
# Instantiate adasyn
ada = ADASYN(random_state=0)
X_train_adasyn, y_train_adasyn = ada.fit_resample(X_train, y_train)

# Befor sampling class distribution
print('Before sampling class distribution:-',Counter(y_train))
# new class distribution
print('New class distribution:-',Counter(y_train_adasyn))
```

```
Before sampling class distribution:- Counter({0: 227449, 1: 396})
New class distribution:- Counter({0: 227449, 1: 227448})
```

Logistic Regression

```
# Creating KFold object with 3 splits
folds = KFold(n_splits=3, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as roc-auc
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train_adasyn, y_train_adasyn)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

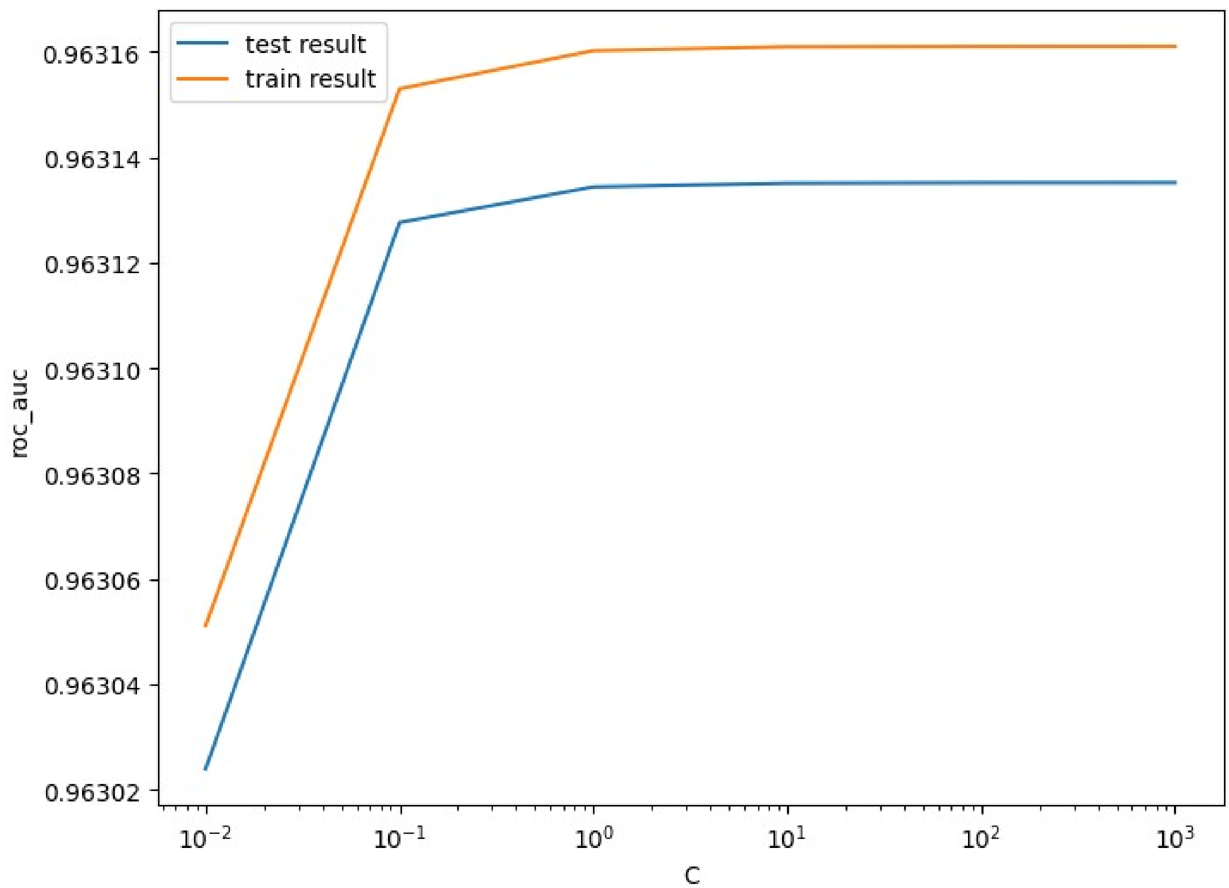
```
GridSearchCV(cv=KFold(n_splits=3, random_state=4, shuffle=True),
             estimator=LogisticRegression(),
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
param_C  \
0       2.829567      0.559018         0.129948        0.010375
0.01
1       2.194193      0.401066         0.106578        0.028898
0.1
2       2.973076      0.186132         0.092796        0.010743
1
3       2.391114      0.050318         0.093079        0.003281
```

```
10
4       2.126840      0.358569         0.092225        0.015129
100
5       2.536523      0.508635         0.136290        0.053766
1000

        params  split0_test_score  split1_test_score
split2_test_score  \
0  {'C': 0.01}           0.963472           0.962327
0.963273
1   {'C': 0.1}           0.963578           0.962435
0.963370
2     {'C': 1}           0.963585           0.962442
0.963376
