Capstone Project: Credit Card Fraud Detection

Problem Statement

This project examines a dataset with **284,807 credit card transactions**, of which **492 are flagged as fraudulent**. The dataset is highly imbalanced, necessitating specialized techniques to accurately train a detection model. The main objective is to build an effective machine learning model to **identify fraudulent transactions** with high precision.

Business Context

Banks prioritize retaining profitable customers, but **fraudulent transactions** pose a serious threat, leading to potential **financial losses** and risking **customer trust**. According to Nilson Report, by 2020, losses due to fraud were projected to reach **$30 billion** globally. As digital transactions grow, the incidence and types of fraudulent activity are rising, making robust fraud detection models critical for banking security.

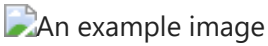
Dataset Overview

The dataset covers transactions by European cardholders over two days in September 2013. It includes **284,807 transactions**, with **fraudulent cases accounting for only 0.172%** of the total. Key details are:

- **Imbalanced Data**: With a small fraction of fraud cases, handling this imbalance is essential for model accuracy.
- **Anonymized Features (V1-V28)**: Confidentiality is preserved using **Principal Component Analysis (PCA)**, resulting in anonymized components.
- **Time**: Captures seconds elapsed from the first transaction, aiding in identifying patterns over time.
- **Amount**: Represents the transaction amount, which can correlate with fraud tendencies.
- **Class Label**: The target variable, with **1 for fraud** and **0 for non-fraud** transactions.

By leveraging this data, the project seeks to build a high-performing fraud detection model that can tackle the challenges of imbalanced data and safeguard financial transactions.

The `Credit Card Fraud Detection` Analysis is divided into 7 parts

 An example image

Importing Libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import pandas as pd

import warnings
warnings.filterwarnings('ignore')
from imblearn.over_sampling import SMOTE
import time


from sklearn import metrics
from sklearn.metrics import precision_recall_curve,accuracy_score,confusion_matrix,f1_score,roc_auc_score,roc_curve
from sklearn import preprocessing
pd.set_option('display.max_columns',50)
from collections import Counter
```

```
from imblearn.over_sampling import ADASYN
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler,MinMaxScaler
```

Step 1: Reading and Understanding the Data

- **Checking Dataset Shape**: Understand the number of rows and columns to gauge dataset size.
- **Inspecting Dataset Info**: View data types, column names, and non-null counts to gain a basic understanding of the dataset structure.

Step 2: Checking for Missing Values

- **Identifying Missing Values**: Scan the dataset for any missing entries to determine if imputation or other handling is necessary.
- **Handling Missing Data**: If missing values are found, apply appropriate techniques (e.g., imputation, removal) based on data type and distribution.

In [2]:
```
data=pd.read_csv('creditcard.csv')
data.head()
```

Out[2]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.31
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.14
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.16
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.28
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.11

In [3]:
```
# check the shape of the data
data.shape
```

Out[3]:
```
(284807, 31)
```

In [4]:
```
# checking the info of data
data.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
```

- **Total Entries**: 284,807
- **Features**: 31 columns (30 float64 features and 1 int64 target label)
- **Data Completeness**: No missing values detected

In [5]:
```
data.describe()
```

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.84807
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.213481e-16	-2.40633
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.09863
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.34340
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.43097
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.14287
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.97139
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.55949

Checking The `Correlation`

In [6]:
```
corr=data.corr()
corr
```

Out[6]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
Time	1.000000	1.173963e-01	-1.059333e-02	-4.196182e-01	-1.052602e-01	1.730721e-01	-6.301647e-02	8.471437e-02	-3.694943e-02	-8.6604
V1	0.117396	1.000000e+00	4.135835e-16	-1.227819e-15	-9.215150e-16	1.812612e-17	-6.506567e-16	-1.005191e-15	-2.433822e-16	-1.5136
V2	-0.010593	4.135835e-16	1.000000e+00	3.243764e-16	-1.121065e-15	5.157519e-16	2.787346e-16	2.055934e-16	-5.377041e-17	1.978488
V3	-0.419618	-1.227819e-15	3.243764e-16	1.000000e+00	4.711293e-16	-6.539009e-17	1.627627e-15	4.895305e-16	-1.268779e-15	5.568367
V4	-0.105260	-9.215150e-16	-1.121065e-15	4.711293e-16	1.000000e+00	-1.719944e-15	-7.491959e-16	-4.104503e-16	5.697192e-16	6.923247
V5	0.173072	1.812612e-17	5.157519e-16	-6.539009e-17	-1.719944e-15	1.000000e+00	2.408382e-16	2.715541e-16	7.437229e-16	7.391702
V6	-0.063016	-6.506567e-16	2.787346e-16	1.627627e-15	-7.491959e-16	2.408382e-16	1.000000e+00	1.191668e-16	-1.104219e-16	4.131207
V7	0.084714	-1.005191e-15	2.055934e-16	4.895305e-16	-4.104503e-16	2.715541e-16	1.191668e-16	1.000000e+00	3.344412e-16	1.122501
V8	-0.036949	-2.433822e-16	-5.377041e-17	-1.268779e-15	5.697192e-16	7.437229e-16	-1.104219e-16	3.344412e-16	1.000000e+00	4.356078
V9	-0.008660	-1.513678e-16	1.978488e-17	5.568367e-16	6.923247e-16	7.391702e-16	4.131207e-16	1.122501e-15	4.356078e-16	1.000000e
V10	0.030617	7.388135e-17	-3.991394e-16	1.156587e-15	2.232685e-16	-5.202306e-16	5.932243e-17	-7.492834e-17	-2.801370e-16	-4.6422
V11	-0.247689	2.125498e-16	1.975426e-16	1.576830e-15	3.459380e-16	7.203963e-16	1.980503e-15	1.425248e-16	2.487043e-16	1.354680
V12	0.124348	2.053457e-16	-9.568710e-17	6.310231e-16	-5.625518e-16	7.412552e-16	2.375468e-16	-3.536655e-18	1.839891e-16	-1.0793
V13	-0.065902	-2.425603e-17	6.295388e-16	2.807652e-16	1.303306e-16	5.886991e-16	-1.211182e-16	1.266462e-17	-2.921856e-16	2.251072
V14	-0.098757	-5.020280e-16	-1.730566e-16	4.739859e-16	2.282280e-16	6.565143e-16	2.621312e-16	2.607772e-16	-8.599156e-16	3.784757

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
V15	**-0.183453**	**3.547782e-16**	**-4.995814e-17**	**9.068793e-16**	**1.377649e-16**	**-8.720275e-16**	**-1.531188e-15**	**-1.690540e-16**	**4.127777e-16**	**-1.0511**
V16	**0.011903**	**7.212815e-17**	**1.177316e-17**	**8.299445e-16**	**-9.614528e-16**	**2.246261e-15**	**2.623672e-18**	**5.869302e-17**	**-5.254741e-16**	**-1.2140**
V17	**-0.073297**	**-3.879840e-16**	**-2.685296e-16**	**7.614712e-16**	**-2.699612e-16**	**1.281914e-16**	**2.015618e-16**	**2.177192e-16**	**-2.269549e-16**	**1.113695**
V18	**0.090438**	**3.230206e-17**	**3.284605e-16**	**1.509897e-16**	**-5.103644e-16**	**5.308590e-16**	**1.223814e-16**	**7.604126e-17**	**-3.667974e-16**	**4.993240**
V19	**0.028975**	**1.502024e-16**	**-7.118719e-18**	**3.463522e-16**	**-3.980557e-16**	**-1.450421e-16**	**-1.865597e-16**	**-1.881008e-16**	**-3.875186e-16**	**-1.3761**
V20	**-0.050866**	**4.654551e-16**	**2.506675e-16**	**-9.316409e-16**	**-1.857247e-16**	**-3.554057e-16**	**-1.858755e-16**	**9.379684e-16**	**2.033737e-16**	**-2.3437**
V21	**0.044736**	**-2.457409e-16**	**-8.480447e-17**	**5.706192e-17**	**-1.949553e-16**	**-3.920976e-16**	**5.833316e-17**	**-2.027779e-16**	**3.892798e-16**	**1.936953**
V22	**0.144059**	**-4.290944e-16**	**1.526333e-16**	**-1.133902e-15**	**-6.276051e-17**	**1.253751e-16**	**-4.705235e-19**	**-8.898922e-16**	**2.026927e-16**	**-7.0718**
V23	**0.051142**	**6.168652e-16**	**1.634231e-16**	**-4.983035e-16**	**9.164206e-17**	**-8.428683e-18**	**1.046712e-16**	**-4.387401e-16**	**6.377260e-17**	**-5.2141**
V24	**-0.016182**	**-4.425156e-17**	**1.247925e-17**	**2.686834e-19**	**1.584638e-16**	**-1.149255e-15**	**-1.071589e-15**	**7.434913e-18**	**-1.047097e-16**	**-1.4303**
V25	**-0.233083**	**-9.605737e-16**	**-4.478846e-16**	**-1.104734e-15**	**6.070716e-16**	**4.808532e-16**	**4.562861e-16**	**-3.094082e-16**	**-4.653279e-16**	**6.757763**
V26	**-0.041407**	**-1.581290e-17**	**2.057310e-16**	**-1.238062e-16**	**-4.247268e-16**	**4.319541e-16**	**-1.357067e-16**	**-9.657637e-16**	**-1.727276e-16**	**-7.8888**
V27	**-0.005135**	**1.198124e-16**	**-4.966953e-16**	**1.045747e-15**	**3.977061e-17**	**6.590482e-16**	**-4.452461e-16**	**-1.782106e-15**	**1.299943e-16**	**-6.7096**
V28	**-0.009413**	**2.083082e-15**	**-5.093836e-16**	**9.775546e-16**	**-2.761403e-18**	**-5.613951e-18**	**2.594754e-16**	**-2.776530e-16**	**-6.200930e-16**	**1.110541**
Amount	**-0.010596**	**-2.277087e-01**	**-5.314089e-01**	**-2.108805e-01**	**9.873167e-02**	**-3.863563e-01**	**2.159812e-01**	**3.973113e-01**	**-1.030791e-01**	**-4.4245**

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
Class	-0.012323	-1.013473e-01	9.128865e-02	-1.929608e-01	1.334475e-01	-9.497430e-02	-4.364316e-02	-1.872566e-01	1.987512e-02	-9.7732

Checking The Percentage Of `Fraud`

In [7]:
```
(data['Class'].value_counts())/len(data['Class'])*100
```

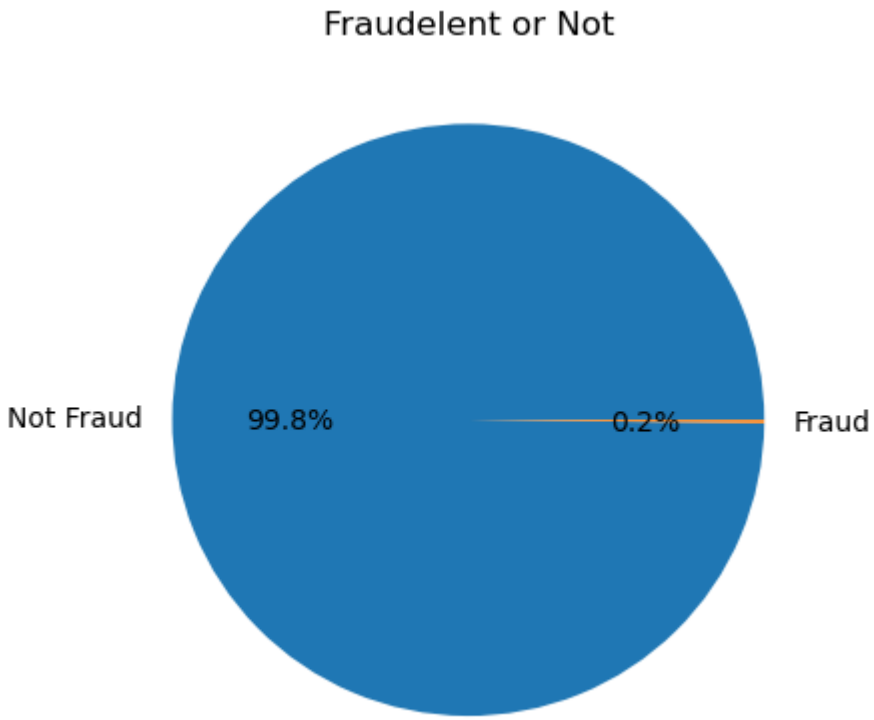
Out[7]:
```
Class
0    99.827251
1     0.172749
Name: count, dtype: float64
```

Out of a total of `284,807` credit card transactions, only `492` are fraudulent. This creates a highly imbalanced dataset, where the majority class consists of genuine transactions. As a result, a model could easily achieve high accuracy by simply predicting the majority class, without effectively identifying fraudulent transactions.

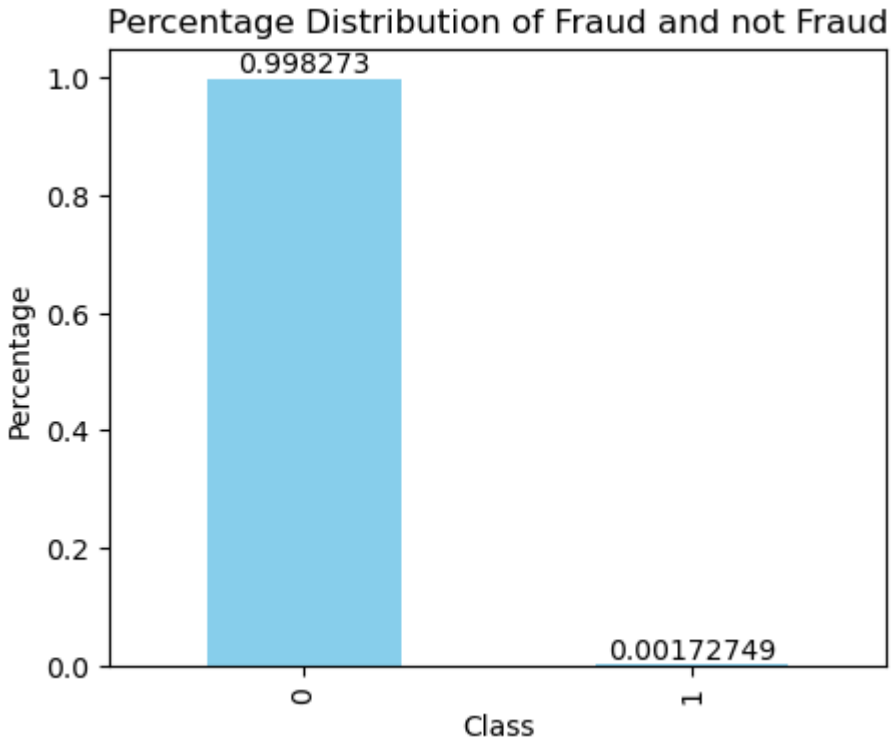
To overcome this limitation, we will use alternative evaluation metrics such as ROC-AUC, precision, and recall. These metrics will give us a more meaningful understanding of the model's performance, particularly in detecting fraud despite the class imbalance.

Step 3:Data Visualisation</hb>

In [8]:
```
plt.title("Fraudelent or Not")
plt.pie(data['Class'].value_counts(),labels=['Not Fraud','Fraud'],autopct='%1.1f%%')
plt.show()
```

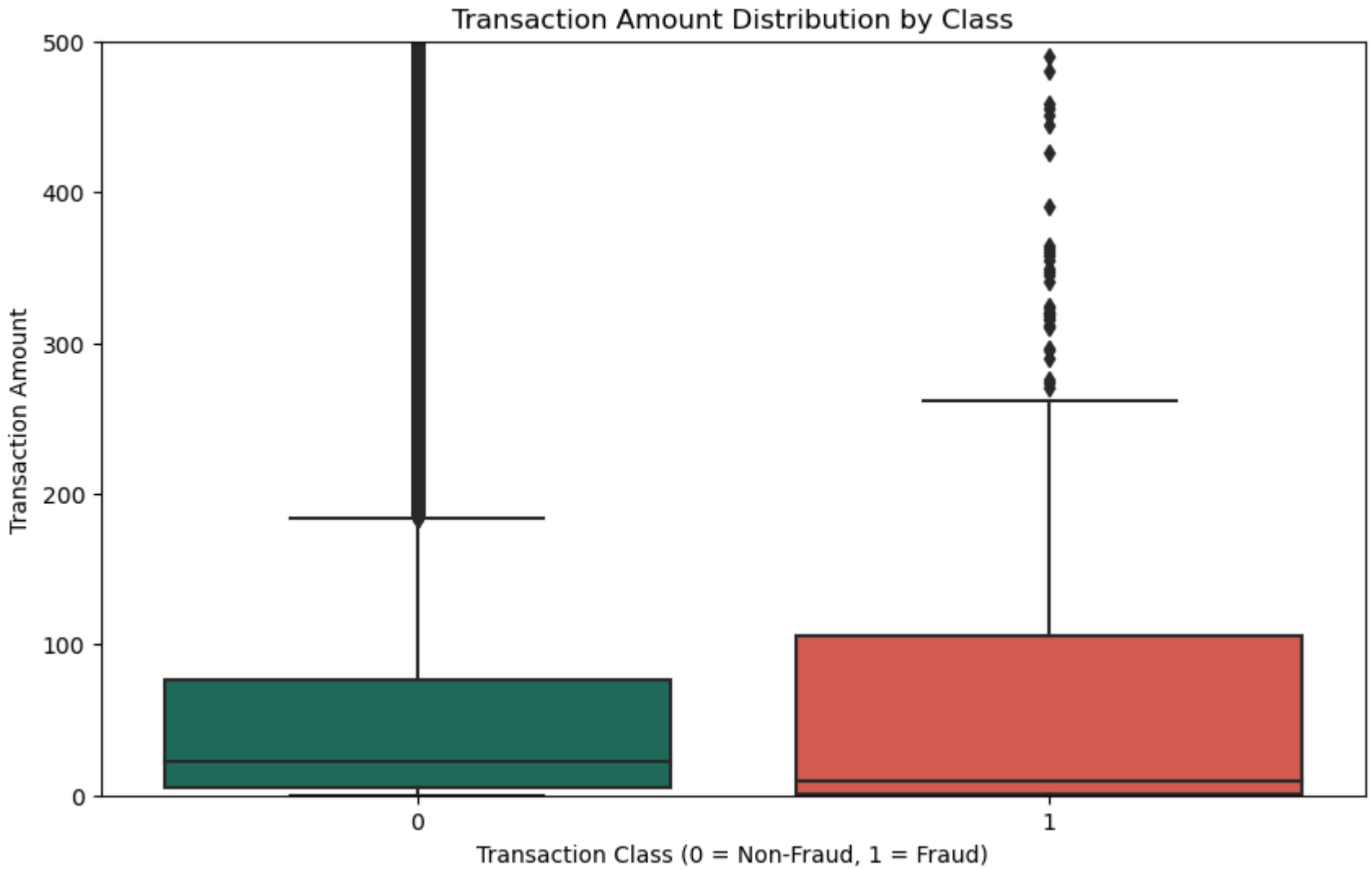


In [9]:
```
plt.figure(figsize=(5,4))
ax=data['Class'].value_counts(normalize = True).plot.bar(color=['skyblue','skyblue'])
ax.bar_label(ax.containers[0], label_type='edge')
plt.title('Percentage Distribution of Fraud and not Fraud')
plt.xlabel('Class')
plt.ylabel('Percentage')
plt.show()
```



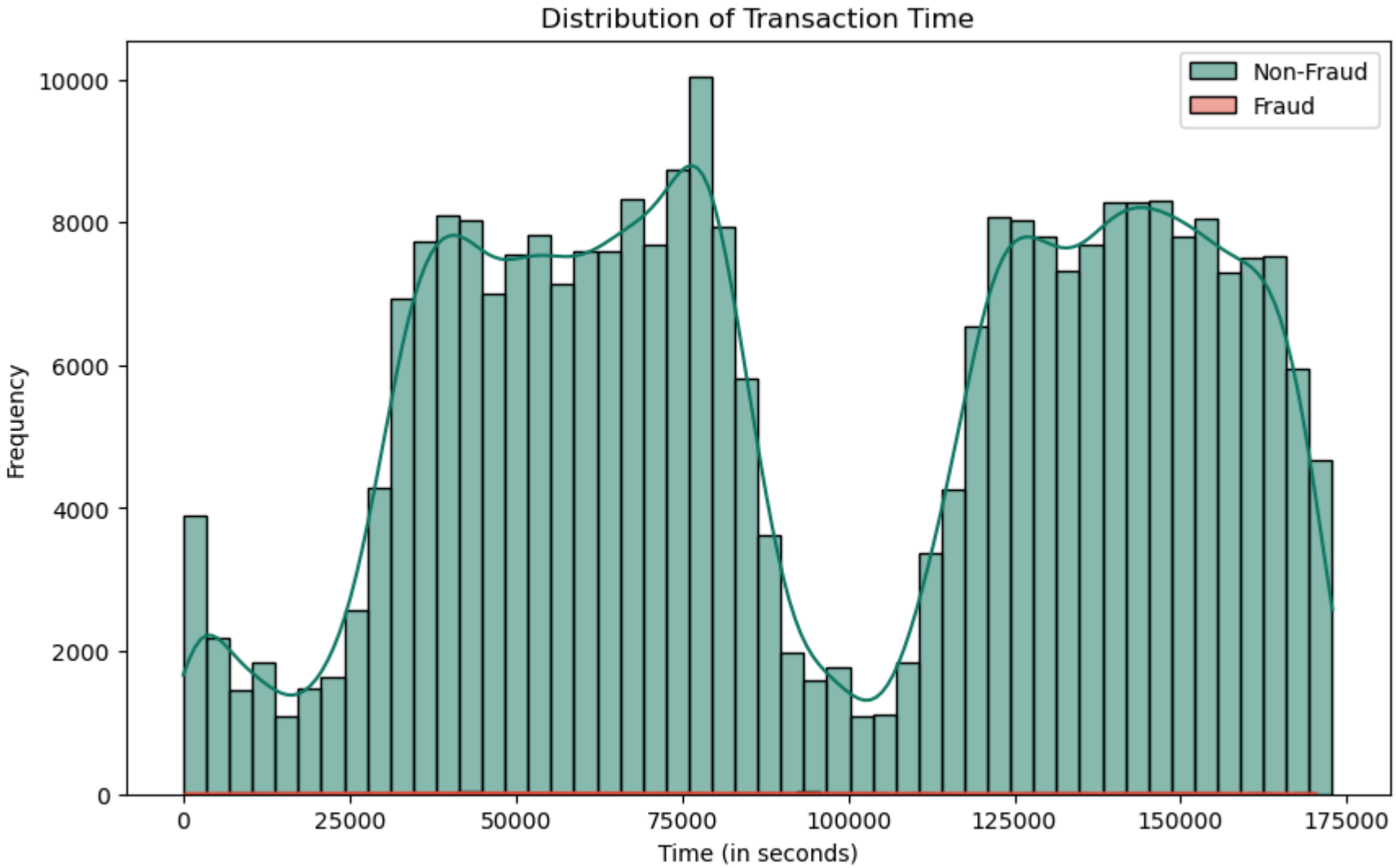
Note: Observe the strong imbalance in our dataset: the majority of transactions are non-fraudulent. Using this dataset without adjustments could lead to inaccurate predictions, as models might overfit by assuming most transactions are genuine. Our goal is not for the model to "assume" but to identify patterns that indicate fraud!

In [10]:
```
plt.figure(figsize=(10, 6))

# Box plot for transaction amount by class
sns.boxplot(x='Class', y='Amount', data=data, palette=['#117A65', '#E74C3C'])
plt.title('Transaction Amount Distribution by Class')
plt.xlabel('Transaction Class (0 = Non-Fraud, 1 = Fraud)')
plt.ylabel('Transaction Amount')
plt.ylim(0, 500)
plt.show()
```



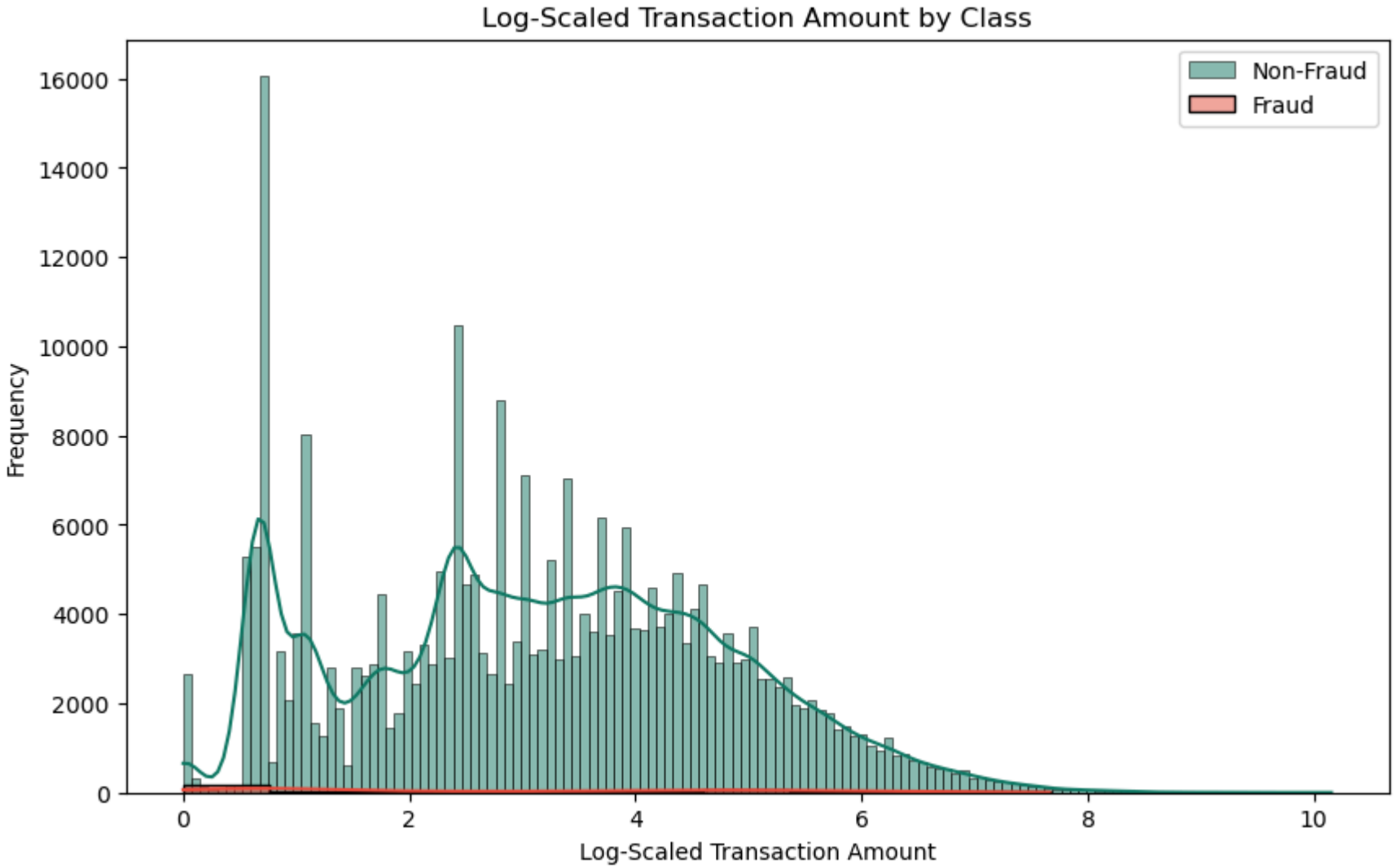
In [11]:
```
plt.figure(figsize=(10, 6))

# Plot distribution of transaction time by class
sns.histplot(data[data['Class'] == 0]['Time'], color='#117A65', label='Non-Fraud', kde=True, bins=50)
sns.histplot(data[data['Class'] == 1]['Time'], color='#E74C3C', label='Fraud', kde=True, bins=50)
plt.title('Distribution of Transaction Time')
plt.xlabel('Time (in seconds)')
plt.ylabel('Frequency')
```

```
plt.legend()
plt.show()
```



In [12]:
```
import numpy as np

plt.figure(figsize=(10, 6))
sns.histplot(np.log1p(data[data['Class'] == 0]['Amount']), color='#117A65', label='Non-Fraud', kde=True)
sns.histplot(np.log1p(data[data['Class'] == 1]['Amount']), color='#E74C3C', label='Fraud', kde=True)
plt.title('Log-Scaled Transaction Amount by Class')
```

```
plt.xlabel('Log-Scaled Transaction Amount')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

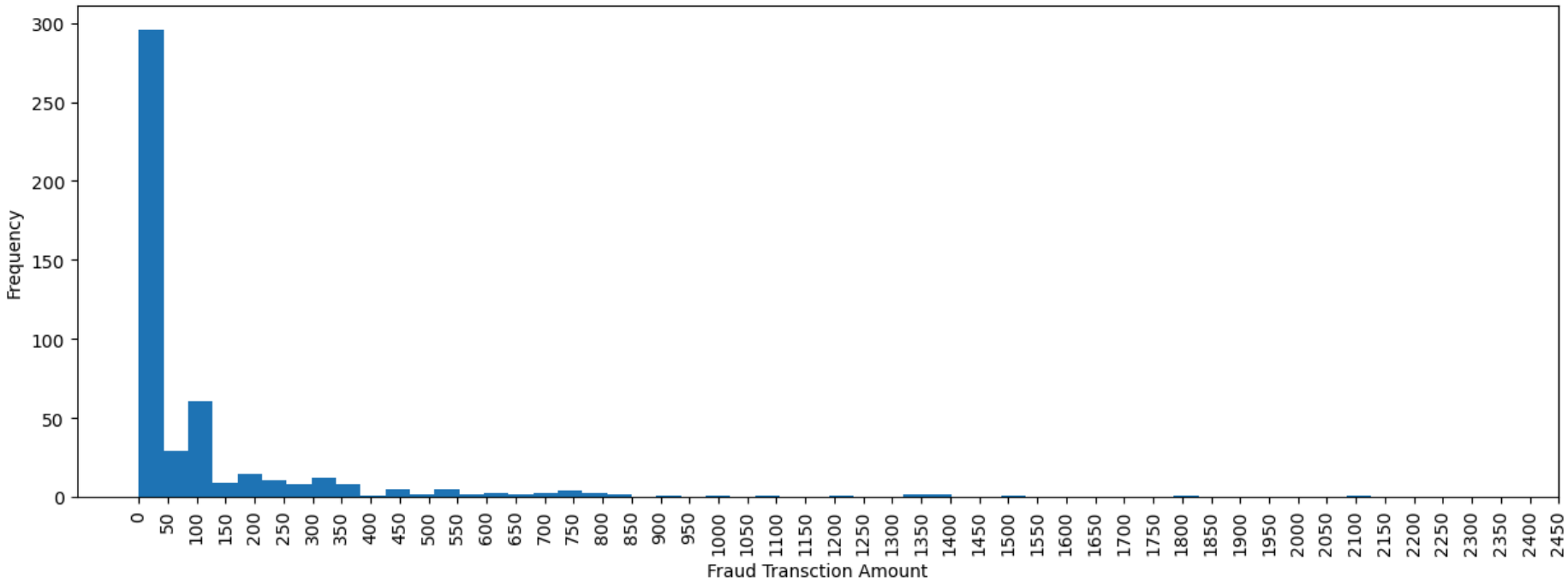


In [13]:
```
fraud=data[data['Class']==1]
no_fraud=data[data['Class']==0]
```

In [14]:
```
fraud.shape,no_fraud.shape
```

Out[14]:
```
((492, 31), (284315, 31))
```

In [15]:
```
plt.figure(figsize=(15,5))
plt.hist(fraud['Amount'],bins=50)
plt.xticks(range(0, 2500, 50), rotation=90)
plt.xlabel('Fraud Transction Amount')
plt.ylabel('Frequency')
plt.show()
```



Key Insights from the Plot Analysis

Observing the plot reveals interesting trends in fraud incidents based on transaction amounts:

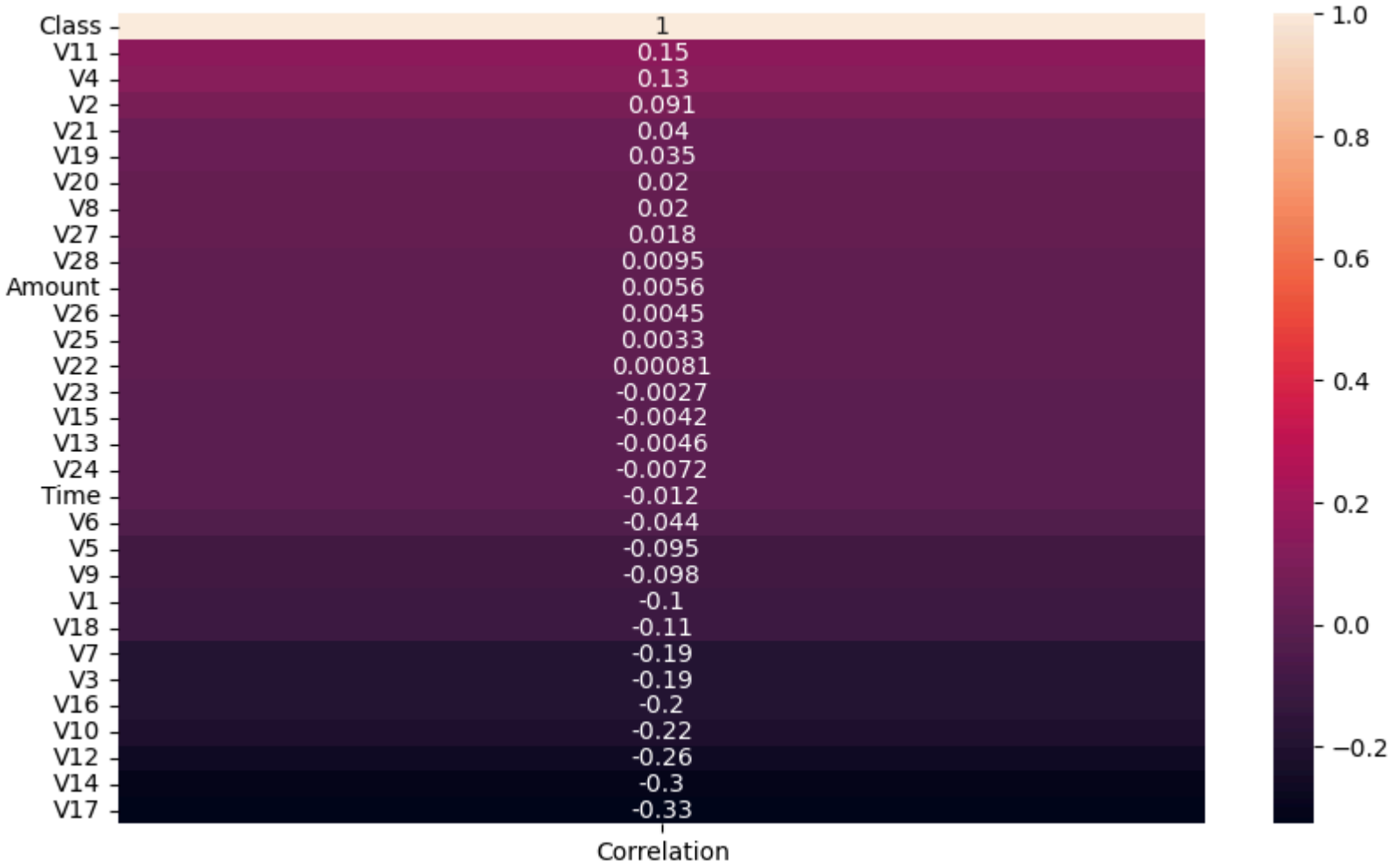
- **High Frequency of Fraud for Smaller Amounts: Transactions below 50 Rupees show the highest fraud occurrence.**
- **Moderate Fraud Cases for Mid-Range Amounts: A significant number of fraud incidents also occur within the 100 to 350 Rupees range.**

- **Rare Frauds for Large Amounts: Fraud incidents involving very large amounts are rare but still present.**

These insights provide guidance for developing targeted strategies in fraud detection across varying transaction sizes.