3    {'C': 10}           0.963585           0.962443
0.963377
4   {'C': 100}           0.963585           0.962443
0.963377
5  {'C': 1000}           0.963585           0.962443
0.963377

   mean_test_score  std_test_score  rank_test_score
split0_train_score  \
0         0.963024        0.000499                6
0.962770
1         0.963128        0.000497                5
0.962881
2         0.963134        0.000497                4
0.962890
3         0.963135        0.000496                3
0.962891
4         0.963135        0.000496                2
0.962891
5         0.963135        0.000496                1
0.962891

   split1_train_score  split2_train_score  mean_train_score
std_train_score
0            0.963211            0.963172          0.963051
0.000199
1            0.963305            0.963272          0.963153
0.000192
2            0.963312            0.963278          0.963160
0.000191
3            0.963312            0.963279          0.963161
0.000191
4            0.963312            0.963279          0.963161
0.000191
5            0.963312            0.963279          0.963161
0.000191
```

```
# plot of C versus train and validation scores

plt.figure(figsize=(8, 6))
plt.plot(cv_results['param_C'], cv_results['mean_test_score'])
plt.plot(cv_results['param_C'], cv_results['mean_train_score'])
plt.xlabel('C')
plt.ylabel('roc_auc')
plt.legend(['test result', 'train result'], loc='upper left')
plt.xscale('log')
```




```
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test roc_auc is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test roc_auc is 0.963135148223901 at C = 1000
```

Logistic regression with optimal C

```
# Instantiate the model with best C
logistic_bal_adasyn = LogisticRegression(C=1000)

# Fit the model on the train set
logistic_bal_adasyn_model = logistic_bal_adasyn.fit(X_train_adasyn,
y_train_adasyn)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = logistic_bal_adasyn_model.predict(X_train_adasyn)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_adasyn, y_train_pred)
print(confusion)
```

```
[[207019  20430]
 [ 31286 196162]]
```

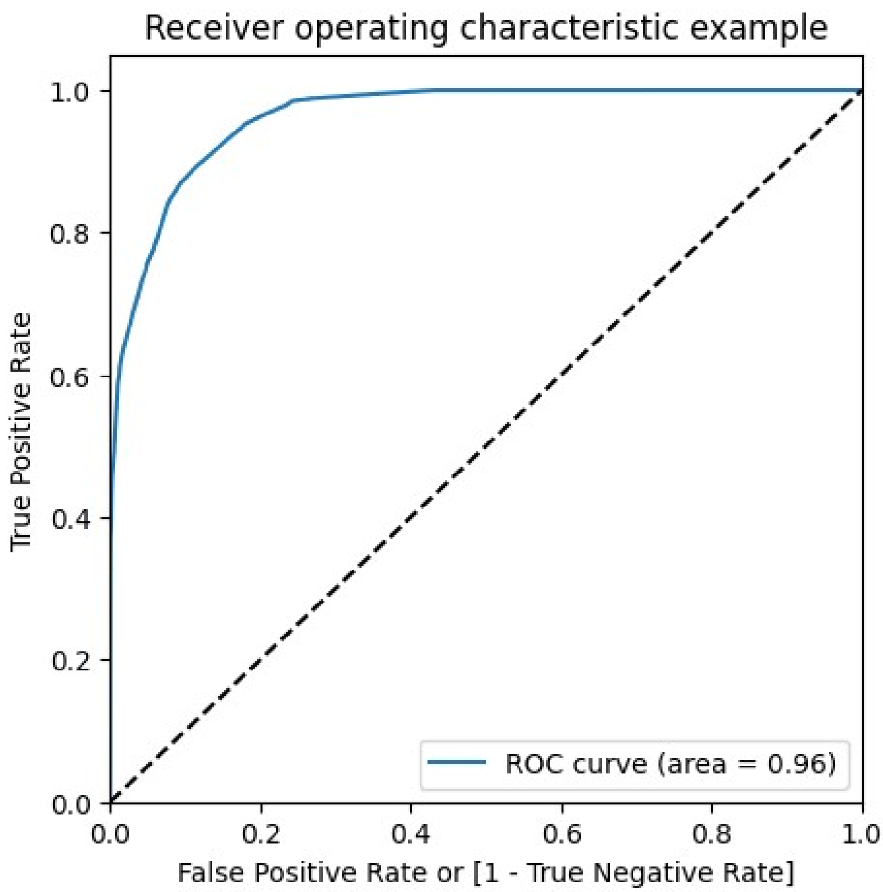
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_adasyn,
y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))