In [16]:
```
data_corr=pd.DataFrame(data.corr()['Class'].sort_values(ascending=False)).rename(columns={'Class':'Correlation'})
```

In [17]:
```
plt.figure(figsize=(10,6))
sns.heatmap(data=data_corr,annot=True)
plt.show()
```



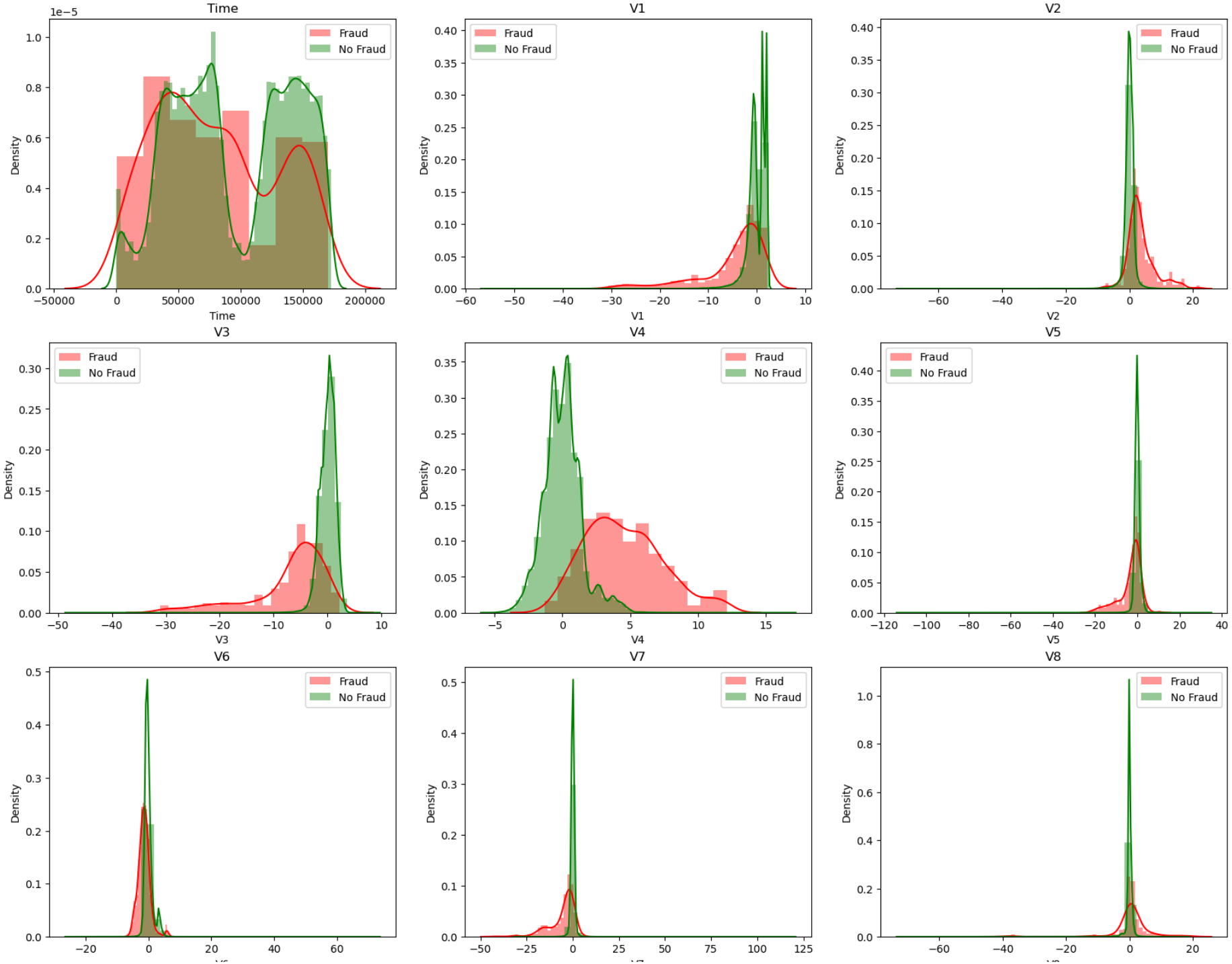
In [18]:
```
cols=list(data.columns.values)
```

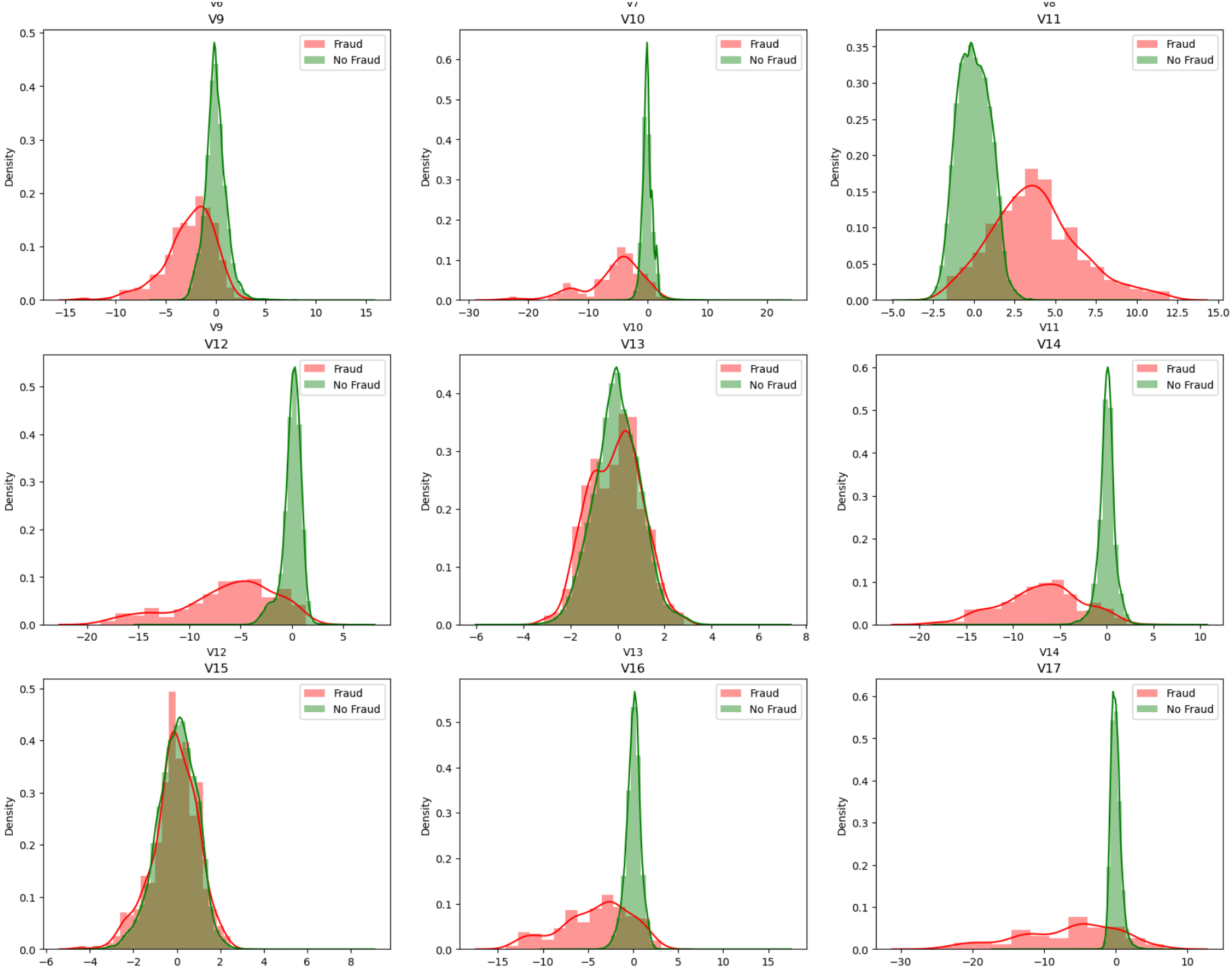
In [19]:
```
print(cols)
```

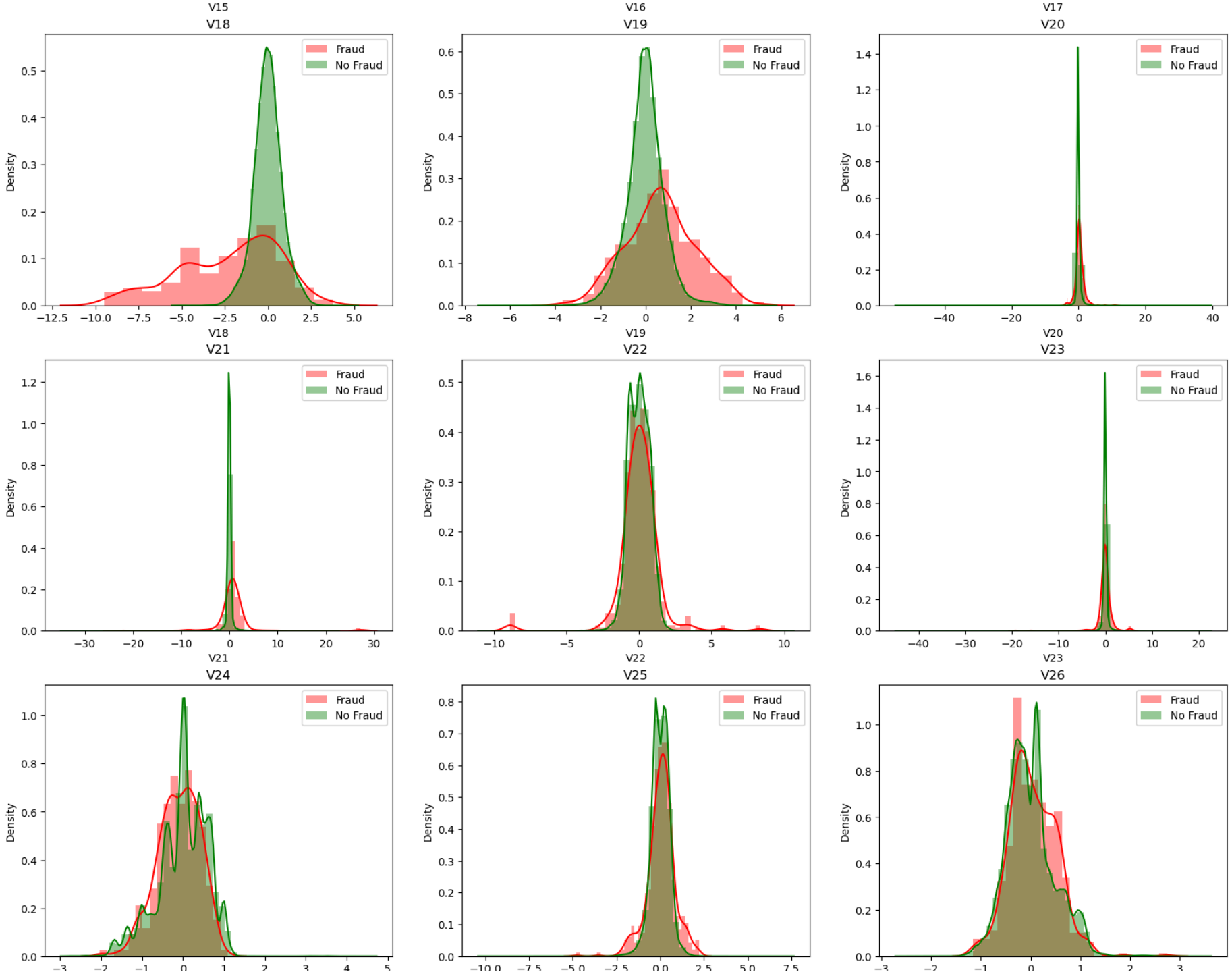
```
['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V
19', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount', 'Class']
```

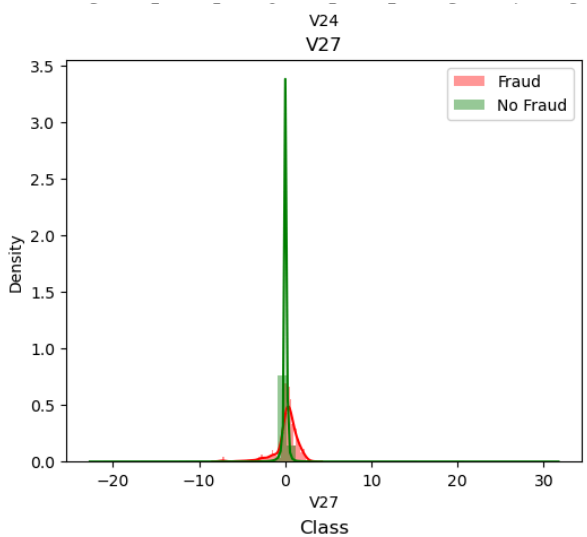
Plotting All The Variables And Seeing Wheather They Are Fraud or Not Fraud

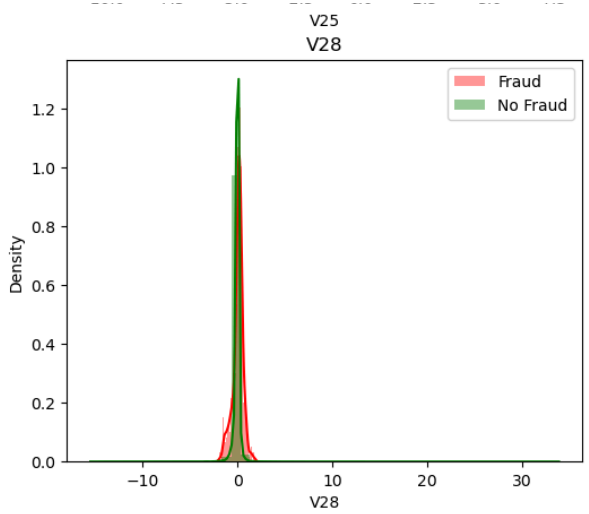
In [20]:
```
plt.figure(figsize=(20,60))

for i,j in enumerate(cols):
    plt.subplot(11,3,i+1)
    sns.distplot(data[j][data.Class==1],color='red',label='Fraud')
    sns.distplot(data[j][data.Class==0],color='green',label='No Fraud')
    plt.legend()
    plt.title(j)
plt.show()
```

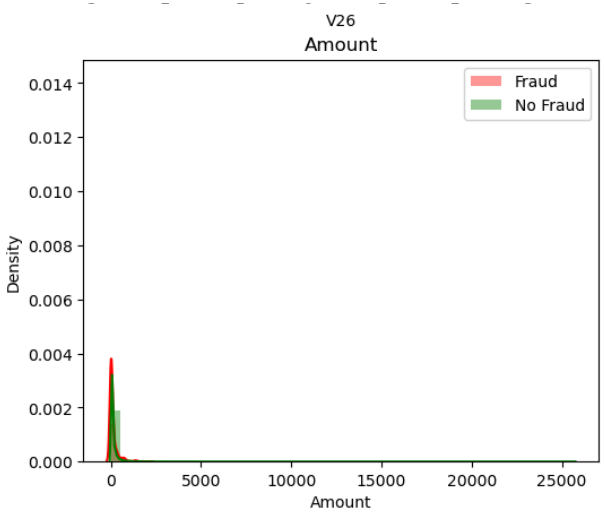

Time
1e−5
Fraud
No Fraud
Density
Time
V1
V2
V3
V4
V5
V6
V7
V8

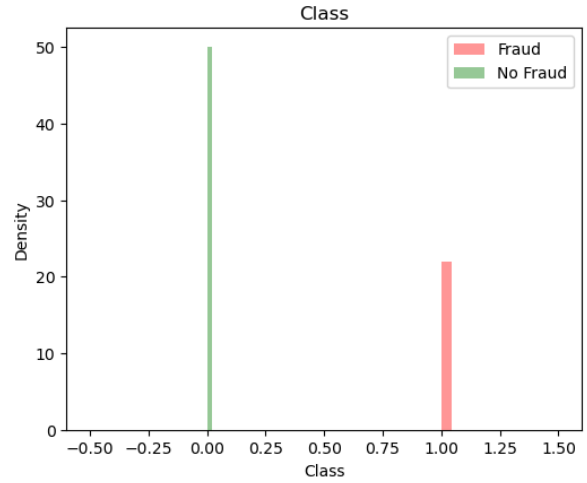

V9
V10
V11
V12
V13
V14
V15
V16
V17
Fraud
No Fraud
Density


V15
V18
V16
V19
V17
V20
V18
V21
V19
V22
V20
V23
V21
V24
V22
V25
V23
V26
Fraud
No Fraud
Density









Step 4: Model selection and Train-Test split

In [21]:
```
from sklearn.model_selection import train_test_split
```

In [22]:
```
X=data.drop(['Class'],axis=1)
```

In [23]:
```
y=data['Class']
```

In [24]:
```
X.shape
```

Out[24]: (284807, 30)

In [25]:
```
y.shape
```

Out[25]: (284807,)

In [26]:
```
X.head(2)
```

Out[26]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095

In [27]:
```
y.head()
```

Out[27]:
```
0    0
1    0
2    0
3    0
4    0
Name: Class, dtype: int64
```

Splitting The Data Into `Train_Test_split`

In [28]:
```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,stratify=y,random_state=21)
```

In [29]:
```
print('The Shape Of The X_train Is',X_train.shape)
print('The Shape Of The X_test Is',X_test.shape)
print('The Shape Of The y_train Is',y_train.shape)
print('The Shape Of The y_test Is',y_test.shape)
```

```
The Shape Of The X_train Is (227845, 30)
The Shape Of The X_test Is (56962, 30)
The Shape Of The y_train Is (227845,)
The Shape Of The y_test Is (56962,)
```

Feature Scaling

We only need to scale the `Amount` , `Time` columns since the other columns are already scaled using PCA.

Feature scaling is important because it ensures that all features are on a similar scale, preventing models from being biased toward features with larger numerical ranges. It improves the convergence speed and accuracy of algorithms

In [30]:
```
scaler=StandardScaler()
```

Scaling The Train Data

In [31]:
```
# before scaling
X_train.head()
```

Out[31]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
1865	1441.0	-0.568089	0.223634	2.784809	1.398554	-0.976417	1.181462	-0.350213	0.544987	0.184840	-0.298910	1.063224	0.987810
47141	43099.0	1.005389	-0.382836	1.280514	1.682977	-0.950041	0.752944	-0.764272	0.535072	1.297392	-0.189560	0.413669	0.613984
155271	104967.0	-3.633303	-1.481175	0.094030	-0.287921	-0.630724	0.531253	1.667692	-0.881678	3.188153	0.232909	0.108605	-2.586208
250359	154865.0	-0.111852	0.255704	1.589792	-0.007662	-0.400043	0.183738	-0.123587	-0.053031	1.720502	-1.091077	-1.173797	0.899706
234143	147846.0	-3.063601	2.554788	-0.294903	-1.092867	-0.770817	-0.987801	-0.270569	0.883005	1.513483	0.350049	-0.754589	0.019737

In [32]:
```
X_train[['Time','Amount']]=scaler.fit_transform(X_train[['Time','Amount']])
```

In [33]:
```
# after scaling
X_train.head(4)
```

Out[33]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
1865	-1.966629	-0.568089	0.223634	2.784809	1.398554	-0.976417	1.181462	-0.350213	0.544987	0.184840	-0.298910	1.063224	0.987810
47141	-1.089159	1.005389	-0.382836	1.280514	1.682977	-0.950041	0.752944	-0.764272	0.535072	1.297392	-0.189560	0.413669	0.613984
155271	0.214008	-3.633303	-1.481175	0.094030	-0.287921	-0.630724	0.531253	1.667692	-0.881678	3.188153	0.232909	0.108605	-2.586208
250359	1.265042	-0.111852	0.255704	1.589792	-0.007662	-0.400043	0.183738	-0.123587	-0.053031	1.720502	-1.091077	-1.173797	0.899706

Scaling The Test Data

In [34]:
```
# before scaling
X_test.head(4)
```

Out[34]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
234721	148082.0	1.946305	-0.511462	-0.458261	0.395096	-0.300224	0.525437	-0.690892	0.264120	1.489920	-0.175138	0.080315	1.299188
151554	95669.0	-7.290632	5.392193	-2.101657	-1.565481	-0.693676	-0.338955	0.454925	0.384947	7.012826	7.421964	-0.164766	-2.023180
46794	42928.0	1.134607	0.130074	0.380272	1.409226	-0.197326	-0.183836	0.062061	0.003161	0.348476	-0.120379	-0.603154	0.321578
163972	116356.0	1.798070	-0.885743	-1.175621	0.143376	-0.160843	0.160874	-0.273504	-0.058693	1.440568	-0.319191	-1.712946	0.483158

In [35]:
```
X_test[['Time','Amount']]=scaler.fit_transform(X_test[['Time','Amount']])
```

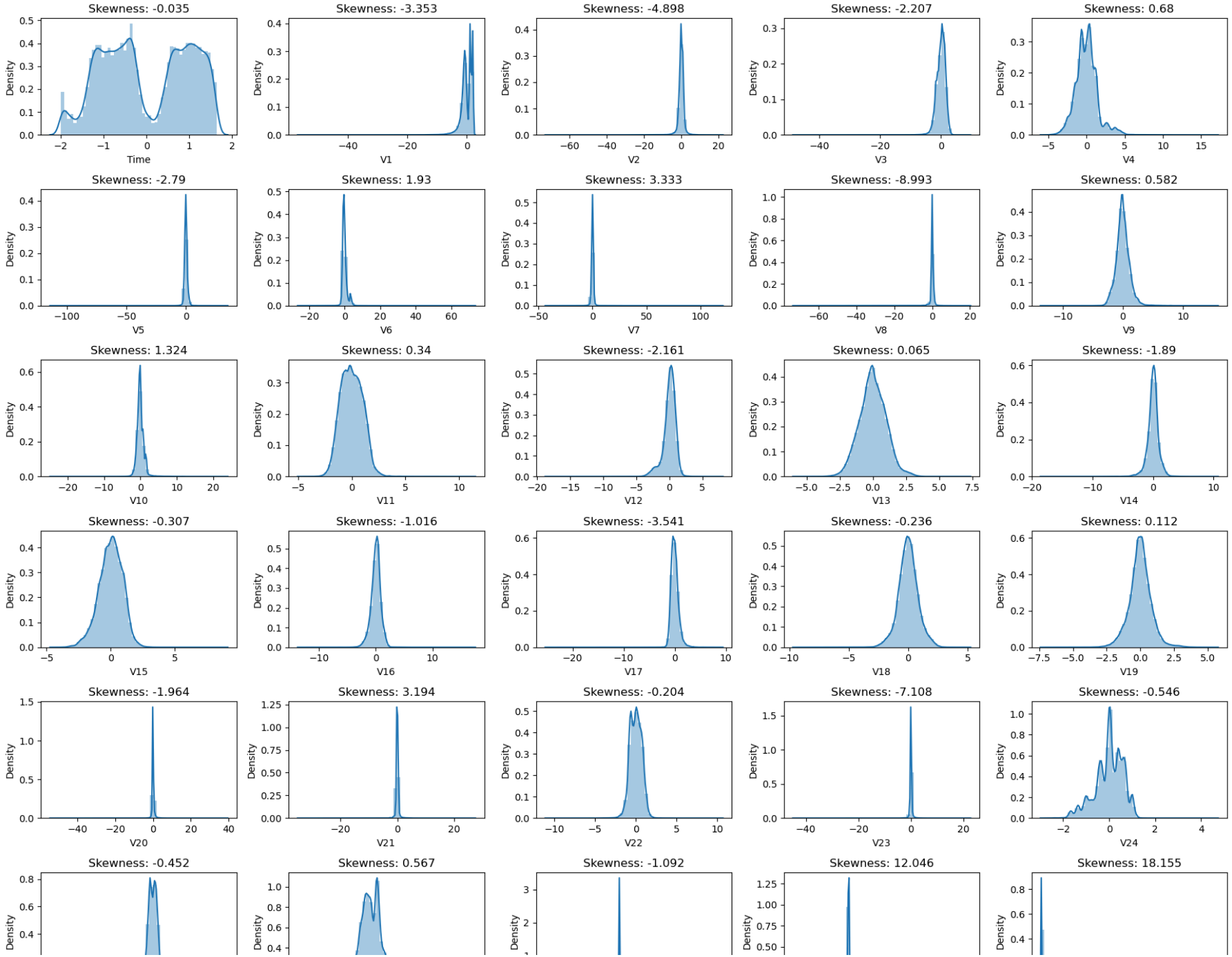
In [36]:
```
# after scaling
X_test.head()
```

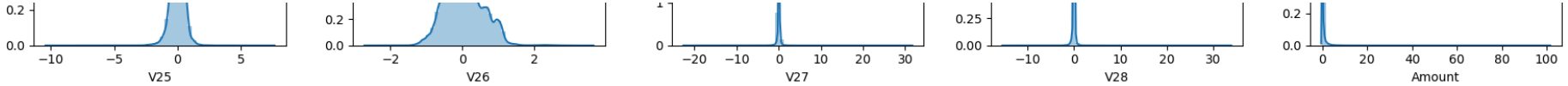
Out[36]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V1
234721	1.119915	1.946305	-0.511462	-0.458261	0.395096	-0.300224	0.525437	-0.690892	0.264120	1.489920	-0.175138	0.080315	1.29918
151554	0.017407	-7.290632	5.392193	-2.101657	-1.565481	-0.693676	-0.338955	0.454925	0.384947	7.012826	7.421964	-0.164766	-2.02318
46794	-1.092001	1.134607	0.130074	0.380272	1.409226	-0.197326	-0.183836	0.062061	0.003161	0.348476	-0.120379	-0.603154	0.3215
163972	0.452558	1.798070	-0.885743	-1.175621	0.143376	-0.160843	0.160874	-0.273504	-0.058693	1.440568	-0.319191	-1.712946	0.4831
211004	0.912405	-1.092804	-0.140712	0.608619	-2.621535	0.095963	0.487388	-0.482433	0.700699	-1.178947	-0.196379	-0.269245	0.0043

In [37]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(18, 15))
k = 0
num_plots = len(cols) - 1  # Adjust number of subplots to fit the number of features

for i in cols[:-1]:
```

```
    k += 1
    if k > num_plots:  # Prevent excess subplot creation
        break
    plt.subplot(6, 5, k)
    sns.distplot(X_train[i], kde=True)  # `kde=True` for smoothness in distribution plot
    plt.tight_layout()
    plt.title(f'Skewness: {round(X_train[i].skew(), 3)}')

plt.show()
```


Skewness: -0.035
Time
Skewness: -3.353
V1
Skewness: -4.898
V2
Skewness: -2.207
V3
Skewness: 0.68
V4
Skewness: -2.79
V5
Skewness: 1.93
V6
Skewness: 3.333
V7
Skewness: -8.993
V8
Skewness: 0.582
V9
Skewness: 1.324
V10
Skewness: 0.34
V11
Skewness: -2.161
V12
Skewness: 0.065
V13
Skewness: -1.89
V14
Skewness: -0.307
V15
Skewness: -1.016
V16
Skewness: -3.541
V17
Skewness: -0.236
V18
Skewness: 0.112
V19
Skewness: -1.964
V20
Skewness: 3.194
V21
Skewness: -0.204
V22
Skewness: -7.108
V23
Skewness: -0.546
V24
Skewness: -0.452
Skewness: 0.567
Skewness: -1.092
Skewness: 12.046
Skewness: 18.155
Density



Handling Skewness in Data

> **Observation: The dataset displays significant skewness, which may introduce biases in model training. Many features are heavily skewed, so reducing skewness is crucial for achieving a more balanced distribution.**

Effective Methods to Address Skewness:

1. **Log Transformation - Compresses the data range by applying a logarithmic function.**
2. **Square Root Transformation - Reduces high values in right-skewed data.**
3. **Cube Root Transformation - A moderate transformation for handling skewness.**
4. **Box-Cox Transformation - Approximates a normal distribution; best for positive values.**
5. **Yeo-Johnson Transformation - Extends Box-Cox to support negative values.**
6. **Power Transformation - Applies power functions to balance data and normalize distributions.**
7. **Winsorizing - Limits extreme values, reducing the impact of outliers.**
8. **Binning/Discretization - Groups data into intervals for balance.**
9. **Square or Cube Transformation - Adjusts for both right and left skewness.**

For this dataset, we use **Power Transformation as it helps to stabilize variance, making the data more suitable for machine learning models and enhancing accuracy and reliability.**

In [38]:
```
# import power transformation
from sklearn.preprocessing import PowerTransformer
```

In [39]:
```
pt=PowerTransformer(method='yeo-johnson',standardize=True,copy=False)
```

In [40]:
```
columns=X_train.columns
```

In [41]:
```
X_train=pt.fit_transform(X_train)
```

In [42]:
```
X_train=pd.DataFrame(X_train)
X_train.columns=columns
```