# F1 score
print("F1-Score:-", f1_score(y_train_adasyn, y_train_pred))
```

```
Accuracy:- 0.8863127257379143
Sensitivity:- 0.862447680348915
Specificity:- 0.9101776662020936
F1-Score:- 0.8835330150436899
```

```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.91      0.89    227449
           1       0.91      0.86      0.88    227448

    accuracy                           0.89    454897
```

```
   macro avg       0.89      0.89      0.89    454897
weighted avg       0.89      0.89      0.89    454897


# Predicted probability
y_train_pred_proba =
logistic_bal_adasyn_model.predict_proba(X_train_adasyn)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc

0.9631610160068508

# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

```
# Prediction on the test set
y_test_pred = logistic_bal_adasyn_model.predict(X_test)
```

```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[51642  5224]
 [    4    92]]
```

```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9082195147642288
Sensitivity:- 0.9583333333333334
Specificity:- 0.9081349136566665
```

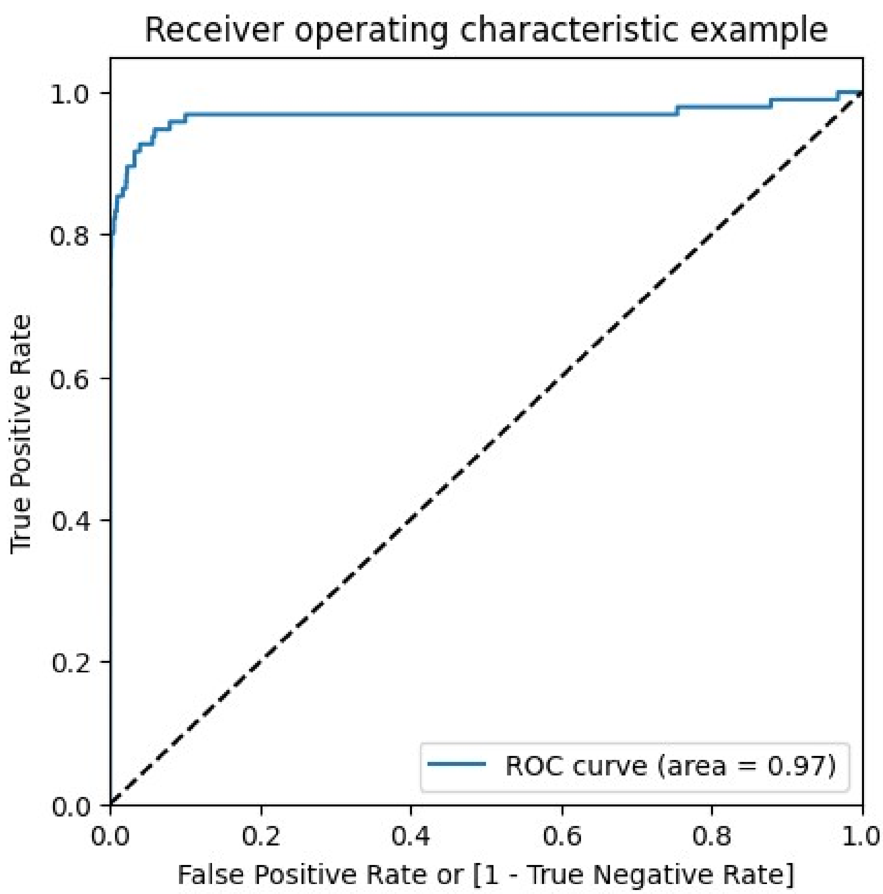
```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.91      0.95     56866
           1       0.02      0.96      0.03        96

    accuracy                           0.91     56962
   macro avg       0.51      0.93      0.49     56962
weighted avg       1.00      0.91      0.95     56962
```

```python
# Predicted probability
y_test_pred_proba = logistic_bal_adasyn_model.predict_proba(X_test)
[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9671573487086602
```

```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.88
  - Sensitivity = 0.86
  - Specificity = 0.91
  - ROC = 0.96
- Test set
  - Accuracy = 0.90
  - Sensitivity = 0.95
  - Specificity = 0.90
  - ROC = 0.97

XGBoost

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
```

```python
             'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train_adasyn, y_train_adasyn)
```

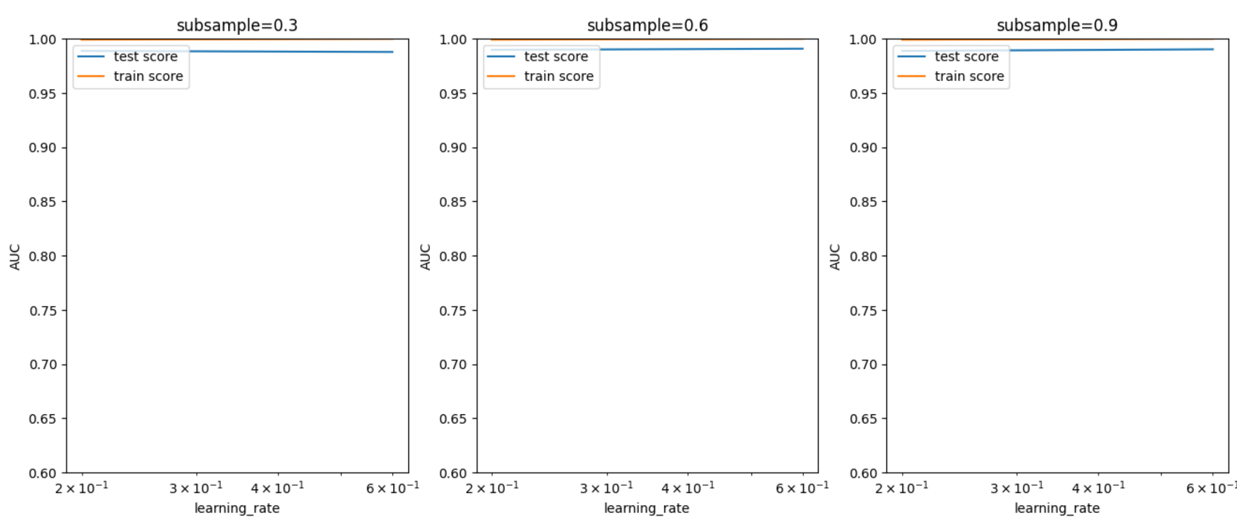
```
Fitting 3 folds for each of 6 candidates, totalling 18 fits

GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None,
colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None,
device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False,
eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None,
importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_cat_threshold=None,
                                     max_cat_to_onehot=None,
                                     max_delta_step=None, max_depth=2,
                                     max_leaves=None,
min_child_weight=None,
                                     missing=nan,
monotone_constraints=None,
                                     multi_strategy=None,
n_estimators=200,
                                     n_jobs=None,
num_parallel_tree=None,
                                     random_state=None, ...),
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             return_train_score=True, scoring='roc_auc', verbose=1)
```

```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       3.846837      0.434049         0.108397        0.004416   
1       3.700305      0.469272         0.136020        0.030375   
2       4.342210      0.903681         0.153409        0.020724   
3       3.817883      0.039716         0.116133        0.012226   
4       9.147208      0.427080         0.254515        0.022016   
5       7.270713      1.113050         0.227000        0.053686   

  param_learning_rate param_subsample  \
0                 0.2             0.3   
1                 0.2             0.6   
2                 0.2             0.9   
3                 0.6             0.3   
4                 0.6             0.6   
5                 0.6             0.9   

                                     params  split0_test_score  \
0  {'learning_rate': 0.2, 'subsample': 0.3}           0.975484   
1  {'learning_rate': 0.2, 'subsample': 0.6}           0.978568   
2  {'learning_rate': 0.2, 'subsample': 0.9}           0.975497   
3  {'learning_rate': 0.6, 'subsample': 0.3}           0.970280   
4  {'learning_rate': 0.6, 'subsample': 0.6}           0.980396   
5  {'learning_rate': 0.6, 'subsample': 0.9}           0.978029   

   split1_test_score  split2_test_score  mean_test_score
std_test_score  \
0           0.996111           0.994796         0.988797
0.009429   
1           0.996275           0.994438         0.989761
0.007949   
2           0.995795           0.995089         0.988794
0.009407   
3           0.995986           0.997243         0.987836
0.012425   
4           0.995514           0.996559         0.990823
0.007385   
5           0.995482           0.997311         0.990274
0.008691   

   rank_test_score  split0_train_score  split1_train_score  \
0                4            0.999302            0.998994   
1                3            0.999290            0.998966   
2                5            0.999231            0.998957   
3                6            0.999918            0.999933   
4                1            0.999940            0.999937   
5                2            0.999930            0.999936   
```