```
X_train.head(3)
```

Out[42]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	
0	-1.956747	-0.560373	0.061289	2.445683	0.997344	-0.710176	0.936322	-0.280283	0.493499	0.221513	-0.239960	1.040326	1.169566	0.11
1	-1.088335	0.468098	-0.326726	0.905905	1.170356	-0.691455	0.655243	-0.614782	0.482406	1.161519	-0.134062	0.443925	0.602914	-2.56
2	0.210075	-1.653454	-0.946734	-0.087467	-0.133325	-0.464170	0.502381	1.340689	-0.862715	2.610628	0.261050	0.152085	-2.106569	1.92

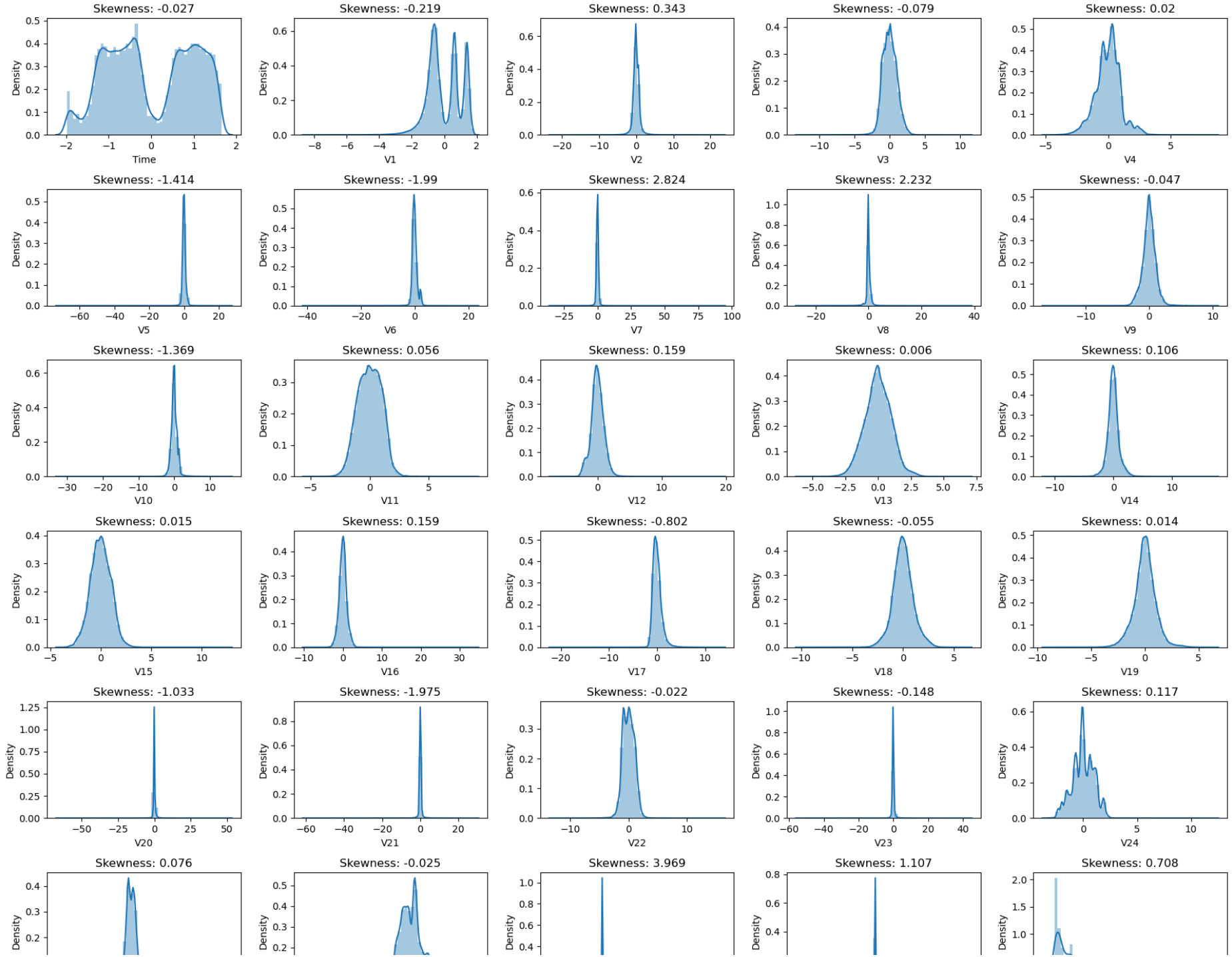
In [43]:
```
X_test=pt.transform(X_test)
```

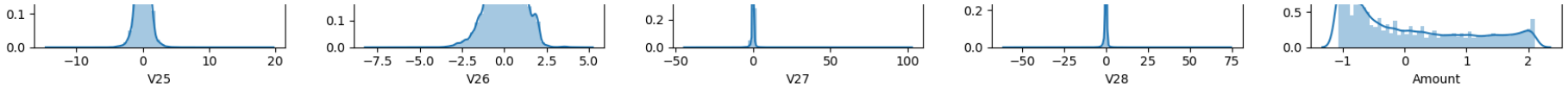
In [44]:
```
X_test=pd.DataFrame(X_test)
X_test.columns=columns
X_test.head()
```

Out[44]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	
0	1.121163	1.336134	-0.403775	-0.456701	0.352977	-0.227467	0.498291	-0.555461	0.187938	1.315829	-0.120199	0.124530	1.687890	-0
1	0.013221	-2.548065	4.483300	-1.341837	-1.160050	-0.509079	-0.172566	0.368137	0.317128	5.268013	5.715079	-0.118246	-1.778214	1
2	-1.091153	0.577595	-0.001636	0.129205	1.003899	-0.153414	-0.041573	0.052112	-0.078240	0.366359	-0.067794	-0.568615	0.205717	-0
3	0.449414	1.188995	-0.620382	-0.870903	0.179783	-0.127112	0.232202	-0.218380	-0.138533	1.276457	-0.259748	-1.775285	0.419960	0
4	0.911988	-0.794646	-0.177250	0.313524	-2.084912	0.058796	0.471423	-0.387034	0.670460	-1.106013	-0.140626	-0.223855	-0.174459	1

In [45]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(18, 15))
num_plots = min(len(cols) - 1, 30)  # Limit to 30 subplots to fit 6x5 grid

for k, i in enumerate(cols[:-1][:num_plots]):  # Limit loop to `num_plots`
    plt.subplot(6, 5, k + 1)  # Using `k + 1` here as `k` starts from 0
    sns.distplot(X_train[i], kde=True)  # `kde=True` for a smoother plot
    plt.tight_layout()
    plt.title(f'Skewness: {round(X_train[i].skew(), 3)}')

plt.show()
```


Skewness: -0.027
Time
Skewness: -0.219
V1
Skewness: 0.343
V2
Skewness: -0.079
V3
Skewness: 0.02
V4
Skewness: -1.414
V5
Skewness: -1.99
V6
Skewness: 2.824
V7
Skewness: 2.232
V8
Skewness: -0.047
V9
Skewness: -1.369
V10
Skewness: 0.056
V11
Skewness: 0.159
V12
Skewness: 0.006
V13
Skewness: 0.106
V14
Skewness: 0.015
V15
Skewness: 0.159
V16
Skewness: -0.802
V17
Skewness: -0.055
V18
Skewness: 0.014
V19
Skewness: -1.033
V20
Skewness: -1.975
V21
Skewness: -0.022
V22
Skewness: -0.148
V23
Skewness: 0.117
V24
Skewness: 0.076
Skewness: -0.025
Skewness: 3.969
Skewness: 1.107
Skewness: 0.708
Density



Skewness Adjustment: Power transformations are used to reduce or eliminate skewness in data, addressing any asymmetry within the distribution. After transformation, the data becomes more balanced and symmetric, which is ideal for analysis.

Normalization: These transformations also help shape the data to closely resemble a normal distribution, supporting statistical and machine learning techniques that rely on normality assumptions for optimal performance.

****

• Observation: With these transformations applied, we observe that all variables now appear to follow a normal distribution in the plots, enhancing both model accuracy and interpretability.

****

Model Selection and Key Insights:

- **Logistic Regression: Ideal when data is linearly separable, offering clear and interpretable outputs.**
- **K-Nearest Neighbors (KNN): Though interpretable and intuitive, KNN is inefficient with large datasets due to high memory and computation costs.**
- **Decision Trees: Often chosen for their intuitive results, yet they can overfit easily without constraints.**
- **KNN Voting Mechanism: Works best with odd k-values to ensure clear majority decisions from neighboring data points.**
- **Gradient Boosting: Iteratively enhances predictions by minimizing previous errors.**
- **XGBoost: Expands on gradient boosting by incorporating regularization and parallel processing, leading to faster, more accurate outcomes.**

Given our large dataset of **284,807 records**, KNN isn't practical as it requires substantial memory for all data points and is computationally heavy.

Step 5: Model Performance Assessment on Imbalanced Data

---

1. Managing Imbalanced Data:

- For datasets with only **0.17% fraudulent transactions**, selecting metrics that assess **minority class performance** is essential for reliable evaluation.

2. Limitations of Accuracy:

- Relying on **accuracy** alone in imbalanced cases can be misleading, as high accuracy often reflects a bias toward the majority class rather than effective fraud detection.

3. Emphasizing ROC-AUC for Balanced Assessment:

- The **ROC-AUC score** provides a balanced measure by evaluating how well the model can **differentiate between positive and negative classes**.

4. Leveraging the ROC Curve:

- Using the **ROC curve** enables a visual representation of the model's **sensitivity and specificity** across different threshold settings, aiding in robust assessment.

5. Choosing an Optimal Threshold:

- Rather than the typical 0.5 threshold, a custom threshold is selected to **maximize true positives** and minimize **false positives**, thereby enhancing predictive accuracy.

6. Calculating F1 Score for Balanced Performance:

- **After threshold optimization, the **F1 Score** provides a balanced view of **precision and recall**, ensuring a thorough evaluation of model performance on critical classes.**

Logistic Regression

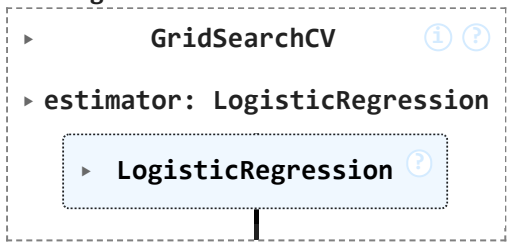
In [46]:
```
# importing libraries
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix,f1_score,classification_report
from sklearn.model_selection import KFold,cross_val_score,GridSearchCV,RandomizedSearchCV
```

In [47]:
```
# creating k fold with 5 splits
folds = KFold(n_splits=5, random_state=4, shuffle=True)

# Specifying score as recall as we are more focused on achieving higher sensitivity
params = {'C': [0.01, 0.1, 1, 10, 100, 1000]}

# Using 'recall' as the scoring metric
model_cv = GridSearchCV(estimator=LogisticRegression(),
                        param_grid=params,
                        scoring='recall',  # Use 'recall' as the scoring metric
                        cv=folds,
                        verbose=1,
                        return_train_score=True)
```

In [48]:
```
# fitting the model
model_cv.fit(X_train,y_train)
```

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

Out[48]:



In [49]:
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[49]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score
0	0.475929	0.072310	0.030397	0.003003	0.01	{'C': 0.01}	0.602740	0.589744	0.556962	0.655172
1	0.457229	0.021534	0.040977	0.004876	0.1	{'C': 0.1}	0.630137	0.602564	0.594937	0.712644
2	0.450775	0.035883	0.040672	0.005358	1	{'C': 1}	0.630137	0.615385	0.632911	0.712644

In [50]:
```
cv_results.columns
```

Out[50]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_C', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'split3_test_score', 'split4_test_score',
       'mean_test_score', 'std_test_score', 'rank_test_score',
       'split0_train_score', 'split1_train_score', 'split2_train_score',
       'split3_train_score', 'split4_train_score', 'mean_train_score',
       'std_train_score'],
      dtype='object')
```

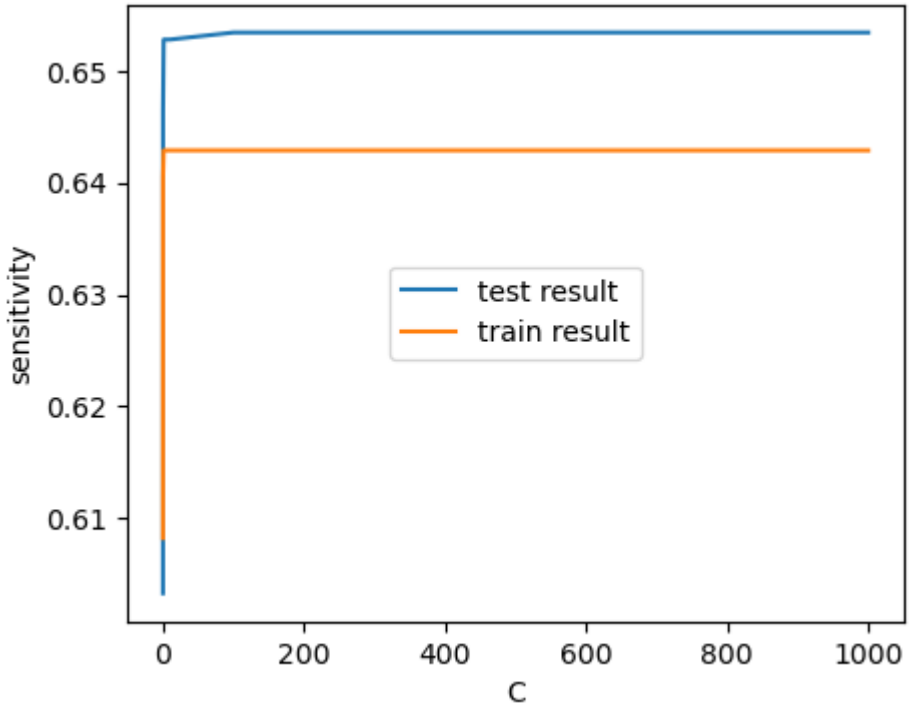
In [51]:
```
cv_results[['param_C','rank_test_score','mean_train_score','mean_test_score']]
```

Out[51]:

	param_C	rank_test_score	mean_train_score	mean_test_score
0	0.01	6	0.603259	0.608196
1	0.1	5	0.646491	0.640524
2	1	1	0.652807	0.642891
3	10	1	0.652807	0.642891
4	100	1	0.653438	0.642891
5	1000	1	0.653438	0.642891

In [52]:
```
plt.figure(figsize=(5,4))
plt.plot(cv_results['param_C'],cv_results['mean_train_score'])
```

```
plt.plot(cv_results['param_C'],cv_results['mean_test_score'])
plt.xlabel('C')
plt.ylabel('sensitivity')
plt.legend(['test result', 'train result'], loc='center')
# plt.xscale('log')
plt.show()
```



In [53]:
```
# best score
best_score=model_cv.best_score_
# best params
best_params=model_cv.best_params_
```

In [54]:
```
print(f"The Best Score Is {best_score}")
print(f"The Best Params Is {best_params}")
```

```
The Best Score Is 0.6428906591257183
The Best Params Is {'C': 1}
```

In [55]:
```
logistic_=LogisticRegression(C=1)
```

In [56]:
```
# fiitting the model on train set
logistic_model1=logistic_.fit(X_train,y_train)
```

In [57]:
```
y_train_pred_logistic=logistic_model1.predict(X_train)
```

In [58]:
```
confusion_matrix_logistic_train=metrics.confusion_matrix(y_train,y_train_pred_logistic)
```

In [59]:
```
confusion_matrix_logistic_train
```

Out[59]:
```
array([[227420,     31],
       [   138,    256]], dtype=int64)
```

In [60]:
```
TN = confusion_matrix_logistic_train[0,0] # True negative
FP = confusion_matrix_logistic_train[0,1] # False positive
FN = confusion_matrix_logistic_train[1,0] # False negative
TP = confusion_matrix_logistic_train[1,1] # True positive
```

In [61]:
```
def calculation_metrics(TN,FP,FN,TP,):
    c=TP / float(TP+FN)
    print('The Sensitivity  is :' ,c)

    d=TN / float(TN+FP)
    print('The Specificity  is :',d)


calculation_metrics(TN,FP,FN,TP)
```

```
The Sensitivity  is : 0.649746192893401
The Specificity  is : 0.9998637069083012
```

In [62]:
```
accuracy=metrics.accuracy_score(y_train,y_train_pred_logistic)
print('The Accuracy of Logistic Regression For Train is :',accuracy)

F1_score=metrics.f1_score(y_train,y_train_pred_logistic)
print("The F1-score of Logistic Regression For Train is :", F1_score)
```

```
The Accuracy of Logistic Regression For Train is : 0.9992582676819768
The F1-score of Logistic Regression For Train is : 0.7518355359765051
```

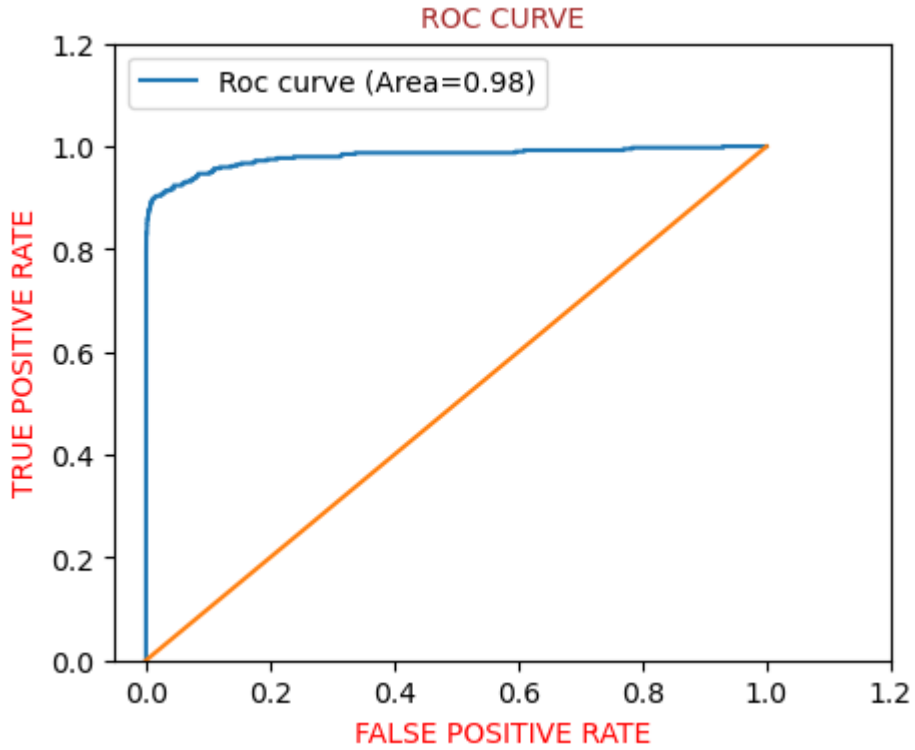
In [63]:
```
# classificatino report
print(classification_report(y_train,y_train_pred_logistic))
```

```
                precision    recall  f1-score   support

           0       1.00      1.00      1.00    227451
           1       0.89      0.65      0.75       394

    accuracy                           1.00    227845
   macro avg       0.95      0.82      0.88    227845
weighted avg       1.00      1.00      1.00    227845
```

In [64]:
```
# predicted probability
y_train_pred_logistic_proba=logistic_model1.predict_proba(X_train)[:,1]
```

In [65]:
```
def draw_roc_curve(actual,probas):
    fpr,tpr,thresholds=metrics.roc_curve(actual,probas,drop_intermediate=False)
    auc_score=metrics.roc_auc_score(actual,probas)
#     plt.figure(figsize=(5,3))
    plt.plot(fpr,tpr,label='Roc curve (Area=%0.2f)'%auc_score)
    plt.plot((0,1))

#     ------------------------------
    plt.title('ROC CURVE',fontdict={'size':10,'color':'brown'})
    plt.xlim([-0.05,1.2])
    plt.ylim([0.0,1.2])
    plt.xlabel('FALSE POSITIVE RATE',fontdict={'size':10,'color':'red'})
    plt.ylabel('TRUE POSITIVE RATE',fontdict={'size':10,'color':'red'})
    plt.legend(loc='upper left')


    return
```

In [66]:
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train,y_train_pred_logistic_proba)
```



Let's Do Predictions On The Test Set

In [67]:
```
y_test_pred_logistic=logistic_model1.predict(X_test)
```

In [68]:
```
y_test_pred_logistic_proba=logistic_model1.predict_proba(X_test)[:,1]
```

In [69]:
```
confusion_matrix_logistic_test=confusion_matrix(y_test,y_test_pred_logistic)
confusion_matrix_logistic_test
```

Out[69]:
```
array([[56853,    11],
       [   36,    62]], dtype=int64)
```

In [70]:
```
TN = confusion_matrix_logistic_test[0,0] # True negative
FP = confusion_matrix_logistic_test[0,1] # False positive
FN = confusion_matrix_logistic_test[1,0] # False negative
TP = confusion_matrix_logistic_test[1,1] # True positive
```

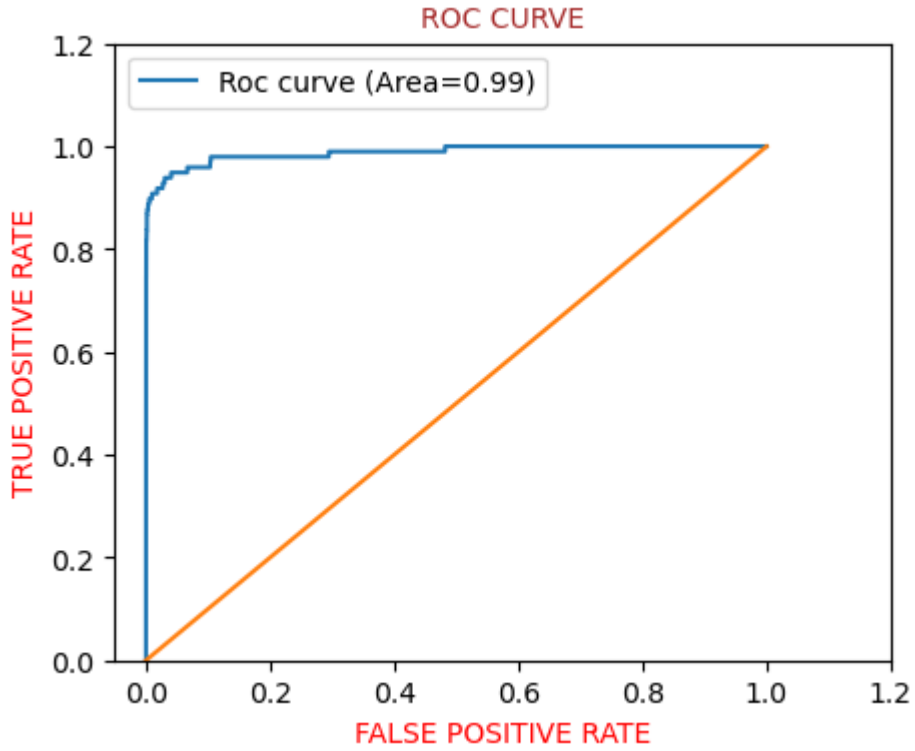
In [71]:
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.6326530612244898
The Specificity  is : 0.9998065559932471
```

In [72]:
```
    accuracy=metrics.accuracy_score(y_test,y_test_pred_logistic)
print('The Accuracy of Logistic Regression For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_logistic)
print("The F1-score of Logistic Regression For Test is :", F1_score)
```

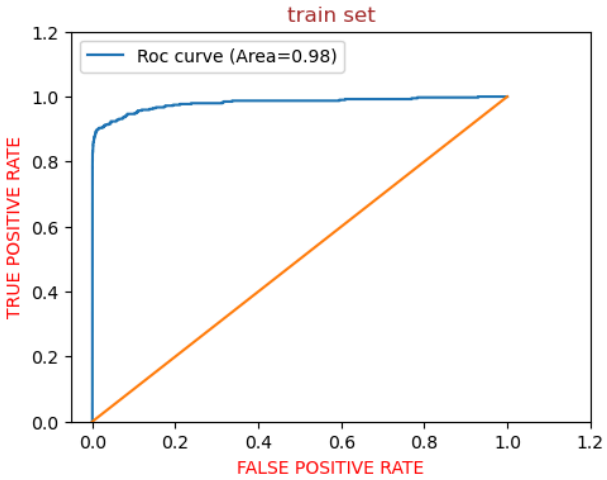
```
The Accuracy of Logistic Regression For Test is : 0.9991748885221726
The F1-score of Logistic Regression For Test is : 0.7251461988304093
```

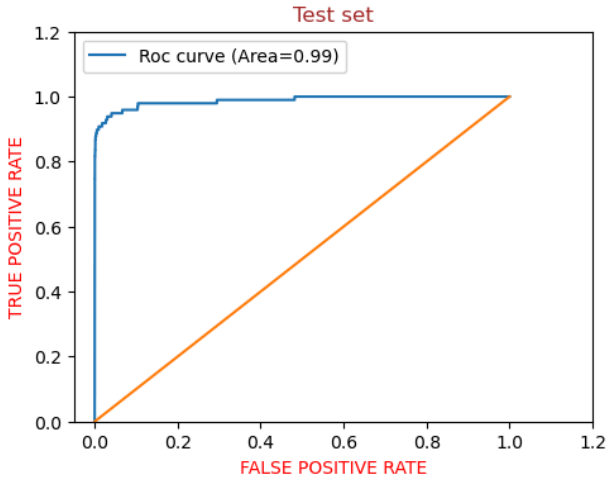
In [73]:
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_logistic_proba)
```

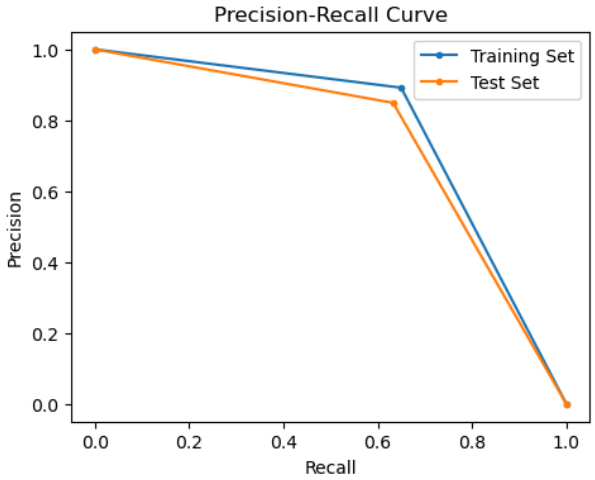


In [74]:
```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
```

```
draw_roc_curve(y_train,y_train_pred_logistic_proba)
plt.title('train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_logistic_proba)
plt.title('Test set')


plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train, y_train_pred_logistic)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_logistic)



# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```







Logistic Regression Model Performance Summary

Metric	Training Set	Test Set
Best Hyperparameter (C)	**1**	**1**
Best Recall (Sensitivity)	**0.6497**	**0.6327**
Best Specificity	**0.9999**	**0.9998**
Accuracy	**99.93%**	**99.92%**
F1-Score	**0.7518**	**0.7251**
Confusion Matrix	**TN=227420, FP=31, FN=138, TP=256**	**TN=56853, FP=11, FN=36, TP=62**
Precision-Recall Curve	**Plotted (Training Set)**	**Plotted (Test Set)**
ROC Curve AUC	**High (closer to 1)**	**High (closer to 1)**

Summary:

- **Recall (Sensitivity) is fairly good, showing that the model is reasonably effective at identifying fraud in both training and test sets.**
- **Specificity is very high, indicating minimal false positives.**
- **The F1-score demonstrates a balanced performance between precision and recall, though there is room for improvement, particularly in terms of recall.**

- **Confusion matrices reveal that the model correctly identifies a vast majority of non-fraudulent transactions but misses some fraud cases, which is common in imbalanced datasets.**

XG Boost

In [75]:
```
# !pip install xgboost
```

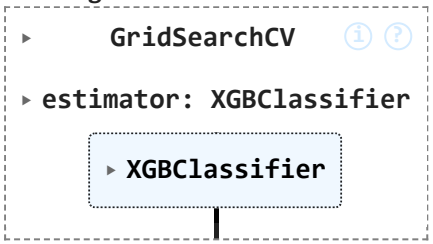
In [76]:
```
# importing Libraries
# !pip install xgboost
from xgboost import XGBClassifier
```

In [77]:
```
folds=3  #creating a KFold object
param_grid={'learning_rate':[0.2,0.6,0.8],'subsample':[0.3,0.6,0.9]}

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

In [78]:
```
model_cv=GridSearchCV(estimator=xgb_model,
                      param_grid=param_grid,cv=folds,
                      scoring='roc_auc',verbose=1,return_train_score=True)

model_cv.fit(X_train,y_train)
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits
```

Out[78]:



In [79]:
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[79]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_subsample	params	split0_test_score	split1_test_scor
0	4.553027	0.222773	0.163812	0.007824	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.985331	0.95882
1	4.648237	0.217798	0.174835	0.003959	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.985168	0.96638
2	4.584842	0.302095	0.149773	0.014528	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.983280	0.96990

In [80]:
```
cv_results.columns
```

Out[80]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_learning_rate', 'param_subsample', 'params', 'split0_test_score',
       'split1_test_score', 'split2_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score', 'split0_train_score',
       'split1_train_score', 'split2_train_score', 'mean_train_score',
       'std_train_score'],
      dtype='object')
```

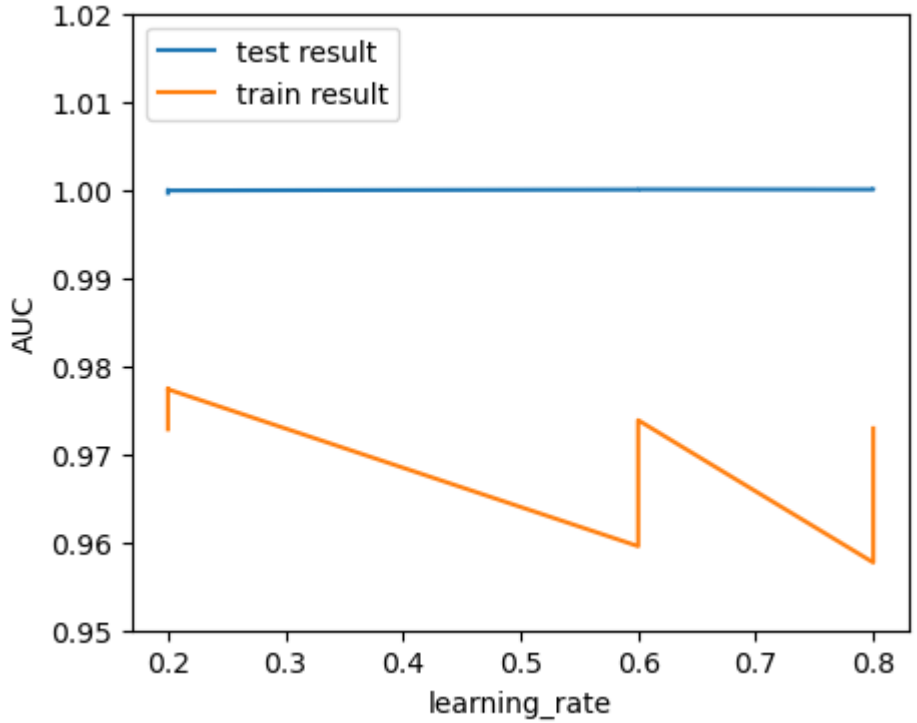
In [81]:
```
cv_results[['param_learning_rate', 'param_subsample','rank_test_score','mean_train_score','mean_test_score']]
```

Out[81]:

	param_learning_rate	param_subsample	rank_test_score	mean_train_score	mean_test_score
0	0.2	0.3	5	0.999600	0.972785
1	0.2	0.6	1	0.999944	0.977423
2	0.2	0.9	2	0.999881	0.977317
3	0.6	0.3	8	0.999981	0.959511
4	0.6	0.6	6	1.000000	0.968934
5	0.6	0.9	3	1.000000	0.973811
6	0.8	0.3	9	0.999993	0.957660
7	0.8	0.6	7	1.000000	0.964242
8	0.8	0.9	4	1.000000	0.972887

In [82]:

```
plt.figure(figsize=(5,4))
plt.plot(cv_results['param_learning_rate'],cv_results['mean_train_score'])
plt.plot(cv_results['param_learning_rate'],cv_results['mean_test_score'])
plt.xlabel('learning_rate')
plt.ylabel('AUC')
plt.legend(['test result', 'train result'], loc='upper left')
plt.ylim(0.95,1.02)
plt.show()

# plt.xscale('log')
```



In [83]:
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'learning_rate': 0.2, 'subsample': 0.6}
The Best score Is 0.9774228373889476
```

In [84]:
```
# chosen hyperparameters
# 'objective':'binary:logistic' which 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'}
```

In [85]:
```
# fit model on training data
xgb_model1 = XGBClassifier(params = params)
xgb_model1.fit(X_train, y_train)
```

Out[85]:

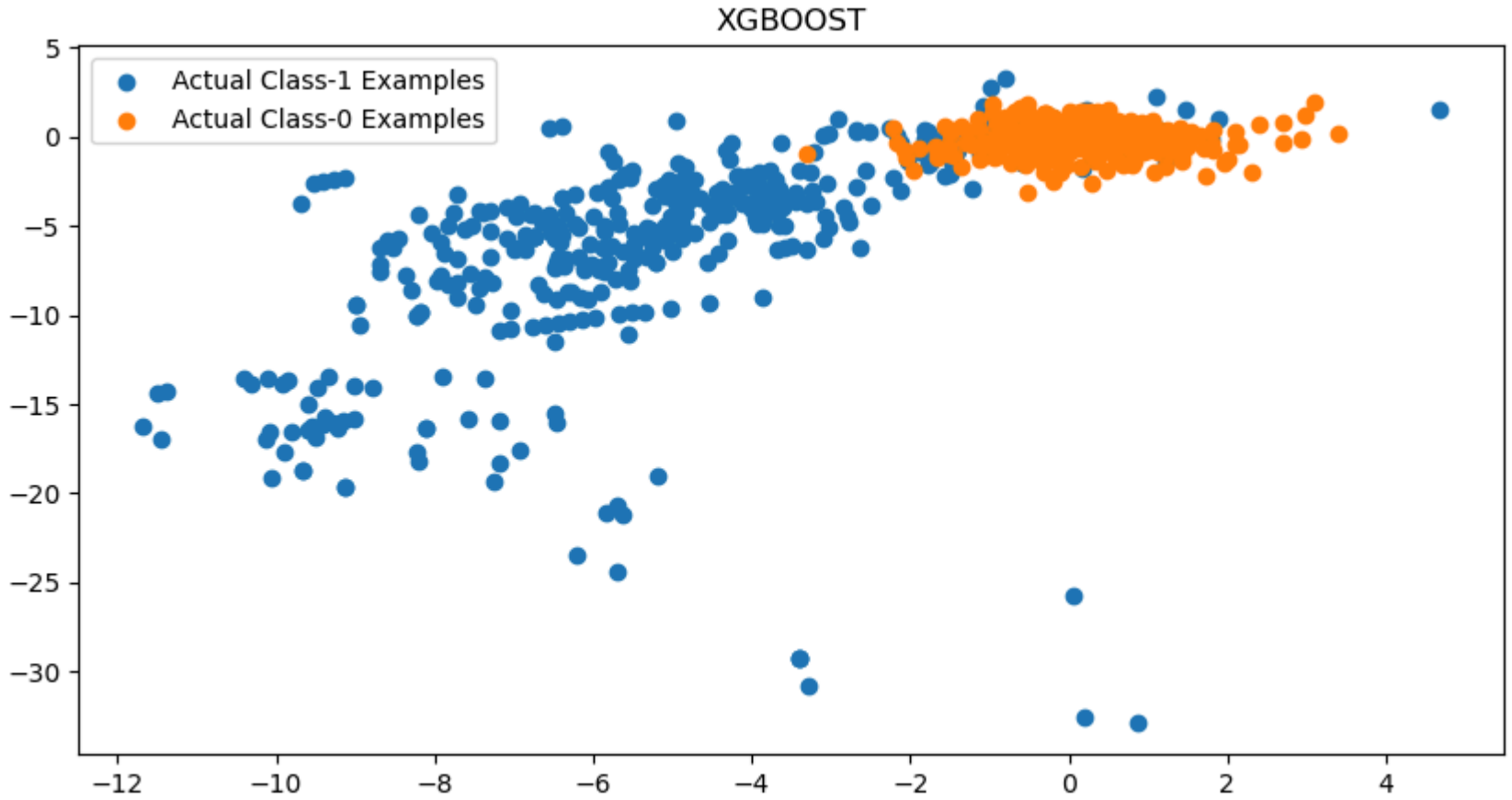
XGBClassifier

```
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,
```