	split2_train_score	mean_train_score	std_train_score
0	0.999229	0.999175	0.000132
1	0.999181	0.999146	0.000135
2	0.999148	0.999112	0.000115
3	0.999941	0.999931	0.000009
4	0.999940	0.999939	0.000001
5	0.999952	0.999939	0.000009

```
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
             'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```




```
model_cv.best_params_
```

```
{'learning_rate': 0.6, 'subsample': 0.6}

# chosen hyperparameters

params = {'learning_rate': 0.6,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.3,
         'objective':'binary:logistic'}

# fit model on training data
xgb_bal_adasyn_model = XGBClassifier(params = params)
xgb_bal_adasyn_model.fit(X_train_adasyn, y_train_adasyn)

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None,
              params={'learning_rate': 0.6, 'max_depth': 2,
'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample':
0.3}, ...)
```

Prediction on the train set

```
# Predictions on the train set
y_train_pred = xgb_bal_adasyn_model.predict(X_train_adasyn)

# Confusion matrix
confusion = metrics.confusion_matrix(y_train_adasyn, y_train_adasyn)
print(confusion)

[[227449      0]
 [     0 227448]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_adasyn,
y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9999934051004953
Sensitivity:- 1.0
Specificity:- 1.0
```

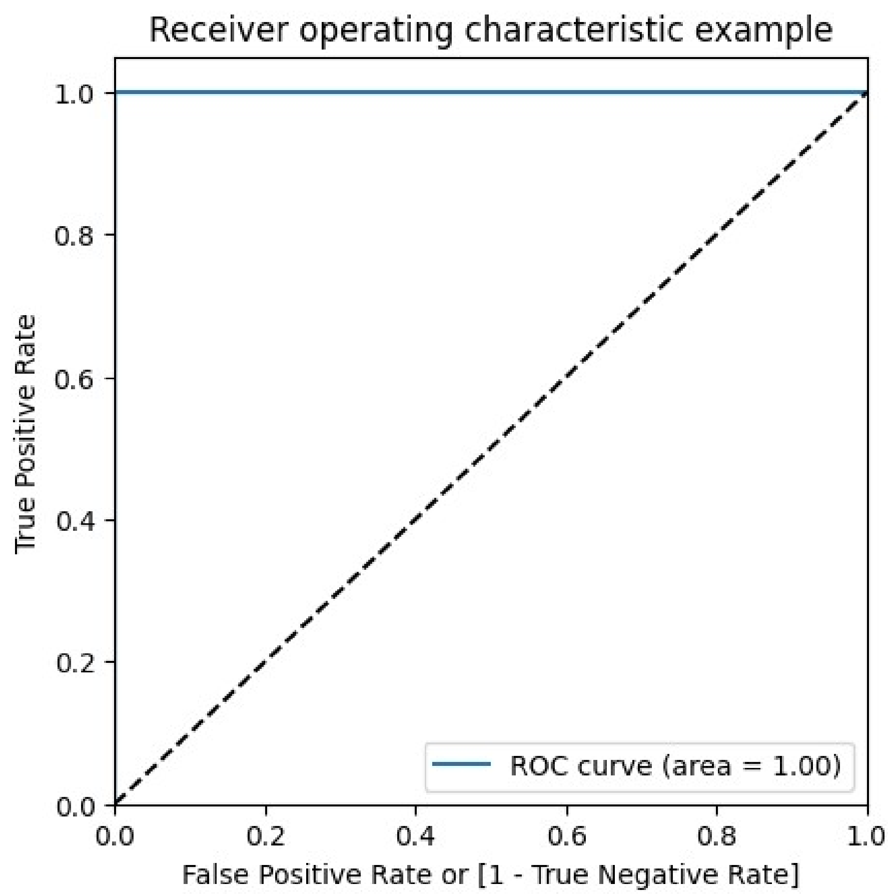
```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227448

    accuracy                           1.00    454897
   macro avg       1.00      1.00      1.00    454897
weighted avg       1.00      1.00      1.00    454897
```

```
# Predicted probability
y_train_pred_proba =
xgb_bal_adasyn_model.predict_proba(X_train_adasyn)[:,1]

# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

```
1.0
```

```
# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

```
# Predictions on the test set
y_test_pred = xgb_bal_adasyn_model.predict(X_test)

# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)

[[56828    38]
 [   22    74]]

TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives

# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))
```

```
# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.9989466661985184
Sensitivity:- 0.7708333333333334
Specificity:- 0.9993317623887736
```