In [86]:

```
var_imp = []
for i in xgb_model1.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(xgb_model1.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(xgb_model1.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(xgb_model1.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_model1.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(xgb_model1.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'] = [10, 5]

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.title('XGBOOST')
plt.legend()
plt.show()
```

```
Top var = 15
2nd Top var = 11
3rd Top var = 13
```



In [87]:
```
y_train_pred_xgboost=xgb_model1.predict(X_train)
```

In [88]:
```
confusion_matrix_xgboost_train=metrics.confusion_matrix(y_train,y_train_pred_xgboost)
```

In [89]:
```
confusion_matrix_xgboost_train
```

Out[89]:
```
array([[227451,      0],
       [     0,    394]], dtype=int64)
```

In [90]:
```
TN = confusion_matrix_xgboost_train[0,0] # True negative
FP = confusion_matrix_xgboost_train[0,1] # False positive
```

```
FN = confusion_matrix_xgboost_train[1,0] # False negative
TP = confusion_matrix_xgboost_train[1,1] # True positive
```

In [91]:
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 1.0
The Specificity  is : 1.0
```

In [92]:
```
print('The Accurays For The Train Set Of Xgboost is ',metrics.accuracy_score(y_train,y_train_pred_xgboost))
print('The F1-Score For The Train Set Of Xgboost is ',metrics.f1_score(y_train,y_train_pred_xgboost))
```

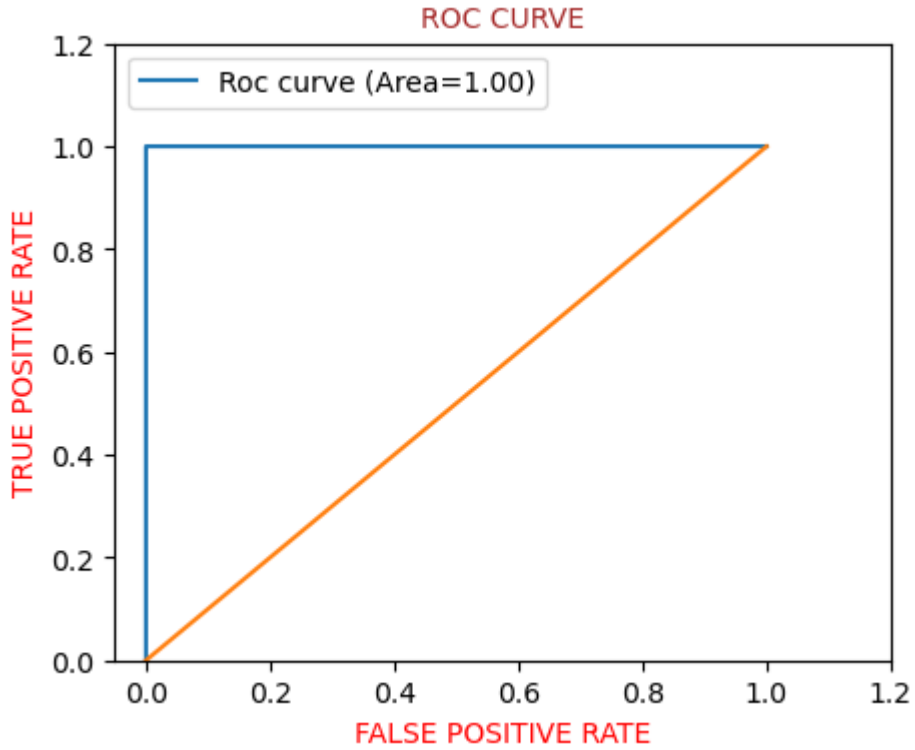
```
The Accurays For The Train Set Of Xgboost is  1.0
The F1-Score For The Train Set Of Xgboost is  1.0
```

In [93]:
```
# classificatino report
print(classification_report(y_train,y_train_pred_xgboost))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227451
           1       1.00      1.00      1.00       394

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

In [94]:
```
# predicted probability
y_train_pred_xgboost_proba=xgb_model1.predict_proba(X_train)[:,1]
```

In [95]:
```
plt.figure(figsize=(5,4))

draw_roc_curve(y_train,y_train_pred_xgboost_proba)
```



Let's Do Predictions On The `Test Set`

In [96]:
```
y_test_pred_xgboost=xgb_model1.predict(X_test)
```

In [97]:
```
y_test_pred_xgboost_proba=xgb_model1.predict_proba(X_test)[:,1]
```

Confusion Metrix For XGBoost `Test Case`

In [98]:
```
confusion_matrix_xgboost_test=confusion_matrix(y_test,y_test_pred_xgboost)
confusion_matrix_xgboost_test
```

Out[98]:
```
array([[56858,     6],
       [   17,    81]], dtype=int64)
```

In [99]:
```
TN = confusion_matrix_xgboost_test[0,0] # True negative
FP = confusion_matrix_xgboost_test[0,1] # False positive
```

```
FN = confusion_matrix_xgboost_test[1,0] # False negative
TP = confusion_matrix_xgboost_test[1,1] # True positive
```

In [100…

```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.826530612244898
The Specificity  is : 0.9998944850872257
```

In [101…

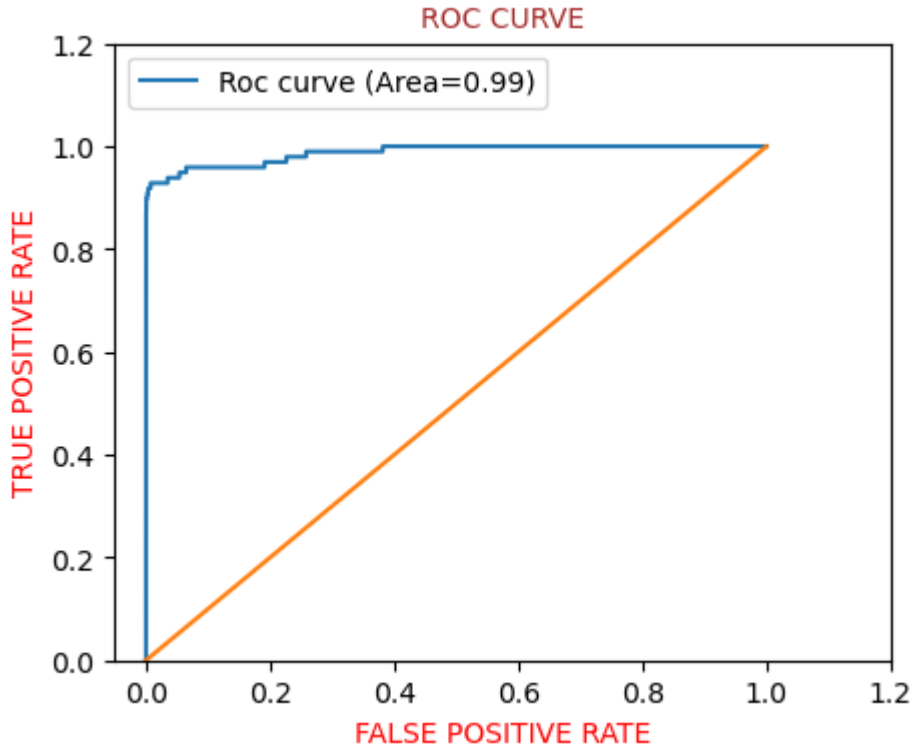
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_xgboost)
print('The Accuracy of XG Boost For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_xgboost)
print("The F1-score of XG Boost For Test is :", F1_score)
```

```
The Accuracy of XG Boost For Test is : 0.9995962220427653
The F1-score of XG Boost For Test is : 0.8756756756756757
```

`ROC_AUC` Curve on Test set For `XGBoost`

In [102…

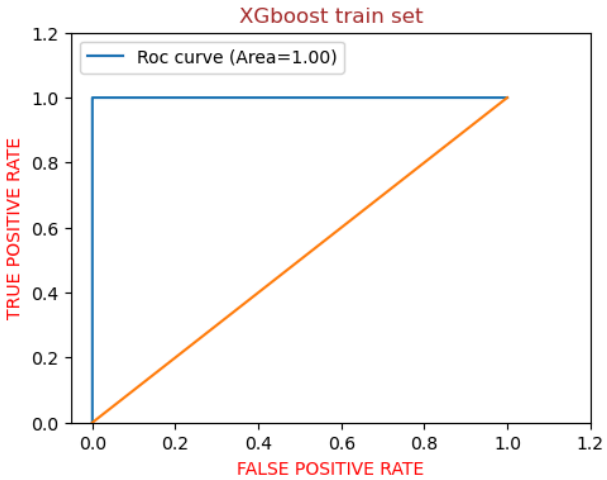
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
```

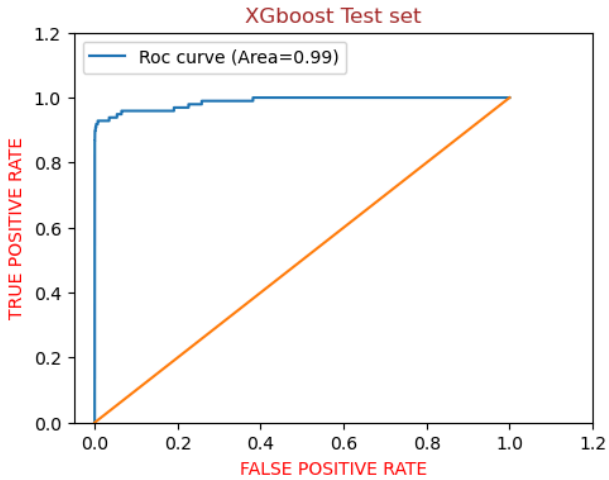


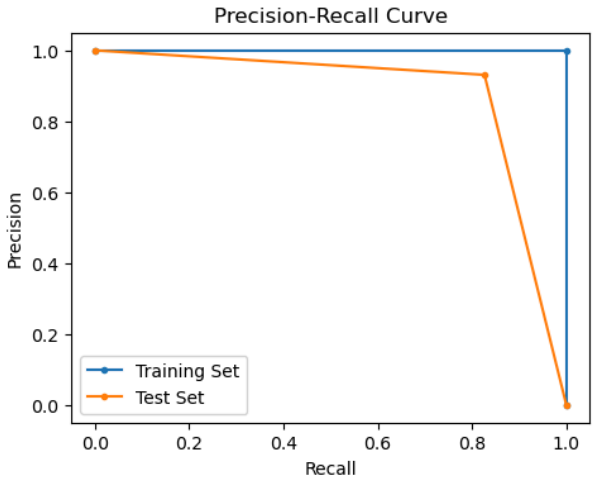
In [103…
```
from sklearn.model_selection import learning_curve
```

In [104…
```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train,y_train_pred_xgboost_proba)

plt.title('XGboost train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
plt.title('XGboost Test set')

plt.subplot(1,3,3)
# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train, y_train_pred_xgboost)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_xgboost)
```

```
# size
# plt.figure(figsize=(5,3))

# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()
plt.show()
```







XGBoost Model Metrics

Metric	Train Set	Test Set
Accuracy	**1.0**	**0.9996**
Sensitivity	**1.0**	**0.8265**
Specificity	**1.0**	**0.9999**
F1-Score	**1.0**	**0.8757**

Decision Trees

In [105…
```
# Importing libraries
from sklearn.tree import DecisionTreeClassifier
```

In [106…
```
# Create the parameter grid

start_time = time.time()


param_grid = {
    'criterion': ['gini'],
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(20, 70, 10),
    'min_samples_split': range(20, 70, 10),
}


#  Instantiate the grid search model
dtree = DecisionTreeClassifier()

model_cv = GridSearchCV(estimator = dtree,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = 3,
                        verbose = 1)

# Fit the grid search to the data
model_cv.fit(X_train,y_train)


end_time = time.time()



elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60

print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
Elapsed Time (Minutes): 23.61
```

In [107…
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[107]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_leaf	param_min_samples_split	
0	8.947303	0.332517	0.041944	0.001672	gini	5	20	20	'r
1	7.948617	0.815913	0.036772	0.010644	gini	5	20	30	'r
2	7.663947	0.281054	0.029319	0.003629	gini	5	20	40	'r

In [108…
```
cv_results.columns
```

Out[108]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_criterion', 'param_max_depth', 'param_min_samples_leaf',
       'param_min_samples_split', 'params', 'split0_test_score',
       'split1_test_score', 'split2_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```

In [109…
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'criterion': 'gini', 'max_depth': 10, 'min_samples_leaf': 60, 'min_samples_split': 50}
The Best score Is 0.9510262949996475
```

In [110…
```
dtree_model1=DecisionTreeClassifier(random_state=100,criterion='gini',max_depth=10,min_samples_leaf=20,min_samples_split=30)
```

In [111…
```
dtree_model1.fit(X_train,y_train)
```

Out[111]:

DecisionTreeClassifier

```
DecisionTreeClassifier(max_depth=10, min_samples_leaf=20, min_samples_split=30,
                       random_state=100)
```

In [112…
```
var_imp = []
for i in dtree_model1.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(dtree_model1.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(dtree_model1.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(dtree_model1.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(dtree_model1.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(dtree_model1.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'] = [10, 8]

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.title('Decision Trees')
plt.legend()
plt.show()
```

```
Top var = 18
2nd Top var = 15
3rd Top var = 11
```


Decision Trees
Actual Class-1 Examples
Actual Class-0 Examples
4
2
0
−2
−4
−6
−8
−10
−12
−20
−15
−10
−5
0
5
10

In [113…
```
y_train_pred_dtree=dtree_model1.predict(X_train)
```

In [114…
```
confusion_matrix_dtree_train=confusion_matrix(y_train,y_train_pred_dtree)
```

In [115…
```
confusion_matrix_dtree_train
```

Out[115]:
```
array([[227413,     38],
       [    96,    298]], dtype=int64)
```

In [116…
```
TN = confusion_matrix_dtree_train[0,0] # True negative
FP = confusion_matrix_dtree_train[0,1] # False positive
FN = confusion_matrix_dtree_train[1,0] # False negative
TP = confusion_matrix_dtree_train[1,1] # True positive
```

In [117…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.7563451776649747
The Specificity  is : 0.9998329310488853
```

In [118…
```
accuracy=metrics.accuracy_score(y_train,y_train_pred_dtree)
print('The Accuracy of Decision Trees For Train is :',accuracy)

F1_score=metrics.f1_score(y_train,y_train_pred_dtree)
print("The F1-score of  Decision Trees For Train is :", F1_score)
```

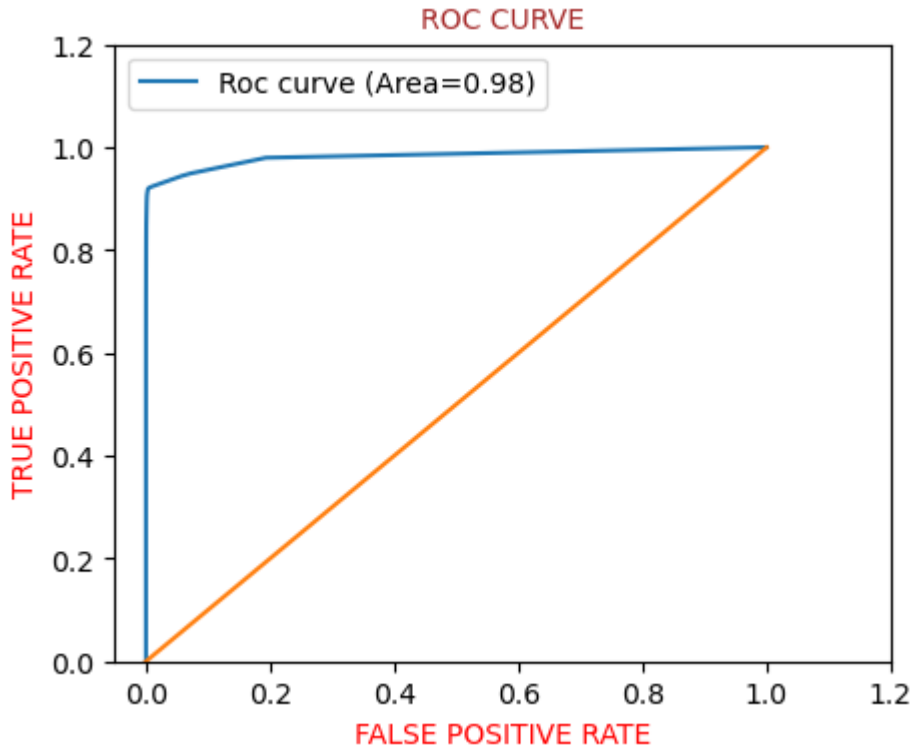
```
The Accuracy of Decision Trees For Train is : 0.9994118808839343
The F1-score of  Decision Trees For Train is : 0.8164383561643835
```

In [119…
```
# classification_report
print(classification_report(y_train, y_train_pred_dtree))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227451
           1       0.89      0.76      0.82       394

    accuracy                           1.00    227845
   macro avg       0.94      0.88      0.91    227845
weighted avg       1.00      1.00      1.00    227845
```

In [120…
```
y_train_pred_dtree_proba=dtree_model1.predict_proba(X_train)[:,1]
```

In [121…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train,y_train_pred_dtree_proba)
```



Let's Do Predictions On The `Test Set`

In [122…
```
y_test_pred_dtree=dtree_model1.predict(X_test)
```

In [123…
```
y_test_pred_dtree_proba=dtree_model1.predict_proba(X_test)[:,1]
```

In [124…
```
confusion_matrix_dtree_test=confusion_matrix(y_test,y_test_pred_dtree)
confusion_matrix_dtree_test
```

Out[124]:
```
array([[56854,    10],
       [   22,    76]], dtype=int64)
```

Confusion Metrix For Decision `Test Case`

In [125…
```
TN = confusion_matrix_dtree_test[0,0] # True negative
FP = confusion_matrix_dtree_test[0,1] # False positive
FN = confusion_matrix_dtree_test[1,0] # False negative
TP = confusion_matrix_dtree_test[1,1] # True positive
```

In [126…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.7755102040816326
The Specificity  is : 0.9998241418120428
```

In [127…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_dtree)
print('The Accuracy of Decision Tree For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_dtree)
print("The F1-score of Decision Tree For Test is :", F1_score)
```

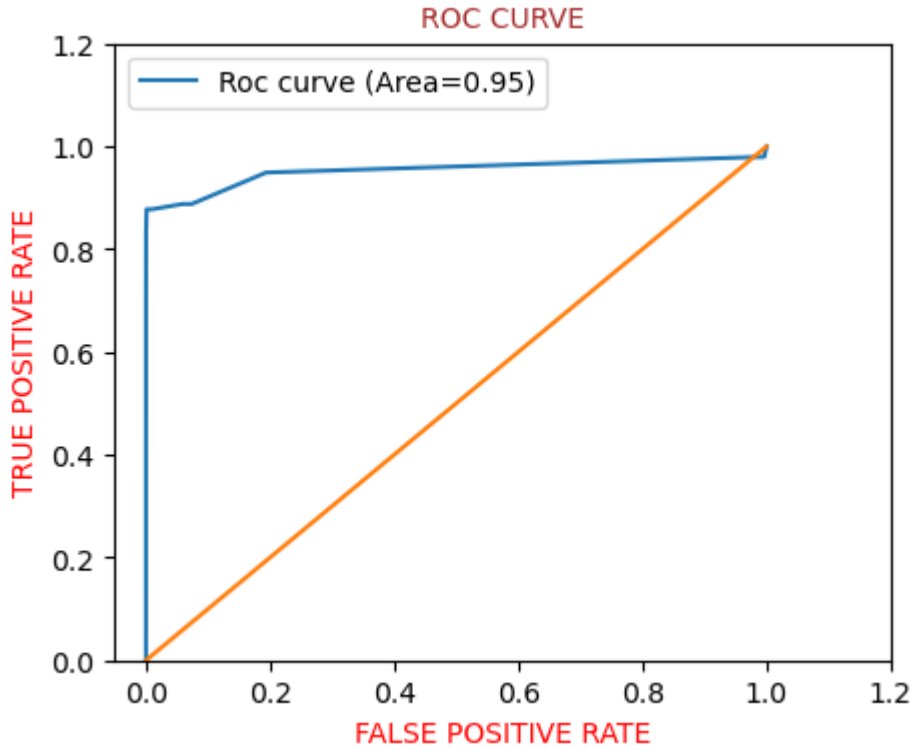
```
The Accuracy of Decision Tree For Test is : 0.9994382219725431
The F1-score of Decision Tree For Test is : 0.8260869565217391
```

In [128…
```
    # classification_report
print(classification_report(y_test, y_test_pred_dtree))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.88      0.78      0.83        98

    accuracy                           1.00     56962
   macro avg       0.94      0.89      0.91     56962
weighted avg       1.00      1.00      1.00     56962
```

`ROC_AUC` Curve on Test set For `Decision Tree`

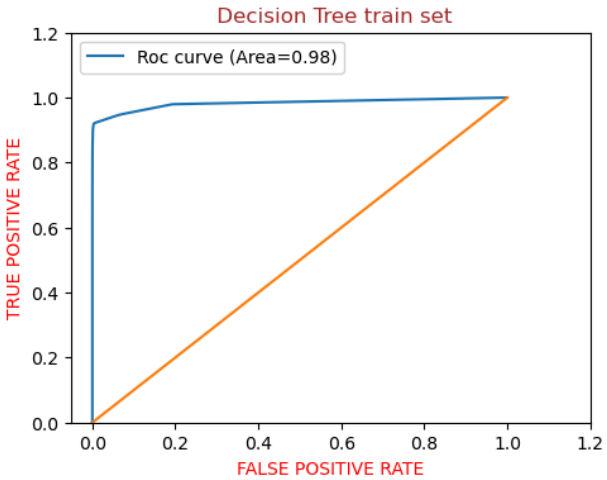
In [129…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_dtree_proba)
```

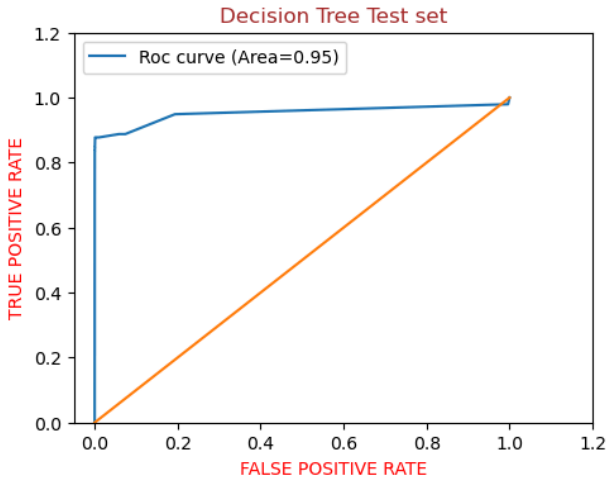


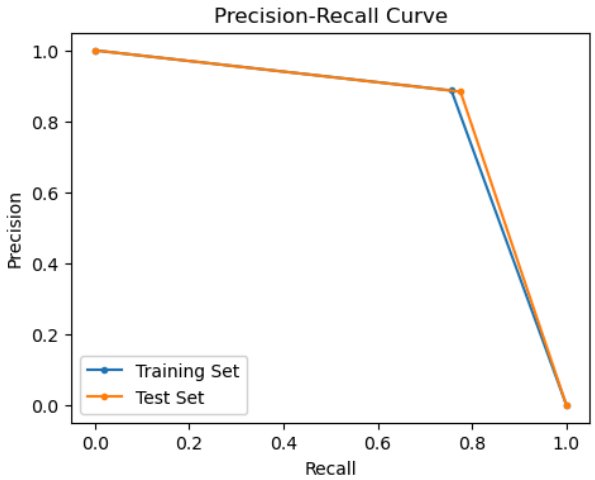
In [130…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train,y_train_pred_dtree_proba)

plt.title(' Decision Tree train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_dtree_proba)

plt.title('Decision Tree Test set')

plt.subplot(1,3,3)
# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train, y_train_pred_dtree)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_dtree)
```

```
# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()


plt.show()
```







Decision Trees Model Metrics

Metric	Train Set	Test Set
Accuracy	0.9994	0.9994
Sensitivity	0.7563	0.7755
Specificity	0.9998	0.9998
F1-Score	0.8164	0.8261

Random Forest

In [131…
```
# importing libraries
from sklearn.ensemble import RandomForestClassifier
```

In [132…
```
# Record the start time
start_time = time.time()

# Create the parameter grid
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [5, 10],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'bootstrap': [True]
}

random_forest=RandomForestClassifier()

model_cv = GridSearchCV(estimator=random_forest,
                        param_grid=param_grid,
                        scoring='roc_auc',
                        cv=3,
                        verbose=2,
                        n_jobs=-1,
                       return_train_score=True)



model_cv .fit(X_train,y_train)




# Record the end time
end_time = time.time()
# Calculate the elapsed time in seconds
elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60
```

```
print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
Elapsed Time (Minutes): 19.85
```

In [133…
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'bootstrap': True, 'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}
The Best score Is 0.9735733695069739
```

In [134…
```
random_forest=RandomForestClassifier(random_state=100,
                                     criterion='gini',
                                     max_depth=model_cv.best_params_['max_depth'],
                                     min_samples_leaf=model_cv.best_params_['min_samples_leaf'],
                                     min_samples_split=model_cv.best_params_['min_samples_split'],
                                     n_estimators=model_cv.best_params_['n_estimators'],bootstrap=True)
```

In [135…
```
random_forest.fit(X_train,y_train)
```

Out[135]:

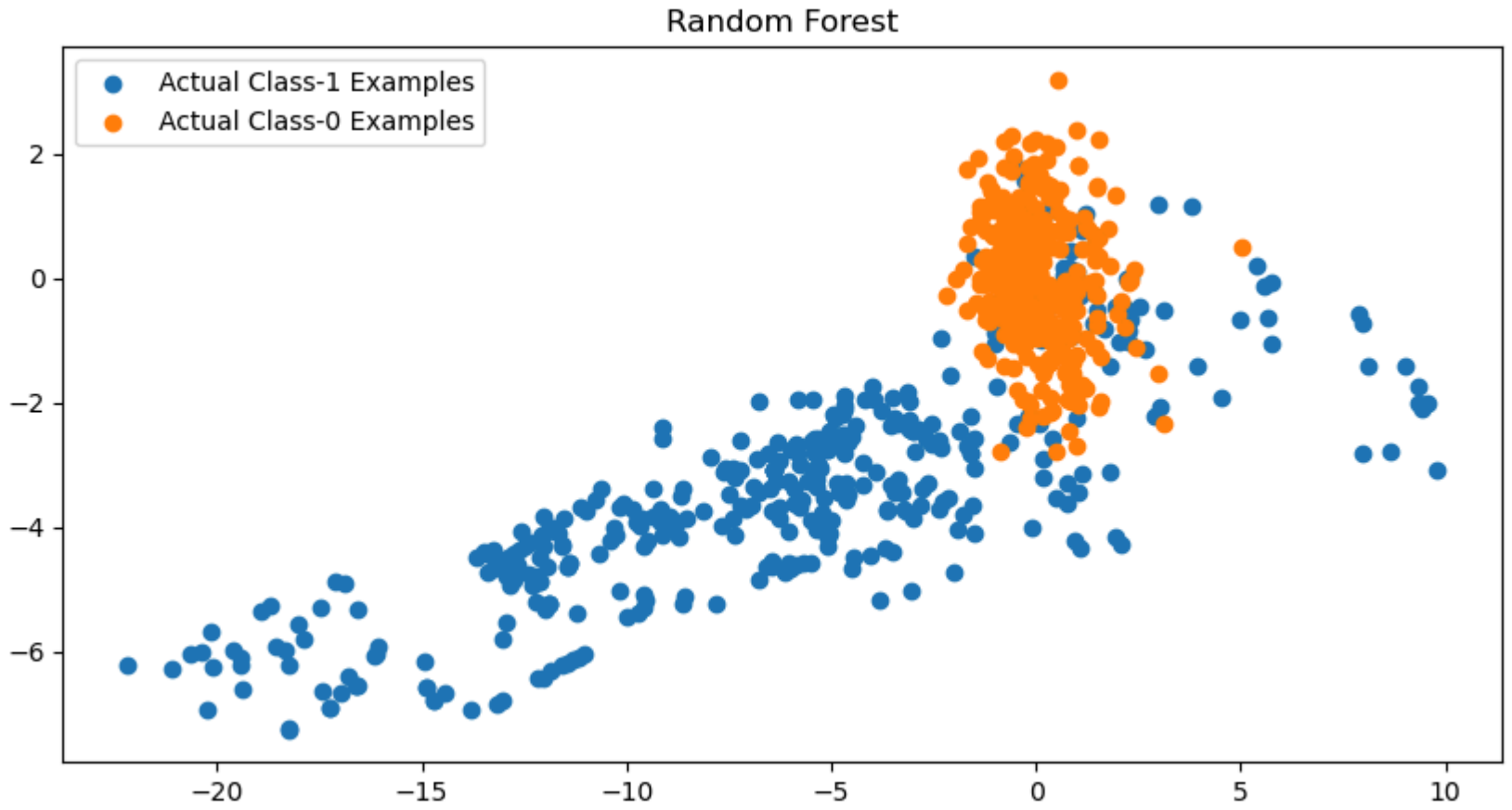
RandomForestClassifier

```
RandomForestClassifier(max_depth=10, min_samples_split=5, random_state=100)
```

In [136…
```
var_imp = []
for i in random_forest.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(random_forest.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(random_forest.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(random_forest.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(random_forest.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(random_forest.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'] = [10, 5]

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.title('Random Forest')
plt.legend()
plt.show()
```

```
Top var = 18
2nd Top var = 13
3rd Top var = 15
```



In [137…

```
y_train_pred_random_forest=random_forest.predict(X_train)
```

In [138…
```
confusion_matrix_random_forest_train=confusion_matrix(y_train,y_train_pred_random_forest)
```

In [139…
```
confusion_matrix_random_forest_train
```

Out[139]:
```
array([[227450,      1],
       [    67,    327]], dtype=int64)
```

In [140…
```
    TN = confusion_matrix_random_forest_train[0,0] # True negative
FP = confusion_matrix_random_forest_train[0,1] # False positive
FN = confusion_matrix_random_forest_train[1,0] # False negative
TP = confusion_matrix_random_forest_train[1,1] # True positive
```

In [141…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8299492385786802
The Specificity  is : 0.9999956034486549
```

In [142…
```
accuracy=metrics.accuracy_score(y_train,y_train_pred_random_forest)
print('The Accuracy of random_forest For Train is :',accuracy)

F1_score=metrics.f1_score(y_train,y_train_pred_random_forest)
print("The F1-score of  random_forest For Train is :", F1_score)
```

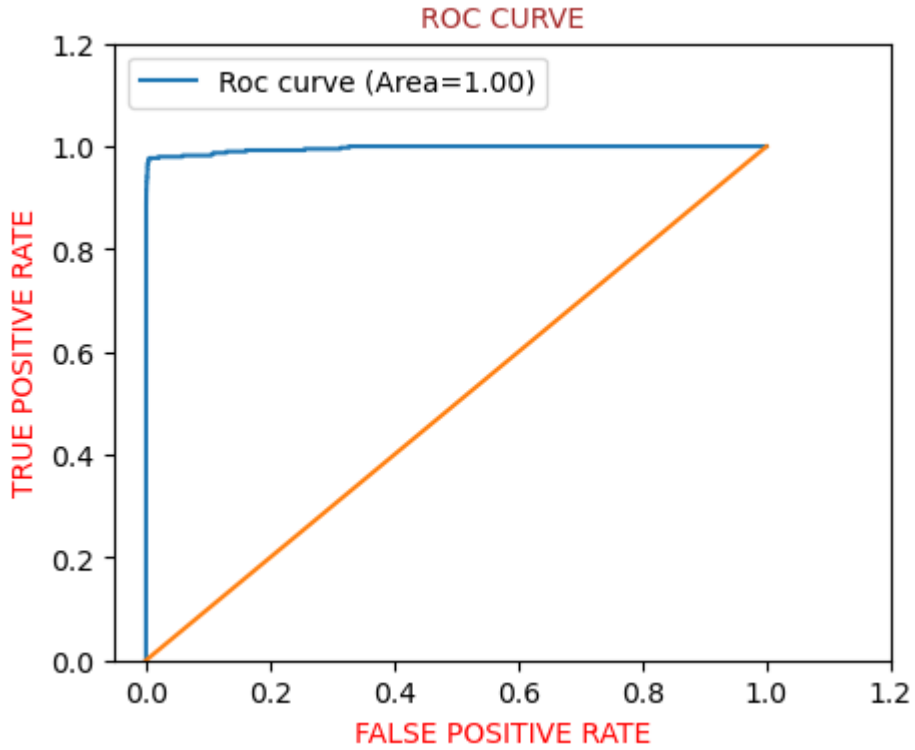
```
The Accuracy of random_forest For Train is : 0.9997015514933397
The F1-score of  random_forest For Train is : 0.9058171745152355
```

In [143…
```
# classification_report
print(classification_report(y_train, y_train_pred_random_forest))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227451
           1       1.00      0.83      0.91       394

    accuracy                           1.00    227845
   macro avg       1.00      0.91      0.95    227845
weighted avg       1.00      1.00      1.00    227845
```

In [144…
```
y_train_pred_random_forest_proba=random_forest.predict_proba(X_train)[:,1]
```

In [145…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train,y_train_pred_random_forest_proba)
```



Let's Do Predictions On The `Test Set`

In [146…
```
y_test_pred_random_forest=random_forest.predict(X_test)
```

In [147…
```
y_test_pred_random_forest_proba=random_forest.predict_proba(X_test)[:,1]
```

Confusion Metrix For Random Forest `Test Case`

In [148…
```
confusion_matrix_random_forest_test=confusion_matrix(y_test,y_test_pred_random_forest)
confusion_matrix_random_forest_test
```

Out[148]:
```
array([[56859,     5],
       [   18,    80]], dtype=int64)
```

In [149…
```
TN = confusion_matrix_random_forest_test[0,0] # True negative
FP = confusion_matrix_random_forest_test[0,1] # False positive
```

```
FN = confusion_matrix_random_forest_test[1,0] # False negative
TP = confusion_matrix_random_forest_test[1,1] # True positive
```

In [150…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8163265306122449
The Specificity  is : 0.9999120709060214
```

In [151…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_random_forest)
print('The Accuracy of random_forest For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_random_forest)
print("The F1-score of random_forest For Test is :", F1_score)
```

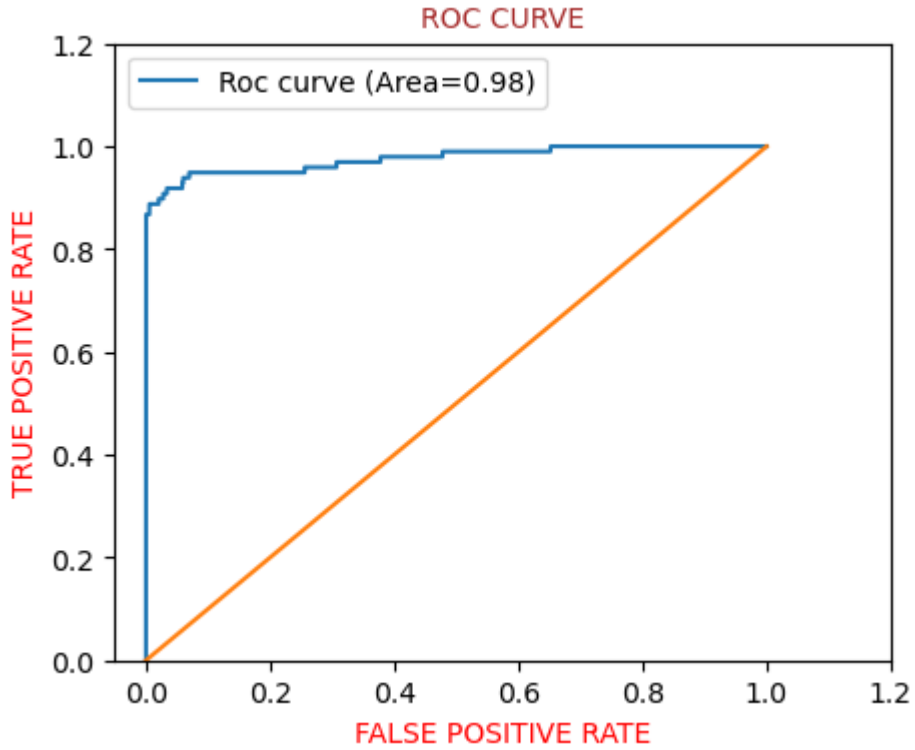
```
The Accuracy of random_forest For Test is : 0.9995962220427653
The F1-score of random_forest For Test is : 0.8743169398907104
```

In [152…
```
# classification_report
print(classification_report(y_test, y_test_pred_random_forest))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.94      0.82      0.87        98

    accuracy                           1.00     56962
   macro avg       0.97      0.91      0.94     56962
weighted avg       1.00      1.00      1.00     56962
```

`ROC_AUC` Curve on Test set For `Random Forest`

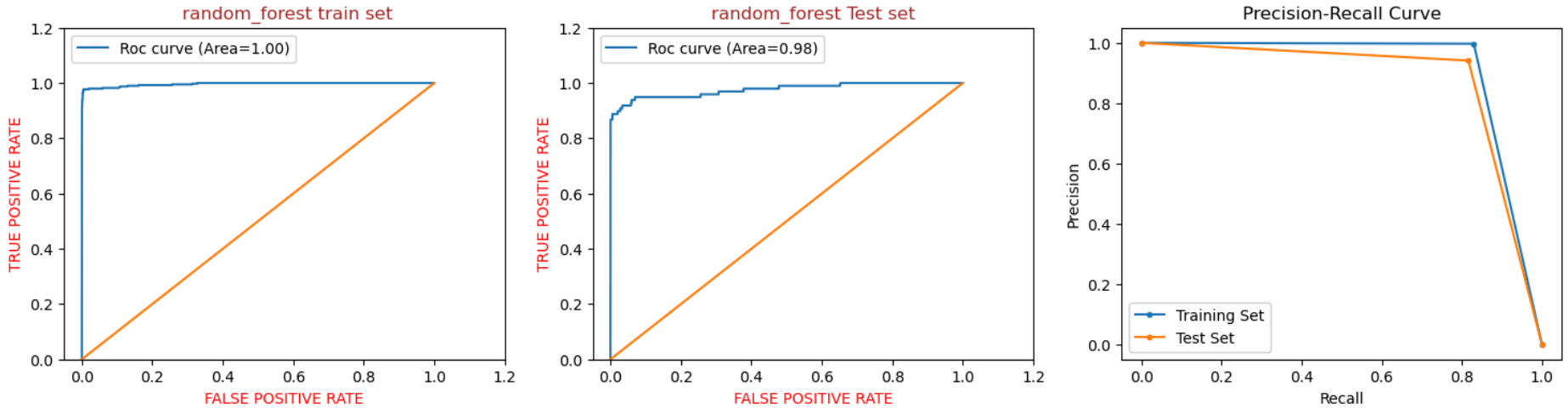
In [153…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_random_forest_proba)
```



In [154…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train,y_train_pred_random_forest_proba)

plt.title(' random_forest train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_random_forest_proba)

plt.title('random_forest Test set')

plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train, y_train_pred_random_forest)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_random_forest)
```

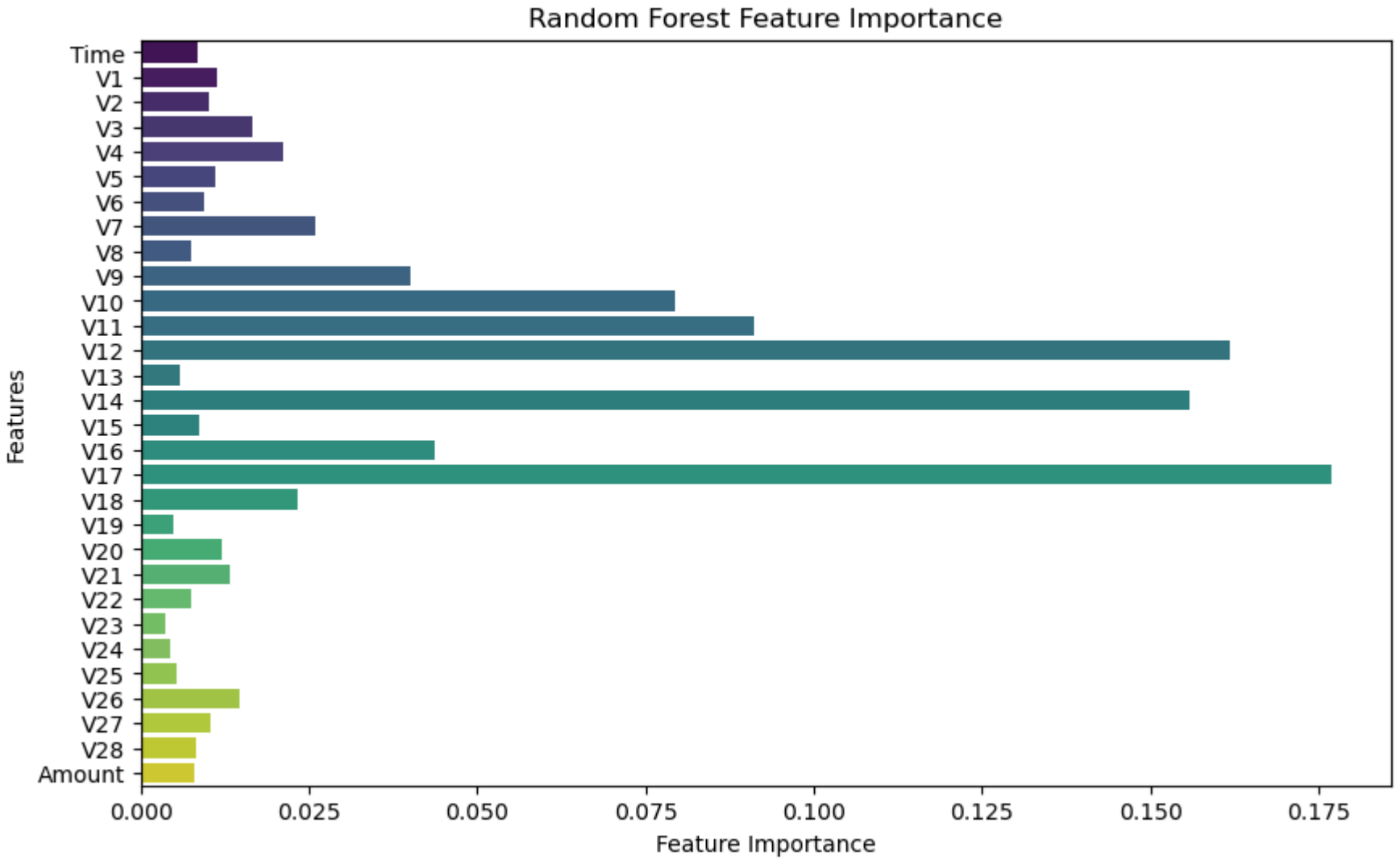
```
# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```



In [155…

```
# from sklearn.ensemble import RandomForestClassifier
# import matplotlib.pyplot as plt

# Assuming clf is your trained Random Forest classifier
feature_importances = random_forest.feature_importances_

# Get feature names if available
feature_names = X.columns if 'X' in locals() else range(len(feature_importances))

# Plotting Feature Importances
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances, y=feature_names, orient='h', palette='viridis')
plt.xlabel('Feature Importance')
plt.ylabel('Features')
```

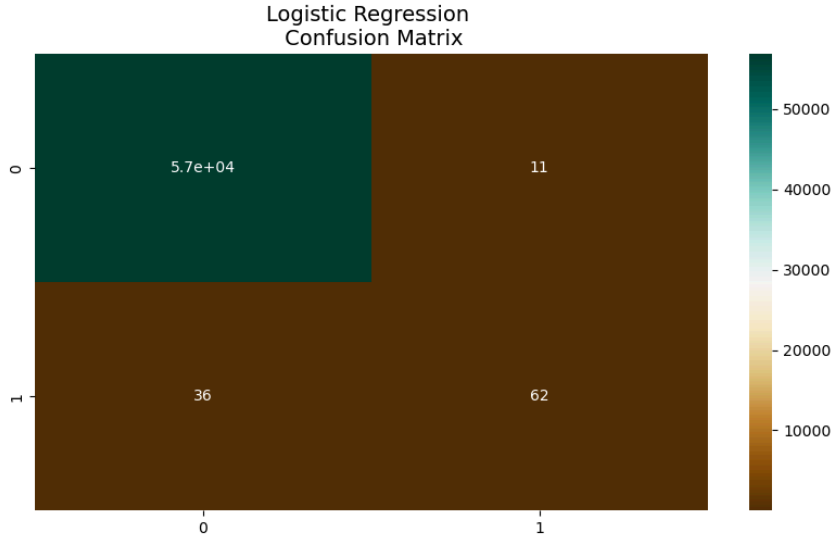
```
plt.title('Random Forest Feature Importance')
plt.show()
```

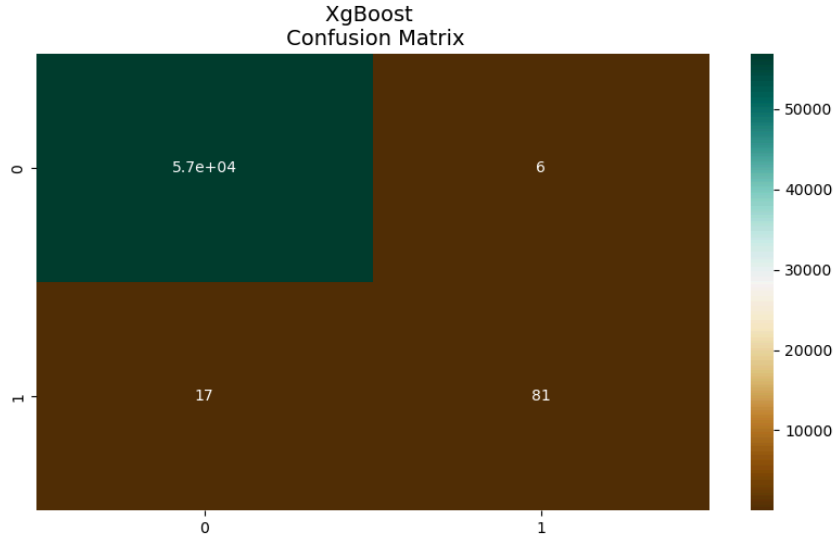


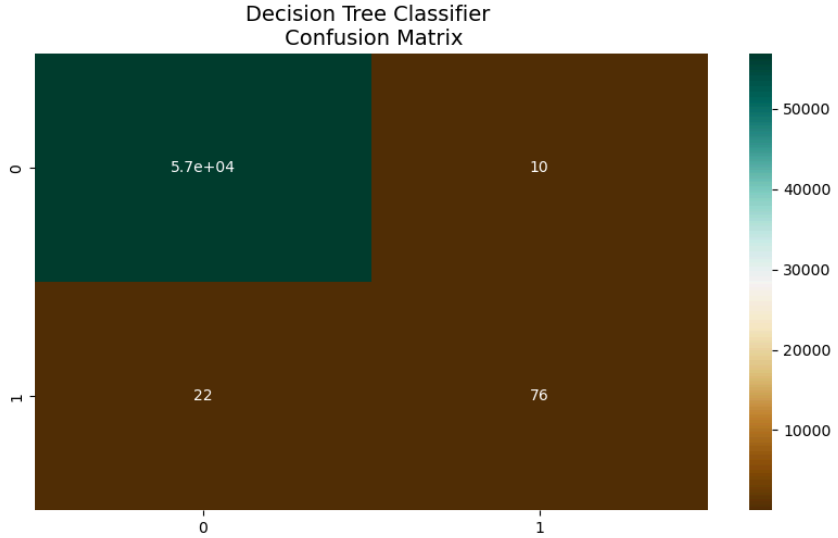
In [156…

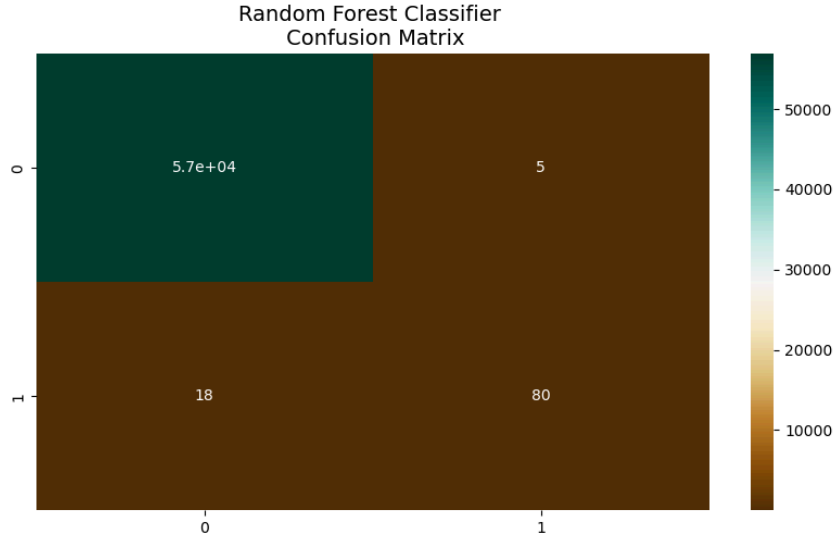
```
from sklearn.metrics import confusion_matrix

# all models fitted using SMOTE technique
y_test_pred_logistic = logistic_model1.predict(X_test)

y_test_pred_xgboost = xgb_model1.predict(X_test)
```

```
y_test_pred_dtree = dtree_model1.predict(X_test)
y_test_pred_random_forest = random_forest.predict(X_test)


log_reg_cf = confusion_matrix(y_test, y_test_pred_logistic)
xgboost_cf = confusion_matrix(y_test, y_test_pred_xgboost)
dtree_cf = confusion_matrix(y_test, y_test_pred_dtree)
random_forest_cf = confusion_matrix(y_test, y_test_pred_random_forest)

fig, ax = plt.subplots(2, 2,figsize=(22,12))


sns.heatmap(log_reg_cf, ax=ax[0][0], annot=True, cmap=plt.cm.BrBG)
ax[0, 0].set_title("Logistic Regression \n Confusion Matrix", fontsize=14)
# ax[0, 0].set_xticklabels(['', ''], fontsize=14, rotation=90)
# ax[0, 0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(xgboost_cf, ax=ax[0][1], annot=True, cmap=plt.cm.BrBG)
ax[0][1].set_title("XgBoost \n Confusion Matrix", fontsize=14)


sns.heatmap(dtree_cf, ax=ax[1][0], annot=True, cmap=plt.cm.BrBG)
ax[1][0].set_title("Decision Tree Classifier \n Confusion Matrix", fontsize=14)


sns.heatmap(random_forest_cf, ax=ax[1][1], annot=True, cmap=plt.cm.BrBG)
ax[1][1].set_title("Random Forest Classifier \n Confusion Matrix", fontsize=14)



plt.show()
```









Random Forest Classifier Results

Metric	Training Set	Test Set
Accuracy	**0.9997**	**0.9996**
Sensitivity	**0.8299**	**0.8163**

Metric	Training Set	Test Set
Specificity	0.9999	0.9999
F1-Score	0.9058	0.8743

K Nearest Neighbors KNN

We `cannot` build the model `by using KNN` because our Data Set contains 284807 `KNN is Needs more memory power` as the data point increases it should store the data points

As you see the below code i have tried to run the model based on KNN but it is `consuming more time`

If You Still Want To Use KNN, You Can Try Reducing The Number Of Neighbors (K) Or Use Approximations To Speed Up The Computation. Additionally, Optimizing The Implementation (Using A More Optimized Library Or Parallel Computing) Might Hel

In [157…

```
from sklearn.neighbors import KNeighborsClassifier
```

In [158…

```
# # Record the start time
# start_time = time.time()

# # Create the modified parameter grid
# param_grid = {
#     'n_neighbors': [3, 5],
#     'weights': ['uniform', 'distance'],
#     'p': [1, 2],
#     'metric': ['euclidean', 'manhattan']
# }
# # Create a KNN classifier
# knn = KNeighborsClassifier()

# # Use GridSearchCV to find the best parameters
# model_cv = GridSearchCV(estimator=knn, param_grid=param_grid, scoring='recall', cv=5)
# model_cv.fit(X_train, y_train)
```

```
# # Record the end time
# end_time = time.time()
# # Calculate the elapsed time in seconds
# elapsed_time_seconds = end_time - start_time

# # Convert elapsed time to minutes
# elapsed_time_minutes = elapsed_time_seconds / 60

# print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

Choosing The Best Model On Imbalanced Data Set Out Of All 4 Models Logistic Regression,XGBoost,Decision Tree, Random Forest

- **Most Of The Models Performed Well On `Training Set` and `Test Set`**
- **Each Model Has Its Strengths. If Your `Primary Concern Is Detecting Fraud` (Which Is Often The Case In Credit Card Fraud Detection), You Might Prioritize Models With `Higher Sensitivity` Because It Focuses On `Correctly Identifying Fraud Cases`.**
- **`XG Boost` Exhibits `The Highest Sensitivity On The Test Set` Among The Models You `Evaluated (0.816)`. `Sensitivity Is Crucial In Fraud Detection` Because It Represents The Ability Of The Model To `Correctly Identify Actual Fraud Cases`, Minimizing False Negatives.**
- **XG Boost Is Known For Its `Robustness To Imbalanced Datasets`. It Handles Class Imbalance Well And Often Provides Better Performance Compared To Other Algorithms When Dealing With Skewed Class Distributions.**
- **`XG Boost` Shows A `Good Balance Between Sensitivity, Specificity, And Accuracy`. While Achieving A `High Sensitivity`, It Also Maintains A `High Level Of Specificity (0.999)` And `Accuracy (0.999)`, Indicating An Overall Strong Performance**
- **`XG Boost` Excels In Various Metrics Like Sensitivity, Specificity, Accuracy, F1-Score, And ROC Curve, Demonstrating Its `Overall Robust And Balanced Performance`**

Handling Class Imbalance

The Data Exhibits A Significant Class Imbalance, With Over `2,84,315 Cases Labeled As 0` And Only Around` 492 Cases Labeled As 1`. Machine Learning Algorithms Perform Best With Balanced Class Representation. However, In This Scenario, Any Model Built Will

Be More Knowledgeable About Non-Fraudulent Cases Than Fraudulent Ones Due To The Imbalance. This Imbalance Creates A Challenge Known As The Minority Class Problem.

To Tackle The Challenge Of `Class Imbalance`, Various Methods Can Be Employed :

- **`Undersampling` : This Approach Involves Selecting Fewer Data Points From The Majority Class During The Model-Building Process. For Instance, If The Minority Class Has Only 500 Data Points, An Equal Number Of 500 Data Points Are Chosen From The Majority Class To Somewhat Balance The Classes. However, Undersampling Has Practical Limitations, Such As Losing Over 99% Of The Original Data, Making It Less Effective In Real-World Scenarios. `Select Fewer Data Points From The Majority Class, But It's Less Effective Due To Significant Data Loss`.**
- **`Oversampling` : In This Method, `Weights` Are Assigned To Randomly Selected Data Points From The Minority Class. This Means That The Occurrence Of Each Data Point Is `Multiplied By The Assigned Weight For Optimization`, But It `May Not Add New Information` And Can `Exaggerate Existing Patterns`.**
- **`Smote ` : Intelligently Generates Synthetic Samples Between Existing Minority Class Data Points, Introducing Diversity And Reducing Bias.**
- **`Adasyn` : Similar To Smote, It Adapts By Introducing Synthetic Samples Based On A Density Distribution, Focusing On Harder-To-Learn Minority Examples.**

SMOTE Synthetic Minority Over-Sampling

- **SMOTE, Which Stands For Synthetic Minority Over-Sampling Technique, Is A `Popular Technique` In The Field Of Machine Learning, Particularly In The Context Of `Imbalanced Classification Problems`. The `Primary Goal` Of SMOTE Is To Address The `Issue Of Class Imbalance` By Generating `Synthetic Samples` For The Minority Class.**
- **In Imbalanced Datasets, Where `One Class (The Minority Class)` Has Significantly Fewer Examples Than The Other `(The Majority Class)`, Machine Learning Models May Struggle To `Effectively Learn Patterns In The Minority Class`. SMOTE Addresses This By Creating Synthetic Instances Of The Minority Class Through A Process Of Interpolating Between Existing Minority Class Samples.**

In [159…
```
sm=SMOTE(sampling_strategy='auto',random_state=42)
```

In [160…
```
# fitting the SMOTE to the train set
X_train_smote, y_train_smote = sm.fit_resample(X_train, y_train)
```

In [161…
```
print('Before SMOTE oversampling X_train shape=',X_train.shape)
print('After SMOTE oversampling X_train shape=',X_train_smote.shape)
```

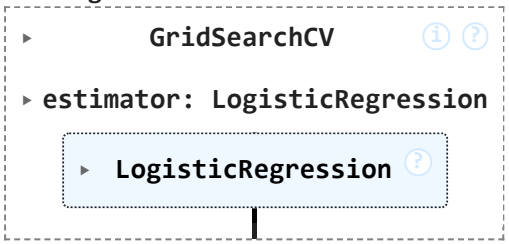
```
Before SMOTE oversampling X_train shape= (227845, 30)
After SMOTE oversampling X_train shape= (454902, 30)
```

Logistic Regression By SMOTE

In [162…
```
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'penalty': ['l1', 'l2'],  # Regularization type
}

# Create K-fold cross-validation
folds = KFold(n_splits=5, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for logistic regression
model_cv = GridSearchCV(estimator=LogisticRegression(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=-1)
model_cv.fit(X_train_smote, y_train_smote)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

Out[162]:



In [163…
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[163]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_penalty	params	split0_test_score	split1_test_score	split2_test_score
0	0.500491	0.020162	0.000000	0.000000	0.01	l1	{'C': 0.01, 'penalty': 'l1'}	NaN	NaN	NaN
1	3.483778	0.105422	0.159446	0.013508	0.01	l2	{'C': 0.01, 'penalty': 'l2'}	0.915648	0.915451	0.916289
2	0.484868	0.036661	0.000000	0.000000	0.1	l1	{'C': 0.1, 'penalty': 'l1'}	NaN	NaN	NaN

In [164…
```
cv_results.columns
```

Out[164]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_C', 'param_penalty', 'params', 'split0_test_score',
       'split1_test_score', 'split2_test_score', 'split3_test_score',
       'split4_test_score', 'mean_test_score', 'std_test_score',
       'rank_test_score'],
      dtype='object')
```

In [165…
```
# best score
best_score=model_cv.best_score_
# best params
best_params=model_cv.best_params_
```

In [166…
```
print(f"The Best Score Is {best_score}")
print(f"The Best Params Is {best_params}")
```

```
The Best Score Is 0.9180097343835285
The Best Params Is {'C': 10, 'penalty': 'l2'}
```

In [167…
```
logistic_=LogisticRegression(C=10,penalty='l2')
```

In [168…
```
# fiitting the model on train set
logistic_smote=logistic_.fit(X_train_smote,y_train_smote)
```

In [169…
```
y_train_pred_logistic_smote=logistic_smote.predict(X_train_smote)
```

Confusion Metrix For `Logistic Regression`

In [170…
```
confusion_matrix_logistic_train_smote=metrics.confusion_matrix(y_train_smote,y_train_pred_logistic_smote)
```

In [171…
```
confusion_matrix_logistic_train_smote
```

Out[171]:
```
array([[220930,   6521],
       [ 18675, 208776]], dtype=int64)
```

In [172…
```
TN = confusion_matrix_logistic_train_smote[0,0] # True negative
FP = confusion_matrix_logistic_train_smote[0,1] # False positive
FN = confusion_matrix_logistic_train_smote[1,0] # False negative
TP = confusion_matrix_logistic_train_smote[1,1] # True positive
```

In [173…
```
calculation_metrics(TN,FP,FN,TP)
```

```
The Sensitivity  is : 0.9178944036297928
The Specificity  is : 0.9713300886784406
```

In [174…
```
accuracy=metrics.accuracy_score(y_train_smote,y_train_pred_logistic_smote)
print('The Accuracy of Logistic Regression By Using Smote For Train is :',accuracy)

F1_score=metrics.f1_score(y_train_smote,y_train_pred_logistic_smote)
print("The F1-score of Logistic Regression By Using Smote For Train is :", F1_score)
```

```
The Accuracy of Logistic Regression By Using Smote For Train is : 0.9446122461541168
The F1-score of Logistic Regression By Using Smote For Train is : 0.943091781329334
```

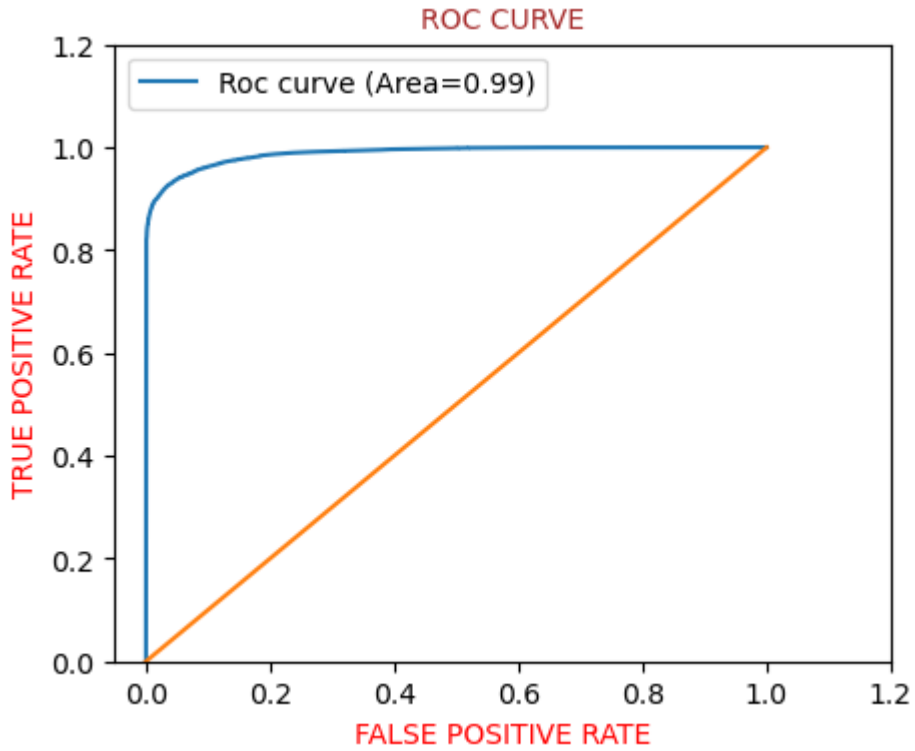
In [175…
```
# classificatino report
print(classification_report(y_train_smote,y_train_pred_logistic_smote))
```

```
              precision    recall  f1-score   support

           0       0.92      0.97      0.95    227451
           1       0.97      0.92      0.94    227451

    accuracy                           0.94    454902
   macro avg       0.95      0.94      0.94    454902
weighted avg       0.95      0.94      0.94    454902
```

In [176…
```
# predicted probability
y_train_pred_logistic_proba_smote=logistic_smote.predict_proba(X_train_smote)[:,1]
```