```
# classification_report
print(classification_report(y_test, y_test_pred))
```

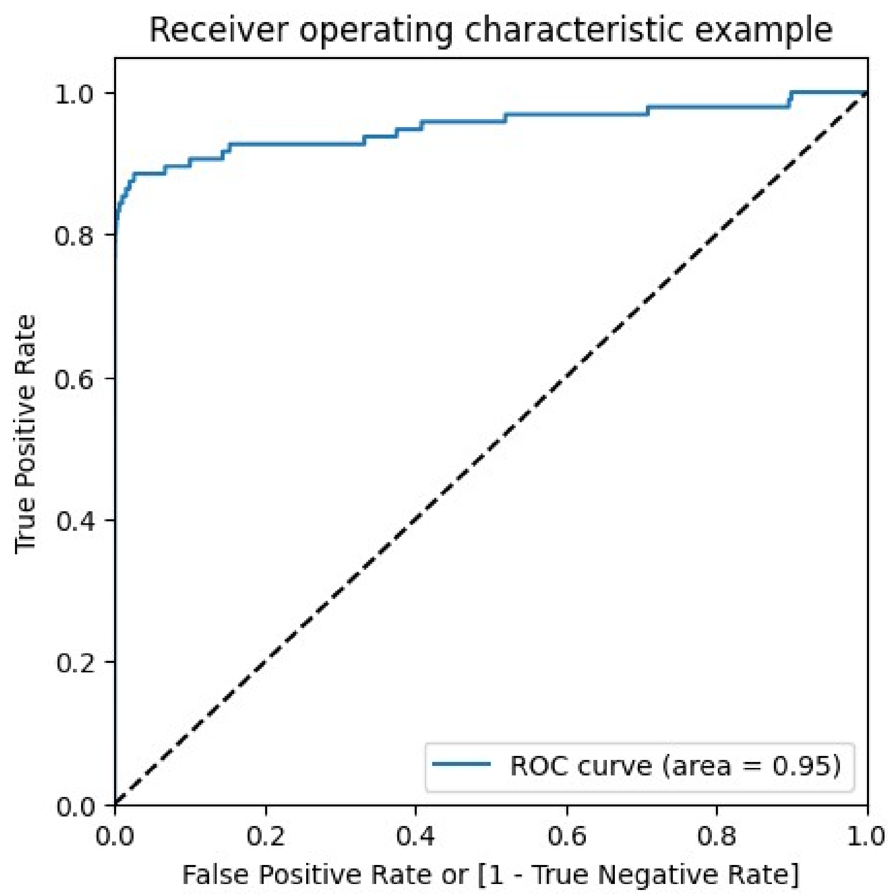
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.66      0.77      0.71        96

    accuracy                           1.00     56962
   macro avg       0.83      0.89      0.86     56962
weighted avg       1.00      1.00      1.00     56962
```

```
# Predicted probability
y_test_pred_proba = xgb_bal_adasyn_model.predict_proba(X_test)[:,1]
```

```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.951038589256615
```

```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.99
  - Sensitivity = 1.0
  - Specificity = 1.0
  - ROC-AUC = 1.0
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.78
  - Specificity = 0.99
  - ROC-AUC = 0.96

Choosing best model on the balanced data

He we balanced the data with various approach such as Undersampling, Oversampling, SMOTE and Adasy. With every data balancing thechnique we built several models such as Logistic, XGBoost, Decision Tree, and Random Forest.

We can see that almost all the models performed more or less good. But we should be interested in the best model.

Though the Undersampling technique models performed well, we should keep mind that by doing the undersampling some imformation were lost. Hence, it is better not to consider the undersampling models.

Whereas the SMOTE and Adasyn models performed well. Among those models the simplist model Logistic regression has ROC score 0.99 in the train set and 0.97 on the test set. We can consider the Logistic model as the best model to choose because of the easy interpretation of the models and also the resourse requirements to build the mdoel is lesser than the other heavy models such as Random forest or XGBoost.

Hence, we can conclude that the `Logistic regression model with SMOTE` is the best model for its simlicity and less resource requirement.

Print the FPR,TPR & select the best threshold from the roc curve for the best model

```
print('Train auc =', metrics.roc_auc_score(y_train_smote,
y_train_pred_proba_log_bal_smote))
fpr, tpr, thresholds = metrics.roc_curve(y_train_smote,
y_train_pred_proba_log_bal_smote)
threshold = thresholds[np.argmax(tpr-fpr)]
print("Threshold=",threshold)
```

```
Train auc = 0.9897539730582245
Threshold= 0.5311563616125217
```

We can see that the threshold is 0.53, for which the TPR is the highest and FPR is the lowest and we got the best ROC score.

Cost benefit analysis

We have tried several models till now with both balanced and imbalanced data. We have noticed most of the models have performed more or less well in terms of ROC score, Precision and Recall.

But while picking the best model we should consider few things such as whether we have required infrastructure, resources or computational power to run the model or not. For the models such as Random forest, SVM, XGBoost we require heavy computational resources and eventually to build that infrastructure the cost of deploying the model increases. On the other hand the simpler model such as Logistic regression requires less computational resources, so the cost of building the model is less.

We also have to consider that for little change of the ROC score how much monetary loss of gain the bank incur. If the amount if huge then we have to consider building the complex model even though the cost of building the model is high.

Summary to the business

For banks with smaller average transaction value, we would want high precision because we only want to label relevant transactions as fraudulent. For every transaction that is flagged as fraudulent, we can add the human element to verify whether the transaction was done by calling the customer. However, when precision is low, such tasks are a burden because the human element has to be increased.

For banks having a larger transaction value, if the recall is low, i.e., it is unable to detect transactions that are labelled as non-fraudulent. So we have to consider the losses if the missed transaction was a high-value fraudulent one.

So here, to save the banks from high-value fraudulent transactions, we have to focus on a high recall in order to detect actual fraudulent transactions.

After performing several models, we have seen that in the balanced dataset with SMOTE technique the simplest Logistic regression model has good ROC score and also high Recall. Hence, we can go with the logistic model here. It is also easier to interpret and explain to the business.