`ROC_AUC` Curve on Training set For `Logistic Regression` Smote

In [177…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train_smote,y_train_pred_logistic_proba_smote)
```



`SMOTE` (Synthetic Minority Over-Sampling Technique) Should Only Be `Applied To The Training Dataset`, And `Not To The Test Dataset`. The Reason Behind This Is That Any `Oversampling Or Data Manipulation Techniques`, Including SMOTE, Should Be `Part Of The Training Process` And Should `Not Leak Information` From The Test Set Into The Training Set.

Here's Why You Should Only Apply SMOTE To The Training Data :

- **`Simulating Real-World Scenarios`: In Real-World Scenarios, Your Model Will Encounter New, Unseen Data When It's Deployed. Therefore, Any Preprocessing Steps, Such As Oversampling, Should Be Applied Only To The Training Data To Mimic This Real-World Scenario.**

- **`Preventing Data Leakage` : If You Apply SMOTE To The Entire Dataset (Including Both Training And Test Sets), You Risk Introducing Information From The Test Set Into The Training Set. This Can Lead To `Overly Optimistic Evaluations` Of Your Model's Performance Because It Has Already Seen Some Of The Test Set Examples During Training.**

- **`Maintaining Independence` : The `Test Set` Should Be Kept `Independent` Of The Training Set To Provide An `Unbiased Evaluation` Of The Model's Generalization Performance On New, Unseen Data.**

  In Summary, Always Apply SMOTE, Or Any Other Data Preprocessing Technique, Exclusively To The Training Dataset Before Splitting It Into Training And Validation Sets. This Ensures A More Accurate Representation Of The Model's Performance On Truly Unseen Data During Evaluation.

Let's Do Predictions On The `Test Set`

In [178…
```
y_test_pred_logistic=logistic_smote.predict(X_test)
```

In [179…
```
y_test_pred_logistic_proba=logistic_smote.predict_proba(X_test)[:,1]
```

In [180…
```
confusion_matrix_logistic_test=confusion_matrix(y_test,y_test_pred_logistic)
confusion_matrix_logistic_test
```

Out[180]:
```
array([[55219,  1645],
       [    5,    93]], dtype=int64)
```

In [181…
```
TN = confusion_matrix_logistic_test[0,0] # True negative
FP = confusion_matrix_logistic_test[0,1] # False positive
FN = confusion_matrix_logistic_test[1,0] # False negative
TP = confusion_matrix_logistic_test[1,1] # True positive
```

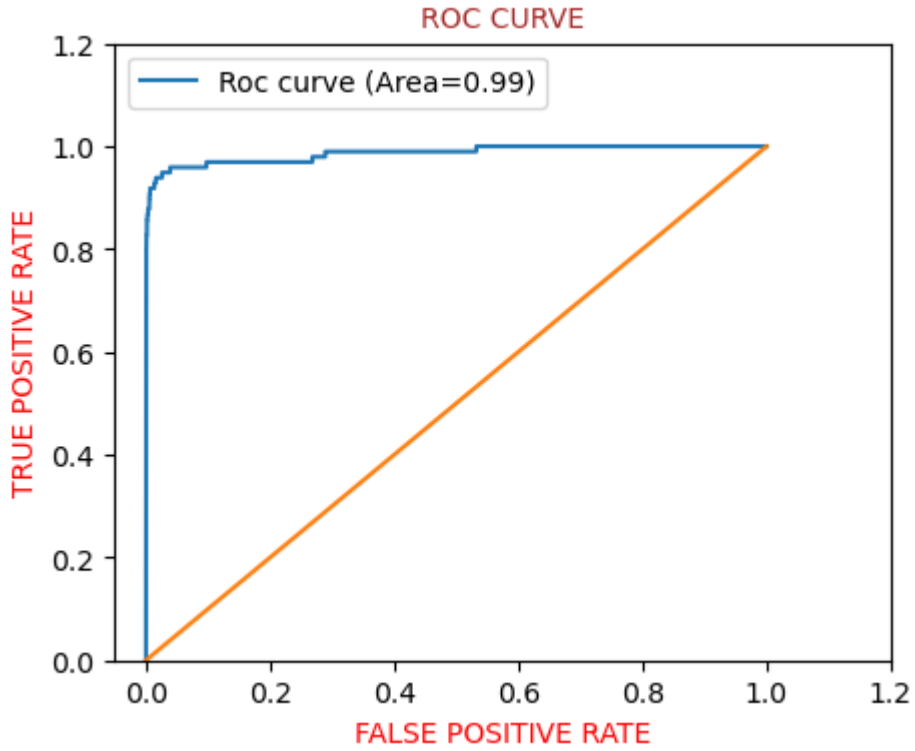
In [182…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.9489795918367347
The Specificity  is : 0.9710713280810355
```

In [183…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_logistic)
print('The Accuracy of Logistic Regression For Test is :',accuracy)
```

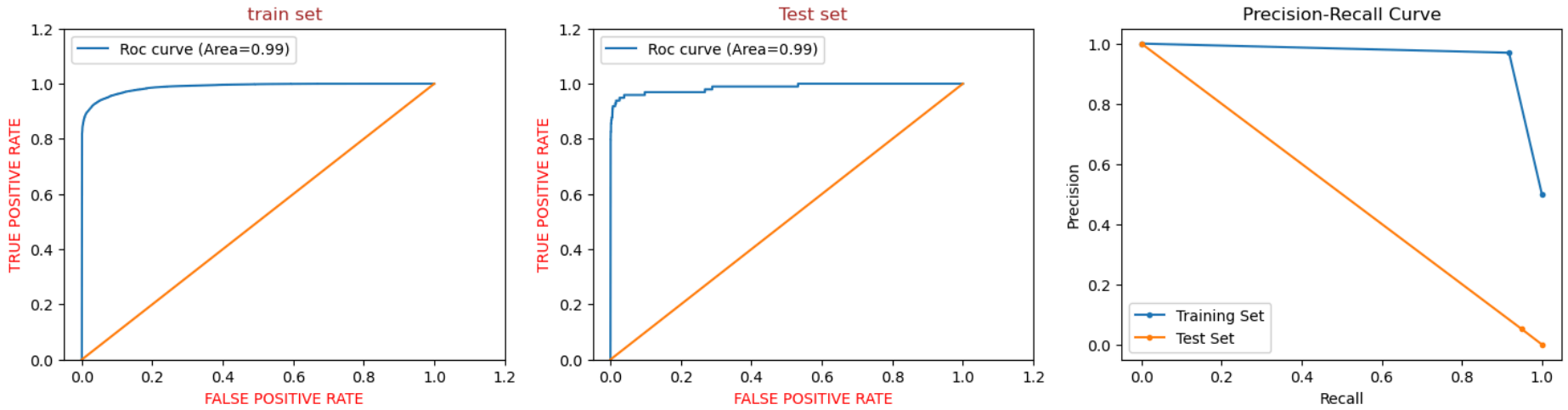
```
The Accuracy of Logistic Regression For Test is : 0.9710333204592535
```

In [184…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_logistic_proba)
```



In [185…
```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_smote,y_train_pred_logistic_proba_smote)
plt.title('train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_logistic_proba)
plt.title('Test set')
```

```
plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_smote, y_train_pred_logistic_smote)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_logistic)



# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```



Logistic Regression Performance (SMOTE Oversampling)

Metric	Training Set	Test Set
Accuracy	**94.46%**	**97.10%**

Metric	Training Set	Test Set
Sensitivity	91.79%	94.90%
Specificity	97.13%	97.11%
F1-score	94.31%	94.31%

XGBoost By SMOTE

In [186…

```
# Record the start time
start_time = time.time()

# Create the extended parameter grid for XGBoost
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees
    'learning_rate': [0.01, 0.1, 0.2],  # Step size shrinkage
    'max_depth': [3, 4, 5],  # Maximum depth of a tree
    'subsample': [0.8, 0.9, 1.0],  # Fraction of samples used for fitting the individual base learner
}

# Create K-fold cross-validation
folds = KFold(n_splits=3, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for XGBoost
model_cv = GridSearchCV(estimator=XGBClassifier(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=-1)
model_cv.fit(X_train_smote, y_train_smote)


# Record the end time
end_time = time.time()
# Calculate the elapsed time in seconds
elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60

print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 81 candidates, totalling 243 fits
Elapsed Time (Minutes): 20.71
```

In [187…
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[187]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_max_depth	param_n_estimators	param_subsample	
0	20.037454	0.063829	0.679519	0.007568	0.01	3	100	0.8	{'learnin… 'max_de…
1	20.124590	0.073133	0.677025	0.015218	0.01	3	100	0.9	{'learnin… 'max_de…
2	15.025690	0.408763	0.703676	0.005973	0.01	3	100	1.0	{'learnin… 'max_de…

In [188…
```
cv_results.columns
```

Out[188]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_learning_rate', 'param_max_depth', 'param_n_estimators',
       'param_subsample', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'mean_test_score', 'std_test_score',
       'rank_test_score'],
      dtype='object')
```

In [189…
```
cv_results[['param_learning_rate', 'param_subsample','rank_test_score','mean_test_score']].sort_values(by= 'rank_test_score' ,asc
```

Out[189]:

	param_learning_rate	param_subsample	rank_test_score	mean_test_score
80	0.2	1.0	1	1.0
79	0.2	0.9	1	1.0
51	0.1	0.8	1	1.0
69	0.2	0.8	1	1.0
52	0.1	0.9	1	1.0

In [190…
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 300, 'subsample': 0.8}
The Best score Is 1.0
```

Choosing Best Parameters For `XGBoost` By Smote

In [191…
```
# chosen hyperparameters
# 'objective':'binary:logistic' which outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': model_cv.best_params_['learning_rate'],
          'max_depth': model_cv.best_params_['max_depth'],
          'n_estimators': model_cv.best_params_['n_estimators'],
          'subsample':model_cv.best_params_['subsample'],
         'objective':'binary:logistic'}
```

In [192…
```
# fit model on training data
xgb_smote= XGBClassifier(params = params)
xgb_smote.fit(X_train_smote, y_train_smote)
```

Out[192]:

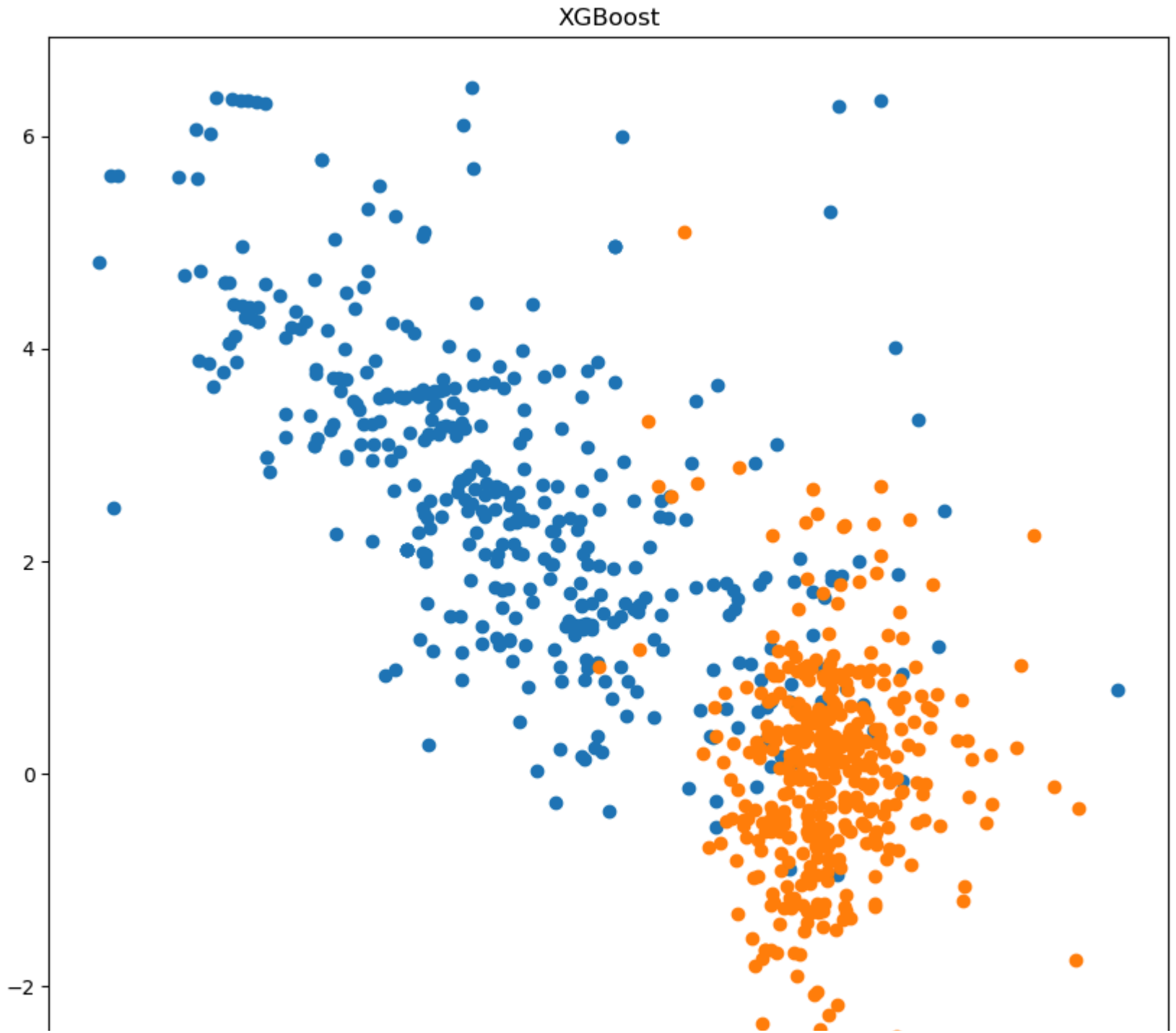
XGBClassifier

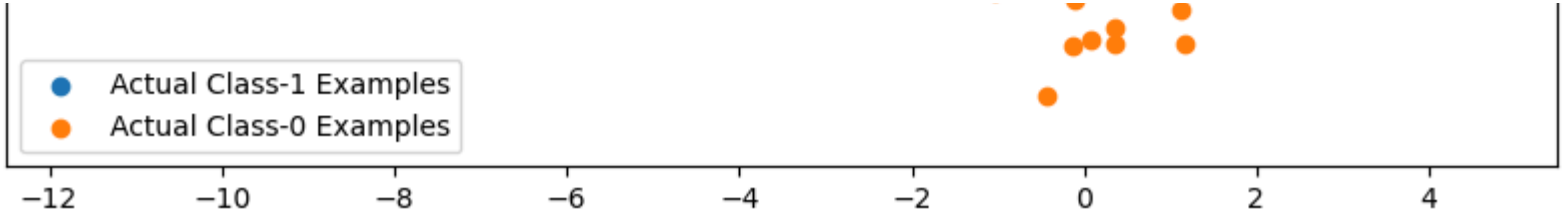
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, 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,
```

In [193…
```
var_imp = []
for i in xgb_smote.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(xgb_smote.feature_importances_)[-1])+1)
```

```
print('2nd Top var =', var_imp.index(np.sort(xgb_smote.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(xgb_smote.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_smote.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(xgb_smote.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'] = [10, 10]

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.title('XGBoost')

plt.legend()
plt.show()
```

```
Top var = 15
2nd Top var = 5
3rd Top var = 9
```


XGBoost
6
4
2
0
−2



In [194…
```
y_train_pred_xgboost_smote=xgb_smote.predict(X_train_smote)
```

Confusion Metrix For XGBoost SMOTE `Train Case`

In [195…
```
confusion_matrix_xgboost_train_smote=metrics.confusion_matrix(y_train_smote,y_train_pred_xgboost_smote)
```

In [196…
```
confusion_matrix_xgboost_train_smote
```

Out[196]:
```
array([[227450,      1],
       [     0, 227451]], dtype=int64)
```

In [197…
```
TN = confusion_matrix_xgboost_train_smote[0,0] # True negative
FP = confusion_matrix_xgboost_train_smote[0,1] # False positive
FN = confusion_matrix_xgboost_train_smote[1,0] # False negative
TP = confusion_matrix_xgboost_train_smote[1,1] # True positive
```

In [198…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 1.0
The Specificity  is : 0.9999956034486549
```

In [199…
```
print('The Accurays For The Train Set Of Xgboost Smote is ',metrics.accuracy_score(y_train_smote,y_train_pred_xgboost_smote))
print('The F1-Score For The Train Set Of Xgboost Smote is ',metrics.f1_score(y_train_smote,y_train_pred_xgboost_smote))
```

```
The Accurays For The Train Set Of Xgboost Smote is  0.9999978017243274
The F1-Score For The Train Set Of Xgboost Smote is  0.9999978017291599
```

In [200…
```
# classificatino report
print(classification_report(y_train_smote,y_train_pred_xgboost_smote))
```

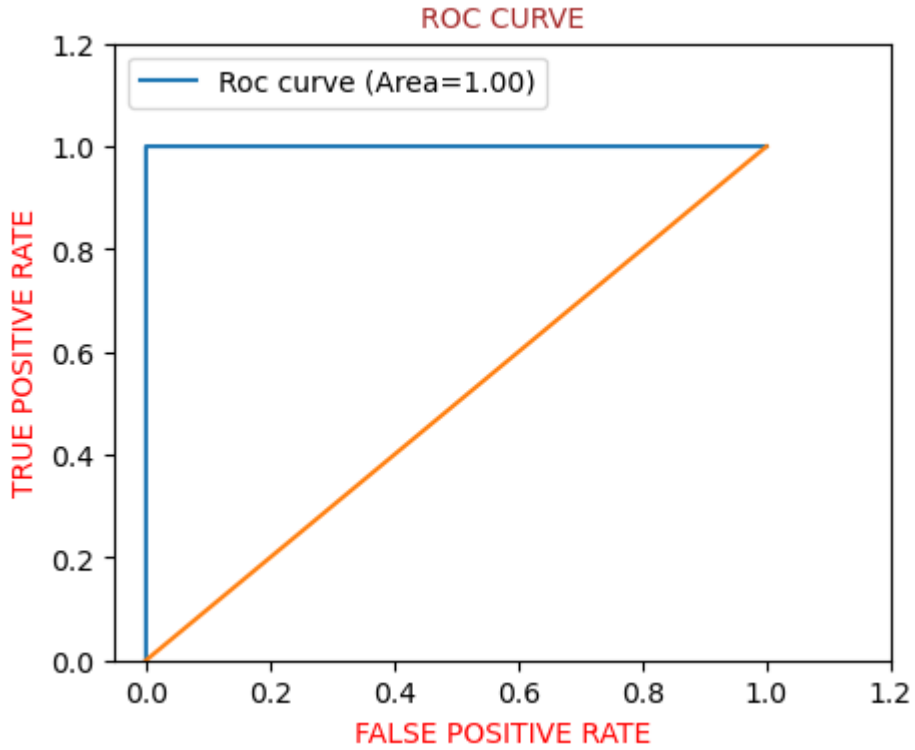
	precision	recall	f1-score	support
0	1.00	1.00	1.00	227451
1	1.00	1.00	1.00	227451
accuracy			1.00	454902
macro avg	1.00	1.00	1.00	454902
weighted avg	1.00	1.00	1.00	454902

In [ ]:

In [201…
```
# predicted probability
y_train_pred_xgboost_proba_smote=xgb_smote.predict_proba(X_train_smote)[:,1]
```

`ROC_AUC` Curve on Training set For `XGBoost SMOTE`

In [202…
```
plt.figure(figsize=(5,4))

draw_roc_curve(y_train_smote,y_train_pred_xgboost_proba_smote)
```



Let's Do Predictions On The `Test Set`

In [203…
```
y_test_pred_xgboost=xgb_smote.predict(X_test)
```

In [204…
```
y_test_pred_xgboost_proba=xgb_smote.predict_proba(X_test)[:,1]
```

Confusion Metrix For XGBoost `Test Case`

In [205…
```
confusion_matrix_xgboost_test=confusion_matrix(y_test,y_test_pred_xgboost)
confusion_matrix_xgboost_test
```

Out[205]:
```
array([[56832,    32],
       [   14,    84]], dtype=int64)
```

In [206…
```
TN = confusion_matrix_xgboost_test[0,0] # True negative
FP = confusion_matrix_xgboost_test[0,1] # False positive
```

```
FN = confusion_matrix_xgboost_test[1,0] # False negative
TP = confusion_matrix_xgboost_test[1,1] # True positive
```

In [207…

```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8571428571428571
The Specificity  is : 0.9994372537985369
```

In [208…

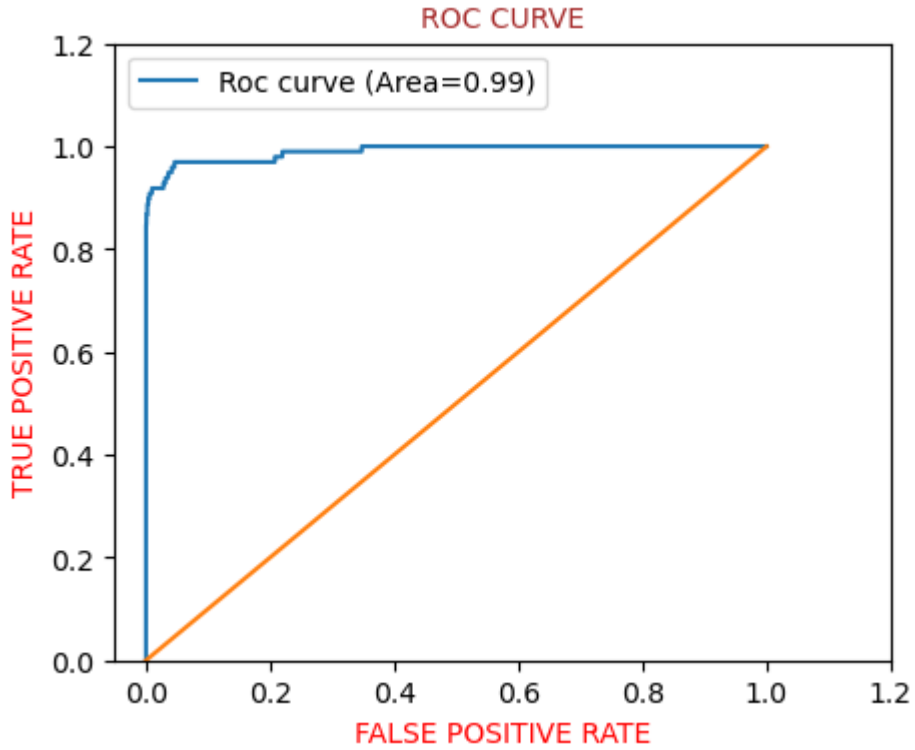
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_xgboost)
print('The Accuracy of XG Boost For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_xgboost)
print("The F1-score of XG Boost For Test is :", F1_score)
```

```
The Accuracy of XG Boost For Test is : 0.9991924440855307
The F1-score of XG Boost For Test is : 0.7850467289719626
```

`ROC_AUC` Curve on Test set For `XGBoost`

In [209…

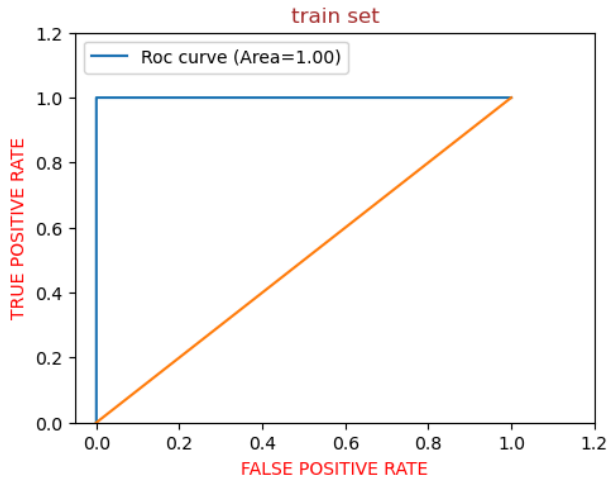
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
```

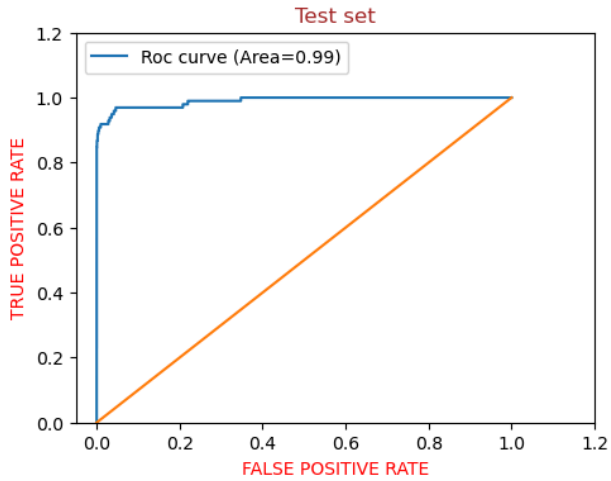


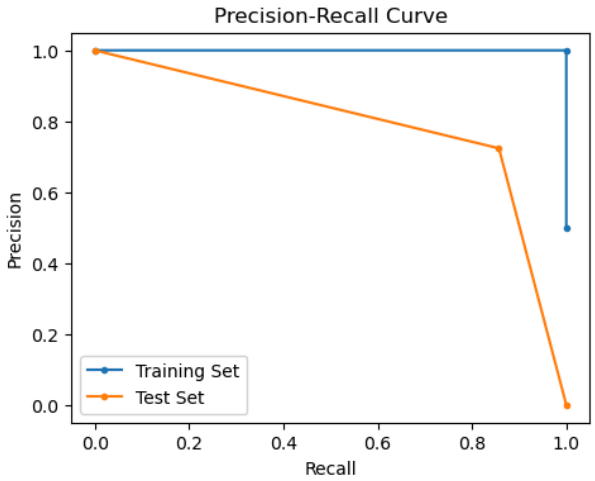
In [210…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_smote,y_train_pred_xgboost_proba_smote)
plt.title('train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
plt.title('Test set')


plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_smote, y_train_pred_xgboost_smote)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_xgboost)
```

```
# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```







XGBoost with SMOTE Performance

Metric	Training Set	Test Set
Accuracy	**0.9999978**	**0.9991924**
Sensitivity	**1.0**	**0.8571**
Specificity	**0.9999956**	**0.9994373**
F1-score	**0.9999978**	**0.7850**

Decision Tree By SMOTE

In [211…

```
# Record the start time
start_time = time.time()



param_grid = {
    'max_depth': [3, 5, 7],  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required to be at a leaf node
    'criterion': ['gini', 'entropy']  # Function to measure the quality of a split
}


# Create K-fold cross-validation
folds = KFold(n_splits=3, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for Decision Tree
model_cv = GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=
model_cv.fit(X_train_smote, y_train_smote)



# Record the end time
end_time = time.time()
# Calculate the elapsed time in seconds
elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60

print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 54 candidates, totalling 162 fits
Elapsed Time (Minutes): 9.74
```

In [212…

```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[212]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_leaf	param_min_samples_split	
0	14.454076	0.099332	0.260531	0.019184	gini	3	1	2	'r
1	14.172156	0.089978	0.222484	0.011341	gini	3	1	5	'r
2	14.089024	0.212746	0.223189	0.018509	gini	3	1	10	'r

In [213…
```
cv_results.columns
```

Out[213]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_criterion', 'param_max_depth', 'param_min_samples_leaf',
       'param_min_samples_split', 'params', 'split0_test_score',
       'split1_test_score', 'split2_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```

In [214…
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'criterion': 'gini', 'max_depth': 7, 'min_samples_leaf': 1, 'min_samples_split': 5}
The Best score Is 0.9705783256944002
```

Choosing Best Parameters For `Decision Trees` Using Smote

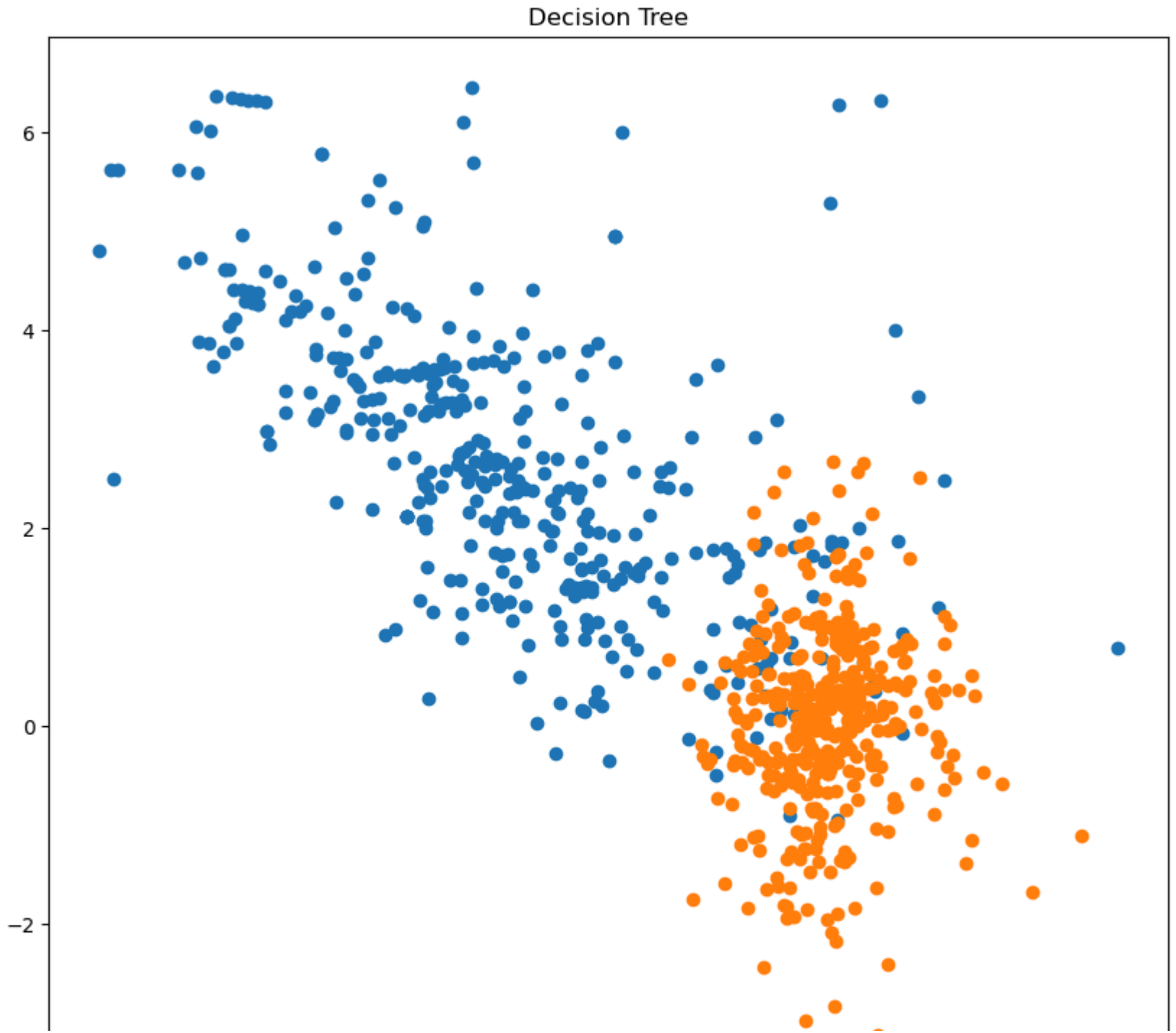
In [215…
```
dtree_smote=DecisionTreeClassifier(random_state=100,
                                   criterion=model_cv.best_params_['criterion'],
                                   max_depth=model_cv.best_params_['max_depth'],
                                   min_samples_leaf=model_cv.best_params_['min_samples_leaf'],
                                   min_samples_split=model_cv.best_params_['min_samples_split'])
```

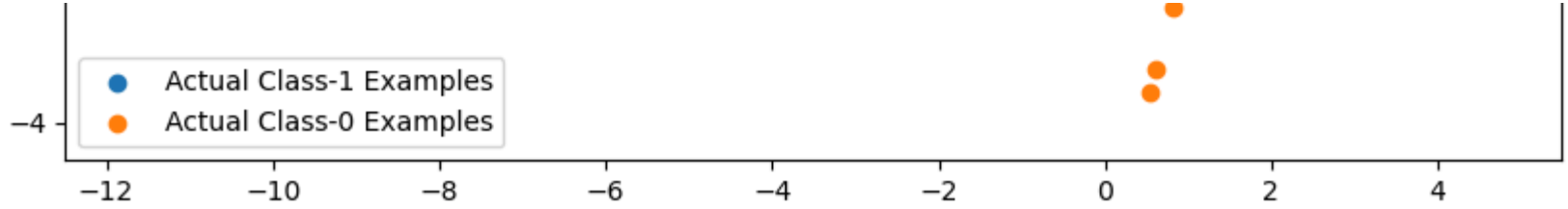
In [216…
```
dtree_smote.fit(X_train_smote,y_train_smote)
```

Out[216]:
```
DecisionTreeClassifier(max_depth=7, min_samples_split=5, random_state=100)
```

In [217…
```
var_imp = []
for i in dtree_smote.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(dtree_smote.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(dtree_smote.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(dtree_smote.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(dtree_smote.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(dtree_smote.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'] = [10, 10]

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.title('Decision Tree')
plt.legend()
plt.show()
```

```
Top var = 15
2nd Top var = 5
3rd Top var = 13
```


Decision Tree
6
4
2
0
−2



In [218…
```
y_train_pred_dtree_smote=dtree_smote.predict(X_train_smote)
```

In [219…
```
confusion_matrix_dtree_train_smote=confusion_matrix(y_train_smote,y_train_pred_dtree_smote)
```

In [220…
```
confusion_matrix_dtree_train_smote
```

Out[220]:
```
array([[220095,   7356],
       [  6204, 221247]], dtype=int64)
```

In [221…
```
TN = confusion_matrix_dtree_train_smote[0,0] # True negative
FP = confusion_matrix_dtree_train_smote[0,1] # False positive
FN = confusion_matrix_dtree_train_smote[1,0] # False negative
TP = confusion_matrix_dtree_train_smote[1,1] # True positive
```

In [222…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.9727237954548452
The Specificity  is : 0.9676589683052613
```

In [223…
```
accuracy=metrics.accuracy_score(y_train,y_train_pred_dtree)
print('The Accuracy of Decision Trees For Train is :',accuracy)

F1_score=metrics.f1_score(y_train,y_train_pred_dtree)
print("The F1-score of  Decision Trees For Train is :", F1_score)
```

```
The Accuracy of Decision Trees For Train is : 0.9994118808839343
The F1-score of  Decision Trees For Train is : 0.8164383561643835
```

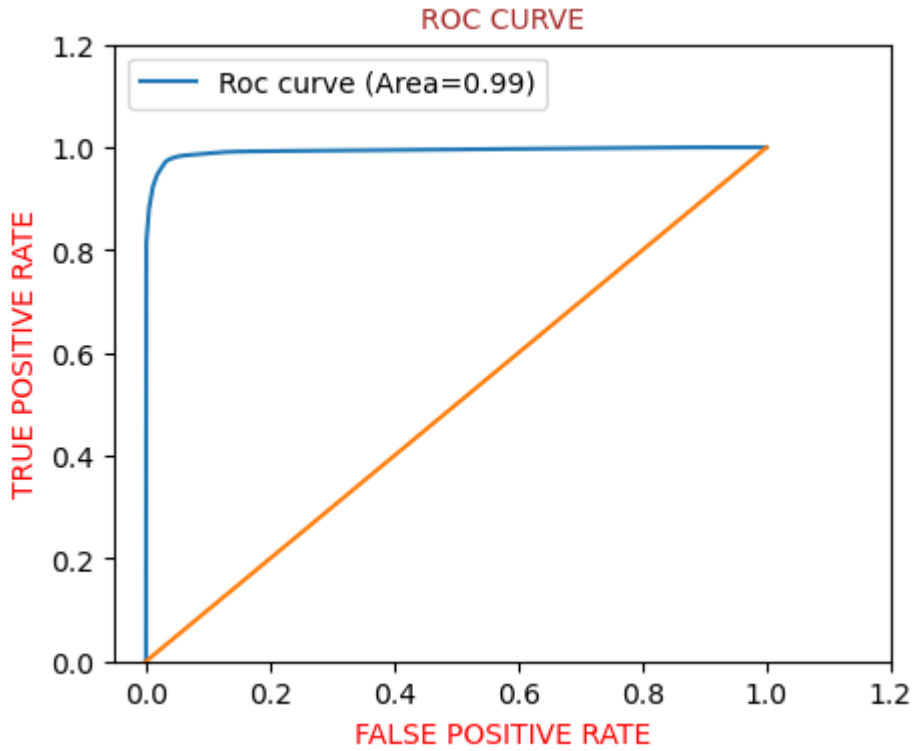
In [224…
```
# classification_report
print(classification_report(y_train, y_train_pred_dtree))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	227451
1	0.89	0.76	0.82	394
accuracy			1.00	227845
macro avg	0.94	0.88	0.91	227845
weighted avg	1.00	1.00	1.00	227845

`ROC_AUC` Curve on Train set For `Decision Tree` Using Smote

In [225…
```
y_train_pred_dtree_proba_smote=dtree_smote.predict_proba(X_train_smote)[:,1]
```

In [226…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train_smote,y_train_pred_dtree_proba_smote)
```



Let's Do Predictions On The `Test Set`

In [227…
```
y_test_pred_dtree=dtree_smote.predict(X_test)
```

In [228…
```
y_test_pred_dtree_proba=dtree_smote.predict_proba(X_test)[:,1]
```

Confusion Metrix For Decision `Test Case`

In [229…
```
confusion_matrix_dtree_test=confusion_matrix(y_test,y_test_pred_dtree)
confusion_matrix_dtree_test
```

Out[229]:
```
array([[54805,  2059],
       [   12,    86]], dtype=int64)
```

In [230…
```
TN = confusion_matrix_dtree_test[0,0] # True negative
FP = confusion_matrix_dtree_test[0,1] # False positive
FN = confusion_matrix_dtree_test[1,0] # False negative
TP = confusion_matrix_dtree_test[1,1] # True positive
```

In [231…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8775510204081632
The Specificity  is : 0.9637907990996061
```

In [232…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_dtree)
print('The Accuracy of Decision Tree For Test is :',accuracy)
```

```
The Accuracy of Decision Tree For Test is : 0.9636424282855237
```

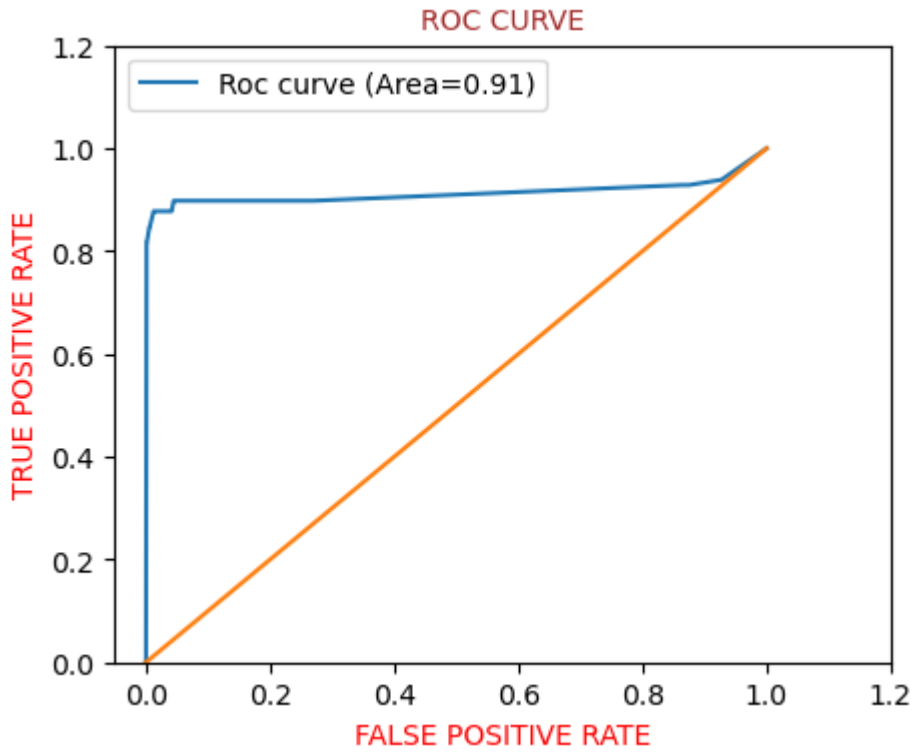
In [233…
```
# classification_report
print(classification_report(y_test, y_test_pred_dtree))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     56864
           1       0.04      0.88      0.08        98

    accuracy                           0.96     56962
   macro avg       0.52      0.92      0.53     56962
weighted avg       1.00      0.96      0.98     56962
```

`ROC_AUC` Curve on Test set For `Decision Tree`

In [234…

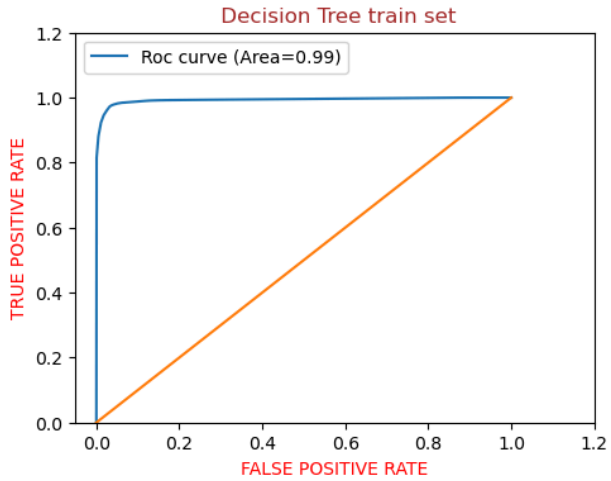
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_dtree_proba)
```

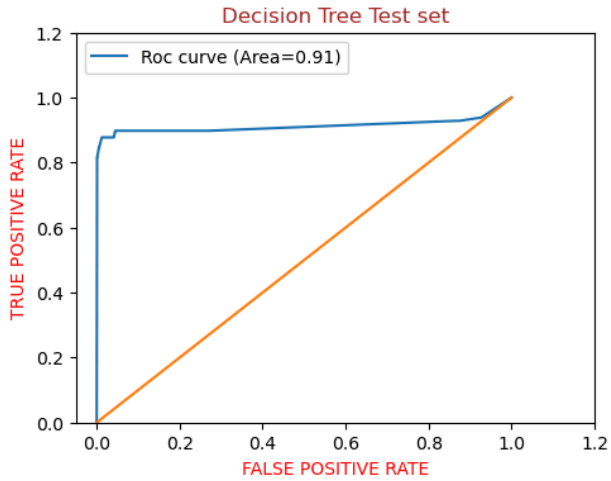


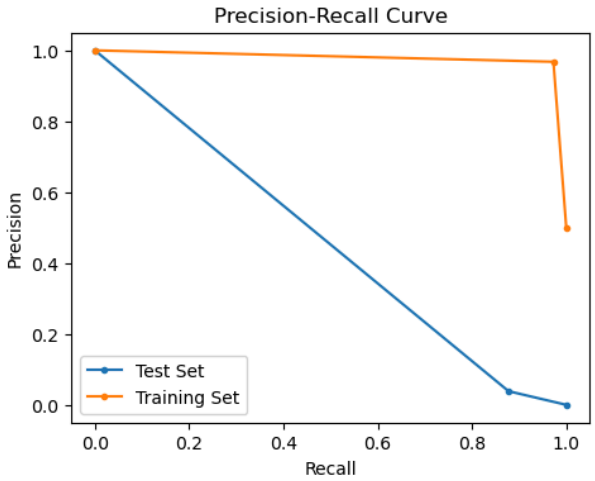
In [235…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_smote,y_train_pred_dtree_proba_smote)

plt.title(' Decision Tree train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_dtree_proba)

plt.title('Decision Tree Test set')

plt.subplot(1,3,3)
```

```
# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_dtree)
# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_smote, y_train_pred_dtree_smote)




# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_test, precision_test, label='Test Set', marker='.')

plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()


plt.show()
```







Decision Tree Performance Evaluation (SMOTE)

Metric	Train Set	Test Set
Accuracy	99.94%	96.36%
Sensitivity	97.27%	87.76%
Specificity	96.77%	96.38%
F1-Score	81.64%	8.00%

Random Foreset By SMOTE

As I Observed That `Random Forest`, Especially With A `Large Number Of Trees And Deep Trees`, Can Indeed Be `Computationally Expensive`, And Applying SMOTE May Further `Increase The Training Time` Due To The `Generation Of Synthetic` Samples.

Using `SMOTE` (Synthetic Minority Over-Sampling Technique) `Increases` The Number Of `Minority Class Samples` By Generating `Synthetic Examples`, Making The Dataset More Balanced. While This Can Be Beneficial For Improving The Performance Of Models, It Comes With A Computational Cost, Especially For Algorithms That Build A Large Number Of Models Or Trees, Such As Random Forests.

1. **Increased Dataset Size**
2. **Complexity of the Model**
3. **Tree Building Process**
4. **Hyperparameter Tuning**
5. **Parallelization**

In [237…

```
# # Record the start time
# start_time = time.time()

# param_grid = {
#     'n_estimators': [100, 200],
#     'max_depth': [None, 10, 20],
#     'min_samples_split': [2, 5],
#     'min_samples_leaf': [1, 2],
#     'max_features': ['auto', 'sqrt'],
```

```
# }
# random_forest=RandomForestClassifier()

# model_cv = GridSearchCV(estimator=random_forest,
#                         param_grid=param_grid,
#                         scoring='roc_auc',
#                         cv=3,
#                         verbose=2,
#                         n_jobs=-1,
#                        return_train_score=True)



# model_cv .fit(X_train_smote,y_train_smote)




# # Record the end time
# end_time = time.time()
# # Calculate the elapsed time in seconds
# elapsed_time_seconds = end_time - start_time

# # Convert elapsed time to minutes
# elapsed_time_minutes = elapsed_time_seconds / 60

# print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits
Elapsed Time (Minutes): 191.89
```

- **Fitting 3 folds for each of 48 candidates, totalling 144 fits**
- **Elapsed Time (Minutes): 191.89**

In [239…

```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}
The Best score Is 0.9999985975072536
```

- **The Best params Is {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}**

- **The Best score Is 0.9999985975072536**

Choosing Best Parameters For `Random Forest` Using Smote

In [240…
```
random_forest_smote=RandomForestClassifier(random_state=100,
                                           criterion='gini',
                                           max_depth=None,
                                           min_samples_leaf=1,
                                           min_samples_split=5,
                                           n_estimators=100,
                                           max_features='sqrt')
```

In [241…
```
random_forest_smote.fit(X_train_smote,y_train_smote)
```

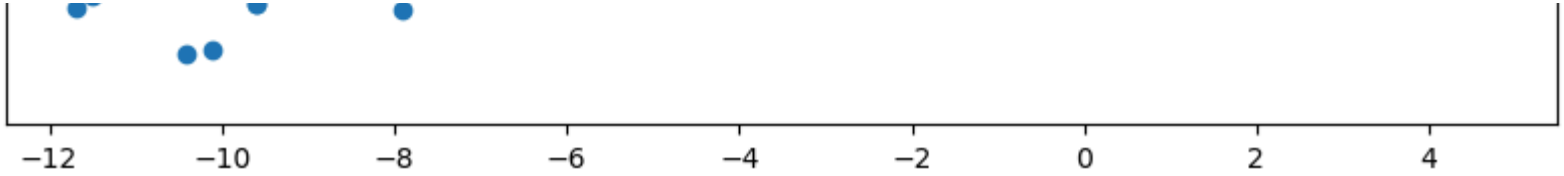
Out[241]:
```
RandomForestClassifier(min_samples_split=5, random_state=100)
```

In [242…
```
var_imp = []
for i in random_forest_smote.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(random_forest_smote.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(random_forest_smote.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(random_forest_smote.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(random_forest_smote.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(random_forest_smote.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'] = [10, 10]

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()
plt.show()
```

```
Top var = 15
2nd Top var = 13
3rd Top var = 12
```


Actual Class-1 Examples
Actual Class-0 Examples
2
0
−2
−4
−6



In [243…
```
y_train_pred_random_forest_smote=random_forest_smote.predict(X_train_smote)
```

In [244…
```
confusion_matrix_random_forest_train_smote=confusion_matrix(y_train_smote,y_train_pred_random_forest_smote)
```

In [245…
```
confusion_matrix_random_forest_train_smote
```

Out[245]:
```
array([[227451,      0],
       [     0, 227451]], dtype=int64)
```

In [246…
```
TN = confusion_matrix_random_forest_train_smote[0,0] # True negative
FP = confusion_matrix_random_forest_train_smote[0,1] # False positive
FN = confusion_matrix_random_forest_train_smote[1,0] # False negative
TP = confusion_matrix_random_forest_train_smote[1,1] # True positive
```

In [247…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 1.0
The Specificity  is : 1.0
```

In [248…
```
accuracy=metrics.accuracy_score(y_train_smote,y_train_pred_random_forest_smote)
print('The Accuracy of random_forest For Train is :',accuracy)

F1_score=metrics.f1_score(y_train_smote,y_train_pred_random_forest_smote)
print("The F1-score of  random_forest For Train is :", F1_score)
```

```
The Accuracy of random_forest For Train is : 1.0
The F1-score of  random_forest For Train is : 1.0
```

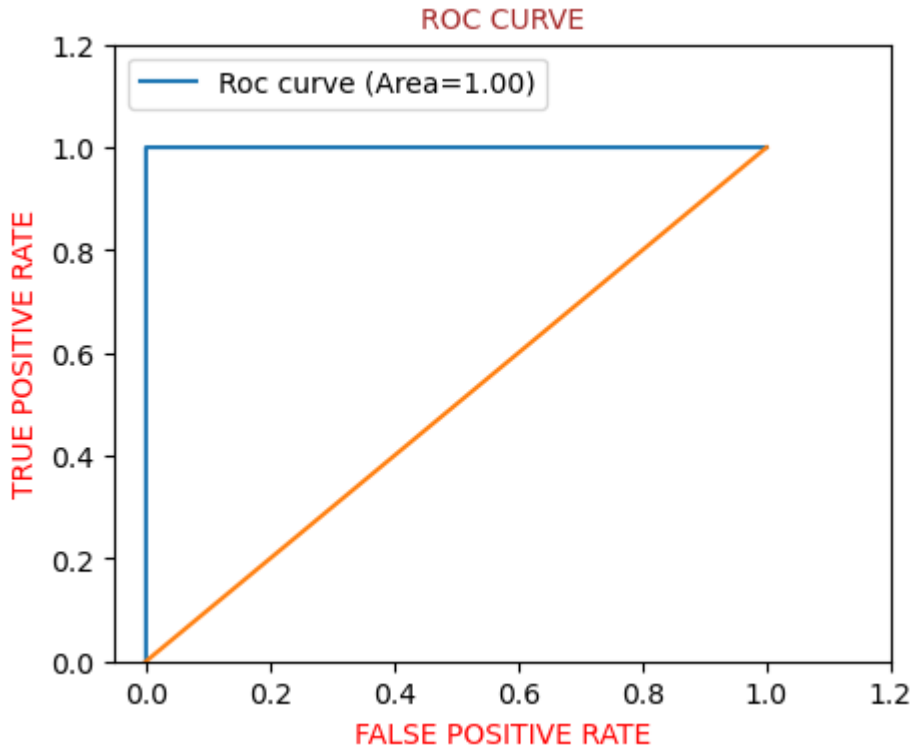
In [249…
```
# classification_report
print(classification_report(y_train_smote,y_train_pred_random_forest_smote))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	227451
1	1.00	1.00	1.00	227451
accuracy			1.00	454902
macro avg	1.00	1.00	1.00	454902
weighted avg	1.00	1.00	1.00	454902

`ROC_AUC` Curve on Train set For `Random Forest`

In [250…
```
y_train_pred_random_forest_proba_smote = random_forest_smote.predict_proba(X_train_smote)[:,1]
```

In [251…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train_smote,y_train_pred_random_forest_proba_smote)
```



Let's Do Predictions On The `Test Set`

In [252…
```
y_test_pred_random_forest=random_forest_smote.predict(X_test)
```

In [253…
```
y_test_pred_random_forest_proba=random_forest_smote.predict_proba(X_test)[:,1]
```

Confusion Metrix For Random Forest `Test Case`

In [254…
```
confusion_matrix_random_forest_test=confusion_matrix(y_test,y_test_pred_random_forest)
confusion_matrix_random_forest_test
```

Out[254]:
```
array([[56855,     9],
       [   16,    82]], dtype=int64)
```

In [255…
```
TN = confusion_matrix_random_forest_test[0,0] # True negative
FP = confusion_matrix_random_forest_test[0,1] # False positive
FN = confusion_matrix_random_forest_test[1,0] # False negative
TP = confusion_matrix_random_forest_test[1,1] # True positive
```

In [256…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8367346938775511
The Specificity  is : 0.9998417276308385
```

In [257…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_random_forest)
print('The Accuracy of random_forest For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_random_forest)
print("The F1-score of random_forest For Test is :", F1_score)
```

```
The Accuracy of random_forest For Test is : 0.9995611109160493
The F1-score of random_forest For Test is : 0.8677248677248677
```

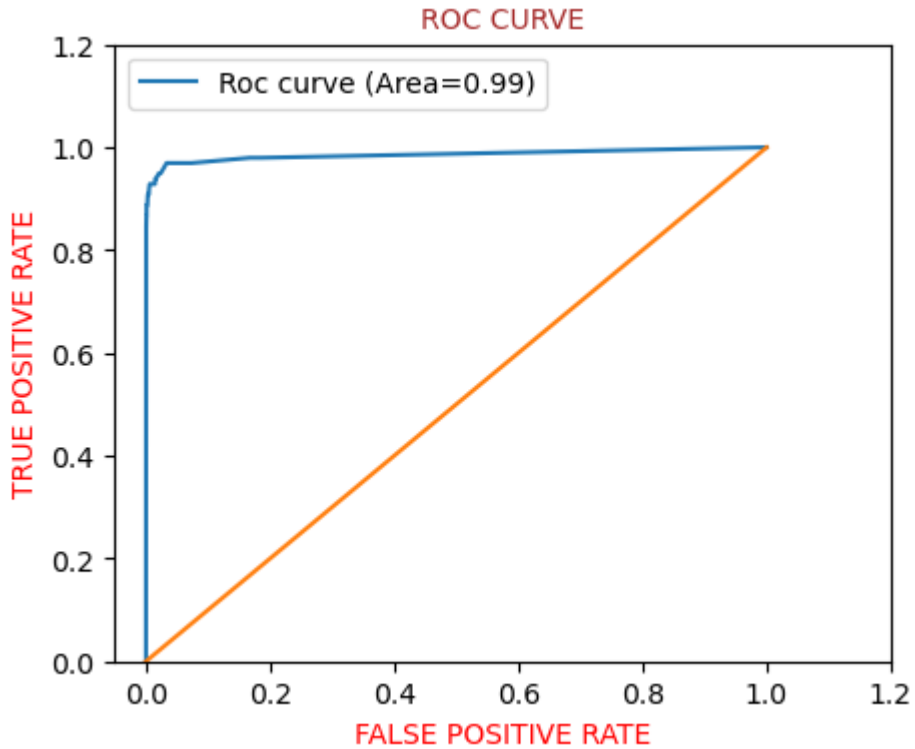
In [258…
```
# classification_report
print(classification_report(y_test, y_test_pred_random_forest))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.90	0.84	0.87	98
accuracy			1.00	56962
macro avg	0.95	0.92	0.93	56962
weighted avg	1.00	1.00	1.00	56962

`ROC_AUC` Curve on Test set For `Random Forest`

In [259…

```
plt.figure(figsize=(5,4))

draw_roc_curve(y_test,y_test_pred_random_forest_proba)
```



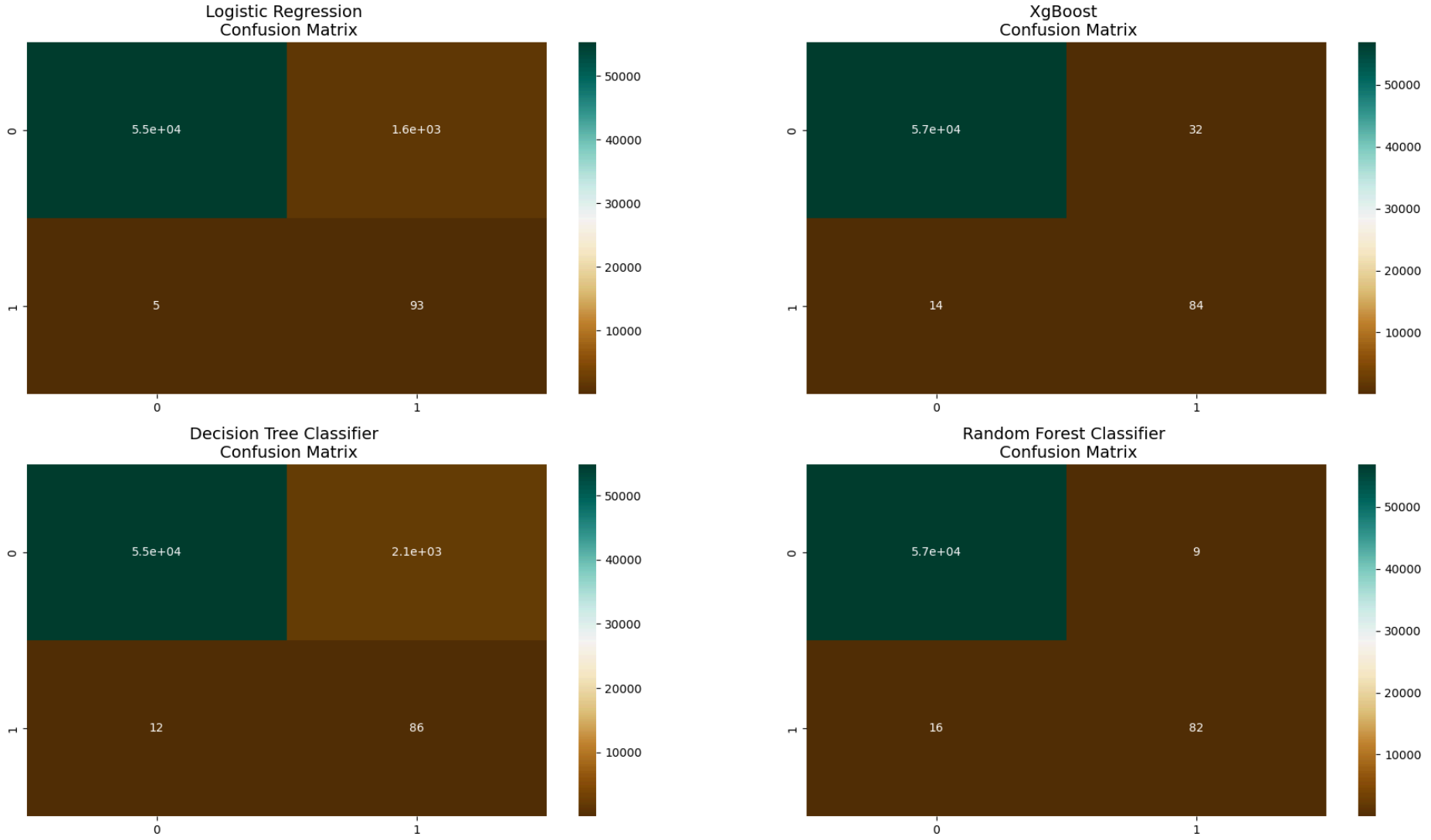
Performance Metrics for Random Forest with SMOTE

Metric	Train Set	Test Set
Accuracy	1.00	0.99956
Sensitivity	1.00	0.8367
Specificity	1.00	0.99984
F1-Score	1.00	0.8677

In [260…

```
from sklearn.metrics import confusion_matrix

# all models fitted using SMOTE technique
y_test_pred_logistic = logistic_smote.predict(X_test)

y_test_pred_xgboost = xgb_smote.predict(X_test)
y_test_pred_dtree = dtree_smote.predict(X_test)
y_test_pred_random_forest = random_forest_smote.predict(X_test)


log_reg_cf = confusion_matrix(y_test, y_test_pred_logistic)
xgboost_cf = confusion_matrix(y_test, y_test_pred_xgboost)
dtree_cf = confusion_matrix(y_test, y_test_pred_dtree)
random_forest_cf = confusion_matrix(y_test, y_test_pred_random_forest)

fig, ax = plt.subplots(2, 2,figsize=(22,12))


sns.heatmap(log_reg_cf, ax=ax[0][0], annot=True, cmap=plt.cm.BrBG)
ax[0, 0].set_title("Logistic Regression \n Confusion Matrix", fontsize=14)
# ax[0, 0].set_xticklabels(['', ''], fontsize=14, rotation=90)
# ax[0, 0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(xgboost_cf, ax=ax[0][1], annot=True, cmap=plt.cm.BrBG)
ax[0][1].set_title("XgBoost \n Confusion Matrix", fontsize=14)


sns.heatmap(dtree_cf, ax=ax[1][0], annot=True, cmap=plt.cm.BrBG)
ax[1][0].set_title("Decision Tree Classifier \n Confusion Matrix", fontsize=14)


sns.heatmap(random_forest_cf, ax=ax[1][1], annot=True, cmap=plt.cm.BrBG)
ax[1][1].set_title("Random Forest Classifier \n Confusion Matrix", fontsize=14)
```

```
plt.show()
```



Adaptive Synthetic Sampling

- **`ADASYN`, which stands for `Adaptive Synthetic Sampling`, is an `oversampling technique` used in machine learning, particularly in the `context of imbalanced datasets`. It is designed to address the challenge of `class imbalance` by generating synthetic samples `for the minority class`. What sets ADASYN apart is its `adaptive nature` – it focuses on `generating more synthetic` examples for the minority class instances that are harder to classify, aiming to alleviate the impact of class imbalance on model performance.**
- **Adaptive Synthetic Sampling (ADASYN) is a technique used to address `imbalanced datasets in machine learning`, `improving classification performance` for underrepresented classes.**
- **`Adasyn` : Similar To Smote, It Adapts By Introducing Synthetic Samples Based On A Density Distribution, Focusing On Harder-To-Learn Minority Examples.**

In [261…
```
# importing adasyn
from imblearn.over_sampling import ADASYN
```

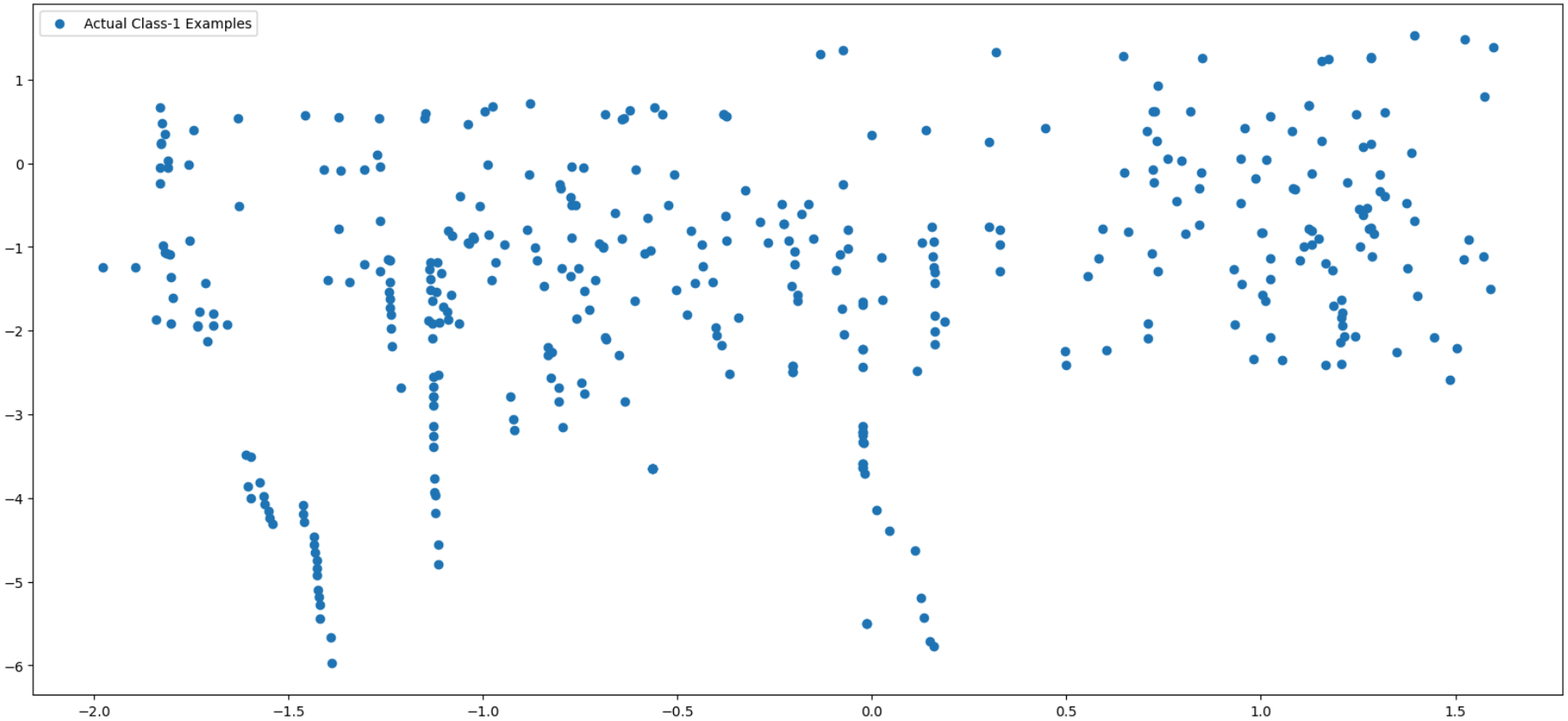
In [262…
```
# Instantiate adasyn
ada = ADASYN(random_state=0)
X_train_adasyn, y_train_adasyn = ada.fit_resample(X_train, y_train)
```

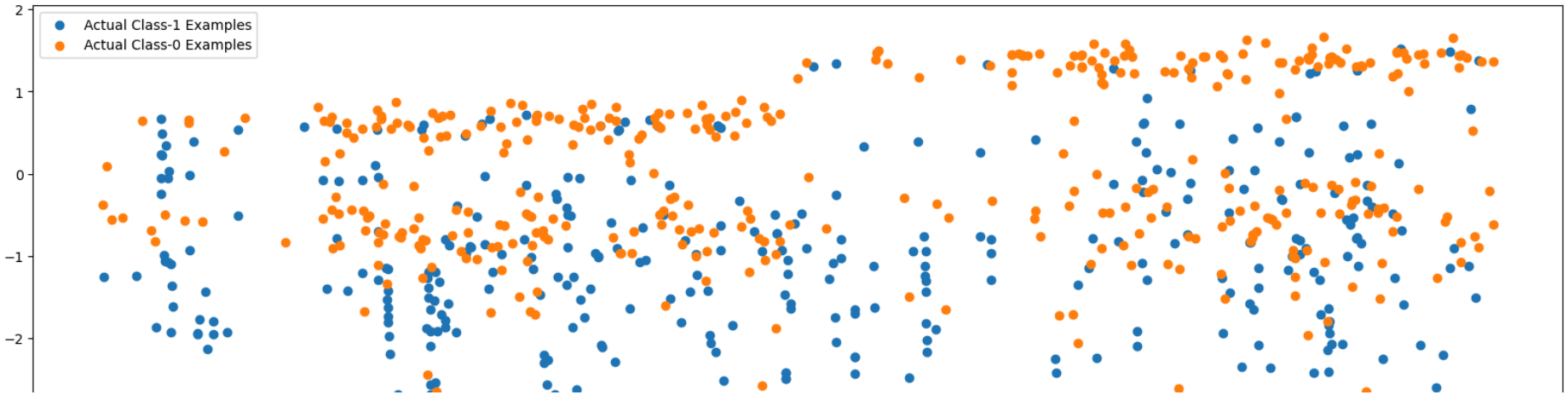
In [263…
```
from collections import Counter

# Before sampling class distribution
print('Before sampling class distribution:', Counter(y_train))

# New class distribution
print('New class distribution:', Counter(y_train_adasyn))
```

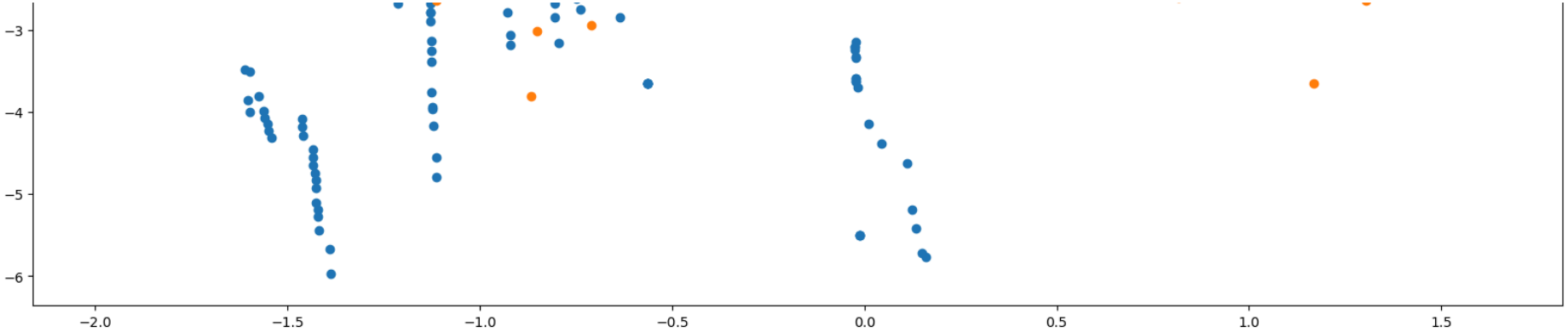
```
Before sampling class distribution: Counter({0: 227451, 1: 394})
New class distribution: Counter({1: 227470, 0: 227451})
```

In [264…
```
# Artificial minority samples and corresponding minority labels from ADASYN are appended
# below X_train and y_train respectively
# So to exclusively get the artificial minority samples from ADASYN, we do
X_train_adasyn_1 = X_train_adasyn[X_train.shape[0]:]

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)]


import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.rcParams['figure.figsize'] = [20, 20]
fig = plt.figure()

plt.subplot(2, 1, 1)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.legend()


plt.subplot(2, 1, 2)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.scatter(X_train_0[:X_train_1.shape[0], 0], X_train_0[:X_train_1.shape[0], 1], label='Actual Class-0 Examples')
plt.legend()
plt.show()
```


Actual Class-1 Examples
1
0
−1
−2
−3
−4
−5
−6
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Actual Class-1 Examples
Actual Class-0 Examples
2
1
0
−1
−2



Logistic Regression By Adasyn

In [265…
```
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10],  # Regularization parameter
    'penalty': ['l1', 'l2'],  # Regularization type
    'class_weight': ['balanced', None],  # Adjusts for class imbalance
}

# Create K-fold cross-validation
folds = KFold(n_splits=5, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for logistic regression
model_cv = GridSearchCV(estimator=LogisticRegression(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=-1)
model_cv.fit(X_train_adasyn, y_train_adasyn)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
```

Out[265]:

▸ GridSearchCV

▸ estimator: LogisticRegression

▸ LogisticRegression

In [266…
```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[266]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_class_weight	param_penalty	params	split0_test_score	split1_test
0	0.658465	0.076207	0.00000	0.000000	0.001	balanced	l1	{'C': 0.001, 'class_weight': 'balanced', 'pena...	NaN	
1	4.323888	0.331534	0.17485	0.015421	0.001	balanced	l2	{'C': 0.001, 'class_weight': 'balanced', 'pena...	0.846387	0.8
2	0.605328	0.119420	0.00000	0.000000	0.001	None	l1	{'C': 0.001, 'class_weight': None, 'penalty': ...	NaN	

In [267…
```
cv_results.columns
```

Out[267]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_C', 'param_class_weight', 'param_penalty', 'params',
       'split0_test_score', 'split1_test_score', 'split2_test_score',
       'split3_test_score', 'split4_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```

In [268…
```
# best score
best_score=model_cv.best_score_
# best params
best_params=model_cv.best_params_
```

In [269…
```
print(f"The Best Score Is {best_score}")
print(f"The Best Params Is {best_params}")
```

```
The Best Score Is 0.8649977191465584
The Best Params Is {'C': 10, 'class_weight': None, 'penalty': 'l2'}
```

Logistic Regression With Optimal Value That Is `c=10`

In [270…
```
logistic_=LogisticRegression(C=10,penalty='l2',class_weight=None)
```

In [271…
```
# fiitting the model on train set
logistic_adasyn=logistic_.fit(X_train_adasyn,y_train_adasyn)
```

In [272…
```
y_train_pred_logistic_adasyn=logistic_adasyn.predict(X_train_adasyn)
```

Confusion Metrix For Adasyn `Logistic Regression`

In [273…
```
confusion_matrix_logistic_train_adasyn=metrics.confusion_matrix(y_train_adasyn,y_train_pred_logistic_adasyn)
```

In [274…
```
confusion_matrix_logistic_train_adasyn
```

Out[274]:
```
array([[204587,  22864],
       [ 30710, 196760]], dtype=int64)
```

In [275…
```
TN = confusion_matrix_logistic_train_adasyn[0,0] # True negative
FP = confusion_matrix_logistic_train_adasyn[0,1] # False positive
FN = confusion_matrix_logistic_train_adasyn[1,0] # False negative
TP = confusion_matrix_logistic_train_adasyn[1,1] # True positive
```

In [276…
```
calculation_metrics(TN,FP,FN,TP)
```

```
The Sensitivity  is : 0.8649931859146262
The Specificity  is : 0.8994772500450646
```

In [277…
```
accuracy=metrics.accuracy_score(y_train_adasyn,y_train_pred_logistic_adasyn)
print('The Accuracy of Logistic Regression By Using Adasyn For Train is :',accuracy)

F1_score=metrics.f1_score(y_train_adasyn,y_train_pred_logistic_adasyn)
print("The F1-score of Logistic Regression By Using Adasyn For Train is :", F1_score)
```

```
The Accuracy of Logistic Regression By Using Adasyn For Train is : 0.8822344978578698
The F1-score of Logistic Regression By Using Adasyn For Train is : 0.8801728495573637
```

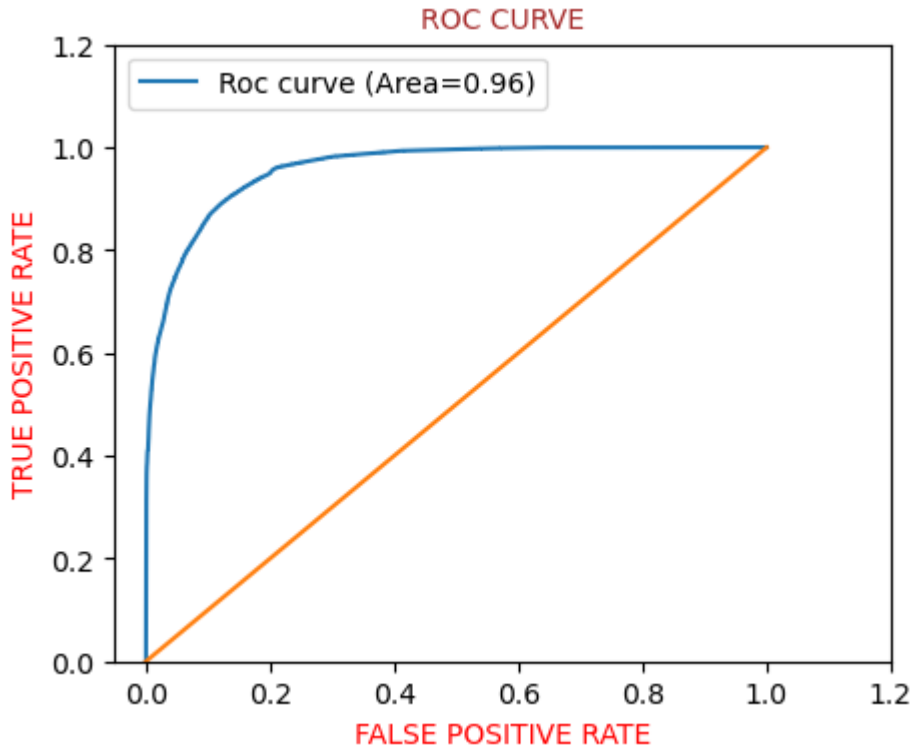
In [278…
```
# classificatino report
print(classification_report(y_train_adasyn,y_train_pred_logistic_adasyn))
```

	precision	recall	f1-score	support
0	0.87	0.90	0.88	227451
1	0.90	0.86	0.88	227470
accuracy			0.88	454921
macro avg	0.88	0.88	0.88	454921
weighted avg	0.88	0.88	0.88	454921

In [279…
```
# predicted probability
y_train_pred_logistic_proba_adasyn=logistic_adasyn.predict_proba(X_train_adasyn)[:,1]
```

`ROC_AUC` Curve on Training set For `Logistic Regression`

In [280…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train_adasyn,y_train_pred_logistic_proba_adasyn)
```



Avoid applying ADASYN or any oversampling technique to the test set. Modifying the test set with synthetic instances can distort its representation and lead to misleading assessments. Keep the test set untouched for fair evaluation, applying oversampling only to the training set to ensure the model's robustness in handling imbalanced data.

Let's Do Predictions On The `Test Set`

In [281…
```
y_test_pred_logistic=logistic_adasyn.predict(X_test)
```

In [282…
```
y_test_pred_logistic_proba=logistic_adasyn.predict_proba(X_test)[:,1]
```

In [283…
```
confusion_matrix_logistic_test=confusion_matrix(y_test,y_test_pred_logistic)
confusion_matrix_logistic_test
```

Out[283]:
```
array([[51023,  5841],
       [    4,    94]], dtype=int64)
```

In [284…
```
TN = confusion_matrix_logistic_test[0,0] # True negative
FP = confusion_matrix_logistic_test[0,1] # False positive
FN = confusion_matrix_logistic_test[1,0] # False negative
TP = confusion_matrix_logistic_test[1,1] # True positive
```

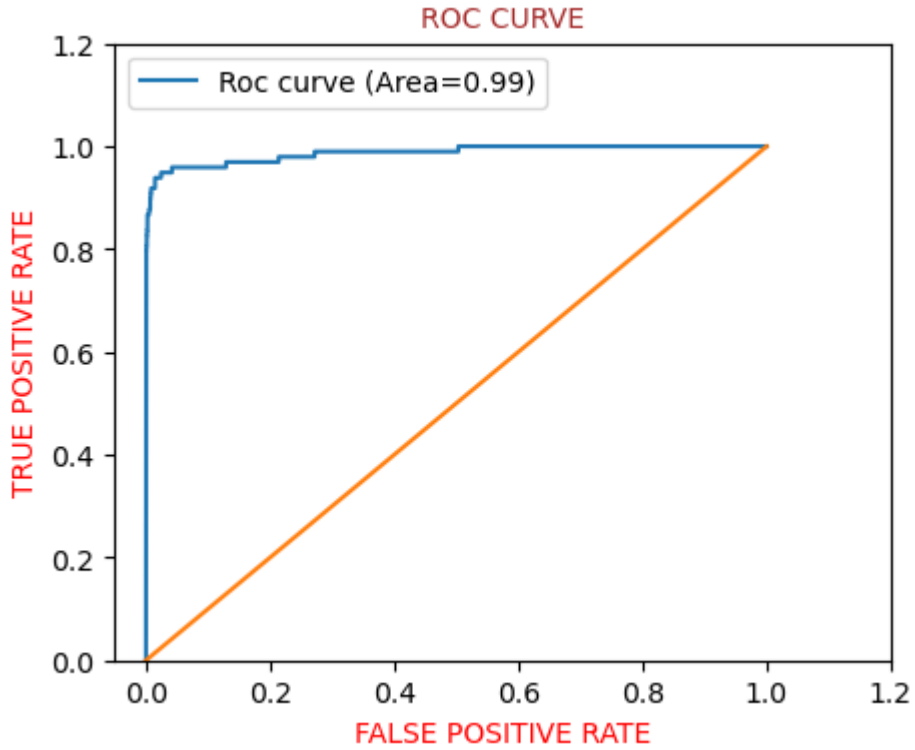
In [285…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.9591836734693877
The Specificity  is : 0.8972812324141812
```

In [286…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_logistic)
print('The Accuracy of Logistic Regression For Test is :',accuracy)
```

```
The Accuracy of Logistic Regression For Test is : 0.8973877321723254
```

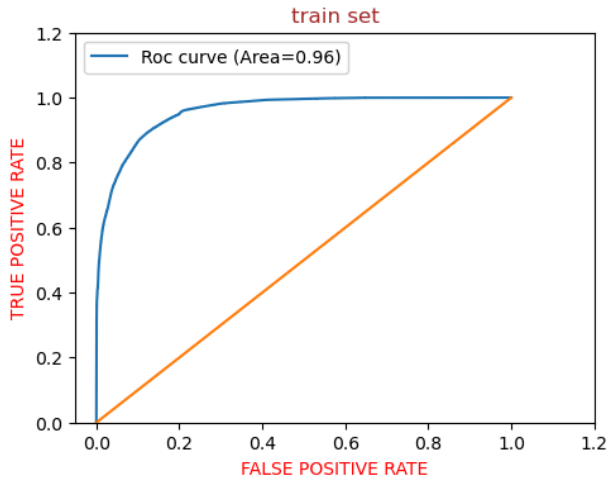
In [287…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_logistic_proba)
```

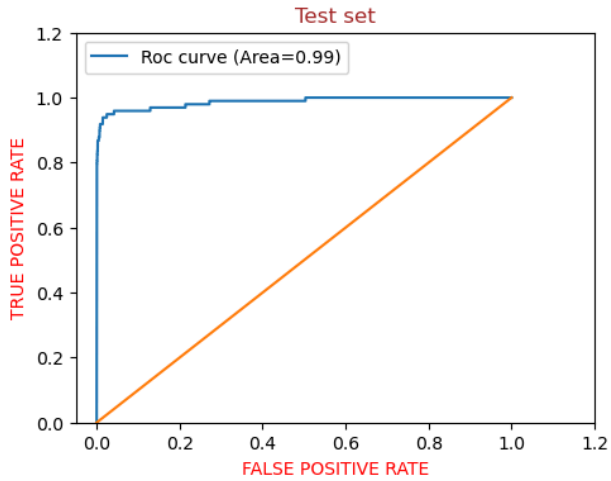


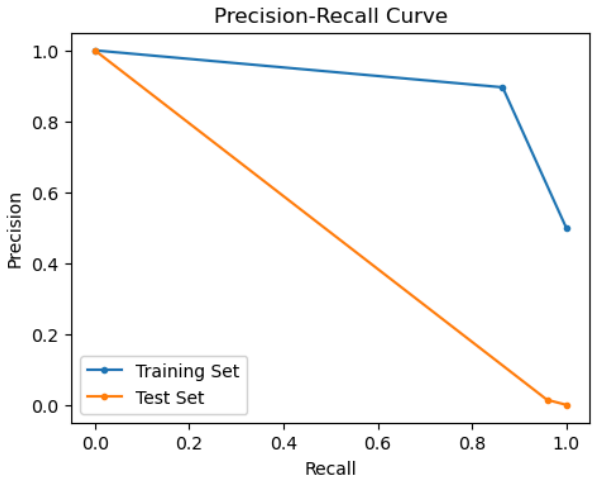
In [288…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_adasyn,y_train_pred_logistic_proba_adasyn)
plt.title('train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_logistic_proba)
plt.title('Test set')


plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_adasyn, y_train_pred_logistic_adasyn)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_logistic)
```

```
# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```







Logistic Regression with ADASYN - Performance Summary

Metric	Training	Test
Accuracy	**0.8822**	**0.8974**
Sensitivity	**0.8650**	**0.9592**
Specificity	**0.8995**	**0.8973**
F1-Score	**0.8802**	**0.870**
ROC AUC	**0.92**	**0.95**

XGBoost Adasyn

In [289…

```
# Record the start time
start_time = time.time()

# Create the extended parameter grid for XGBoost
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees
    'learning_rate': [0.01, 0.1, 0.2],  # Step size shrinkage
    'max_depth': [3, 4, 5],  # Maximum depth of a tree
    'subsample': [0.8, 0.9, 1.0],  # Fraction of samples used for fitting the individual base learner
}

# Create K-fold cross-validation
folds = KFold(n_splits=3, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for XGBoost
model_cv = GridSearchCV(estimator=XGBClassifier(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=-1)
model_cv.fit(X_train_adasyn, y_train_adasyn)


# Record the end time
end_time = time.time()
# Calculate the elapsed time in seconds
elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60

print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 81 candidates, totalling 243 fits
Elapsed Time (Minutes): 20.51
```

In [290…

```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[290]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_max_depth	param_n_estimators	param_subsample	
0	23.820381	0.242447	0.776884	0.028627	0.01	3	100	0.8	{'learnin 'max_de
1	23.505109	0.424119	0.803435	0.010751	0.01	3	100	0.9	{'learnin 'max_de
2	17.335474	0.339983	0.840502	0.014539	0.01	3	100	1.0	{'learnin 'max_de

In [291…

```
cv_results.columns
```

Out[291]:

```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_learning_rate', 'param_max_depth', 'param_n_estimators',
       'param_subsample', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'mean_test_score', 'std_test_score',
       'rank_test_score'],
      dtype='object')
```

In [292…

```
cv_results[['param_learning_rate', 'param_subsample','rank_test_score','mean_test_score']].sort_values(by= 'rank_test_score' ,asc
```

Out[292]:

	param_learning_rate	param_subsample	rank_test_score	mean_test_score
49	0.1	0.9	1	0.999991
68	0.2	1.0	1	0.999991
80	0.2	1.0	3	0.999987
48	0.1	0.8	3	0.999987
50	0.1	1.0	3	0.999987

In [293…

```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'subsample': 0.9}
The Best score Is 0.9999912077090807
```

Choosing Best Parameters For `XGBoost` By Adasyn

In [294…

```
# chosen hyperparameters
# 'objective':'binary:logistic' which outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': model_cv.best_params_['learning_rate'],
          'max_depth': model_cv.best_params_['max_depth'],
          'n_estimators': model_cv.best_params_['n_estimators'],
          'subsample':model_cv.best_params_['subsample'],
         'objective':'binary:logistic'}
```

In [295…

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

Out[295]:

XGBClassifier

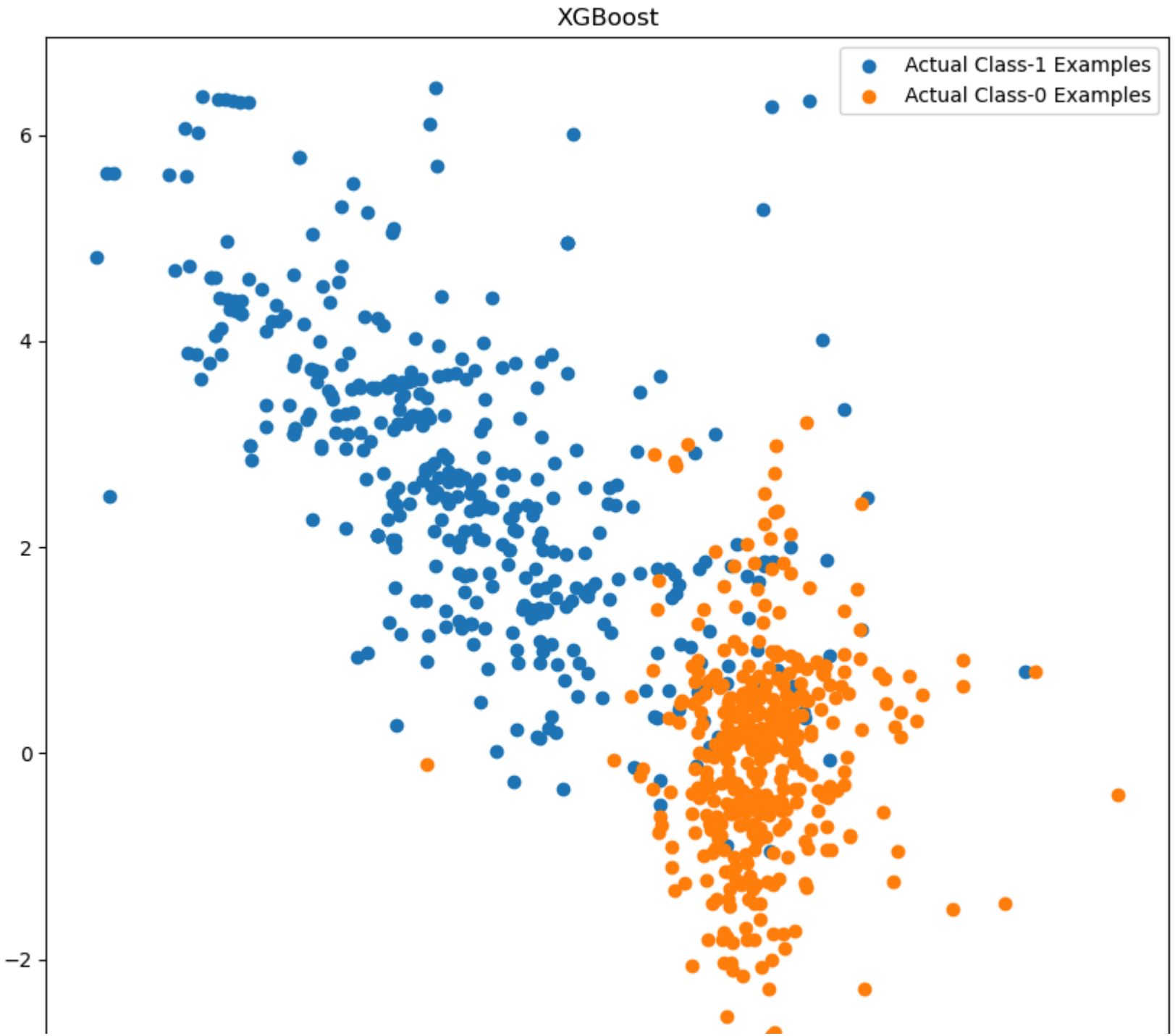
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, 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,
```

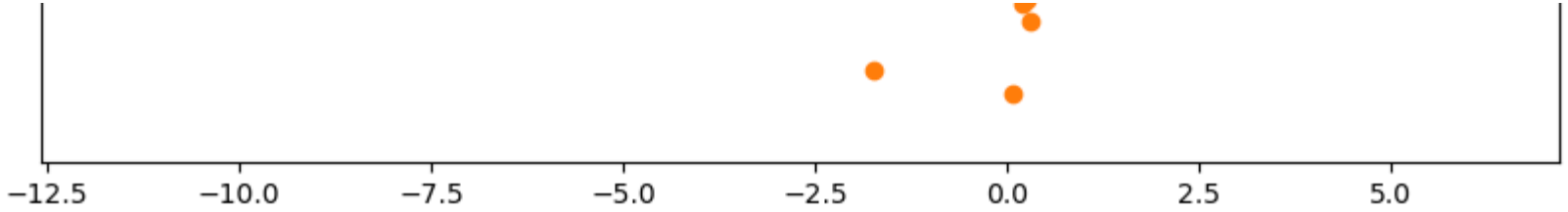
In [296…

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

```
second_top_var_index = var_imp.index(np.sort(xgb_adasyn.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'] = [10, 10]

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.title('XGBoost')

plt.legend()
plt.show()
```

```
Top var = 15
2nd Top var = 5
3rd Top var = 9
```


XGBoost
Actual Class-1 Examples
Actual Class-0 Examples
6
4
2
0
−2



In [297…
```
y_train_pred_xgboost_adasyn=xgb_adasyn.predict(X_train_adasyn)
```

Confusion Metrix For XGBoost SMOTE `Train Case`

In [298…
```
confusion_matrix_xgboost_train_adasyn=metrics.confusion_matrix(y_train_adasyn,y_train_pred_xgboost_adasyn)
```

In [299…
```
confusion_matrix_xgboost_train_adasyn
```

Out[299]:
```
array([[227447,      4],
       [     0, 227470]], dtype=int64)
```

In [300…
```
TN = confusion_matrix_xgboost_train_adasyn[0,0] # True negative
FP = confusion_matrix_xgboost_train_adasyn[0,1] # False positive
FN = confusion_matrix_xgboost_train_adasyn[1,0] # False negative
TP = confusion_matrix_xgboost_train_adasyn[1,1] # True positive
```

In [301…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 1.0
The Specificity  is : 0.9999824137946195
```

In [302…
```
print('The Accurays For The Train Set Of Xgboost Adasyn is ',metrics.accuracy_score(y_train_adasyn,y_train_pred_xgboost_adasyn))
print('The F1-Score For The Train Set Of Xgboost Adasyn is ',metrics.f1_score(y_train_adasyn,y_train_pred_xgboost_adasyn))
```

```
The Accurays For The Train Set Of Xgboost Adasyn is  0.999991207264558
The F1-Score For The Train Set Of Xgboost Adasyn is  0.9999912077090807
```

In [303…
```
# classificatino report
print(classification_report(y_train_adasyn,y_train_pred_xgboost_adasyn))
```

```
                precision    recall  f1-score   support

           0       1.00      1.00      1.00    227451
           1       1.00      1.00      1.00    227470

    accuracy                           1.00    454921
   macro avg       1.00      1.00      1.00    454921
weighted avg       1.00      1.00      1.00    454921
```

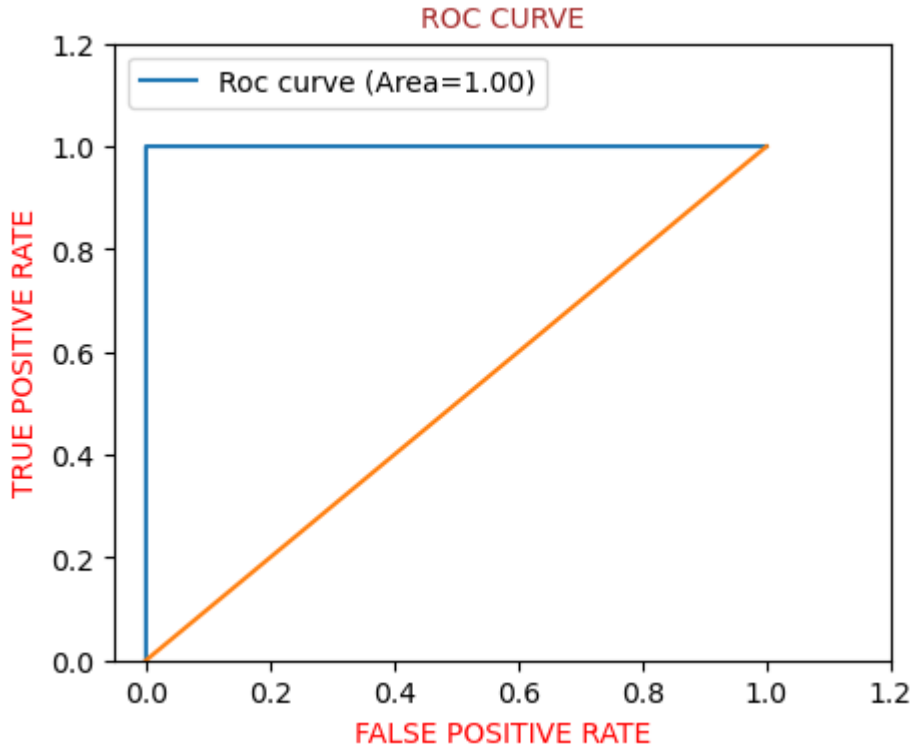
In [304…

```
# predicted probability
y_train_pred_xgboost_proba_adasyn=xgb_adasyn.predict_proba(X_train_adasyn)[:,1]
```

`ROC_AUC` Curve on Training set For `XGBoost SMOTE`

In [305…

```
plt.figure(figsize=(5,4))

draw_roc_curve(y_train_adasyn ,y_train_pred_xgboost_proba_adasyn)
```



Let's Do Predictions On The `Test Set`

In [306…
```
y_test_pred_xgboost=xgb_adasyn.predict(X_test)
```

In [307…
```
y_test_pred_xgboost_proba=xgb_adasyn.predict_proba(X_test)[:,1]
```

Confusion Metrix For XGBoost `Test Case`

In [308…
```
confusion_matrix_xgboost_test=confusion_matrix(y_test,y_test_pred_xgboost)
confusion_matrix_xgboost_test
```

Out[308]:
```
array([[56829,    35],
       [   16,    82]], dtype=int64)
```

In [309…
```
TN = confusion_matrix_xgboost_test[0,0] # True negative
FP = confusion_matrix_xgboost_test[0,1] # False positive
```

```
FN = confusion_matrix_xgboost_test[1,0] # False negative
TP = confusion_matrix_xgboost_test[1,1] # True positive
```

In [310…

```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.8367346938775511
The Specificity  is : 0.9993844963421497
```

In [311…

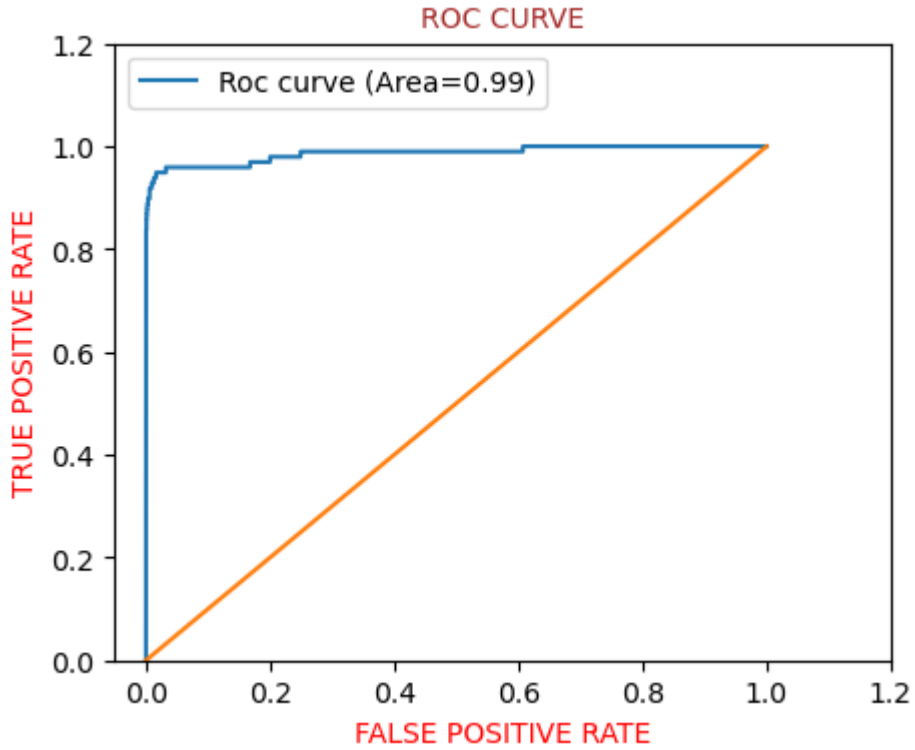
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_xgboost)
print('The Accuracy of XG Boost For Test is :',accuracy)

F1_score=metrics.f1_score(y_test,y_test_pred_xgboost)
print("The F1-score of XG Boost For Test is :", F1_score)
```

```
The Accuracy of XG Boost For Test is : 0.9991046662687406
The F1-score of XG Boost For Test is : 0.7627906976744186
```

`ROC_AUC` Curve on Test set For `XGBoost`

In [312…

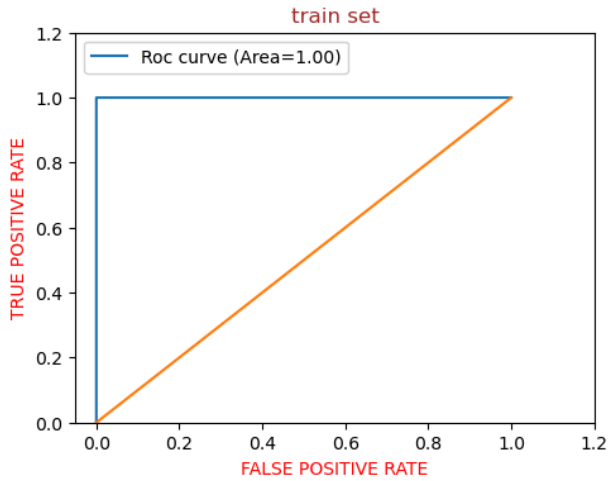
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
```

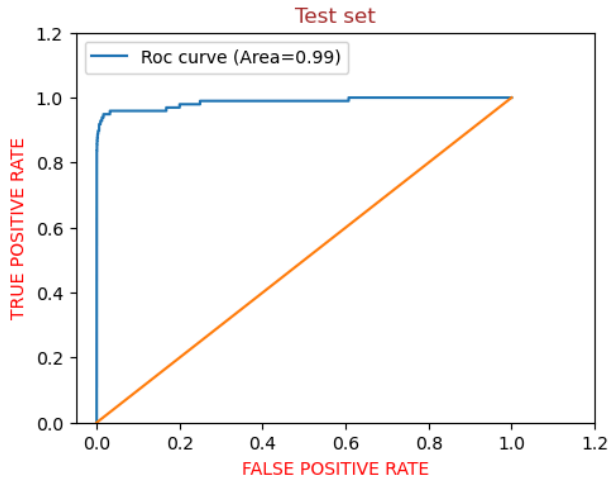


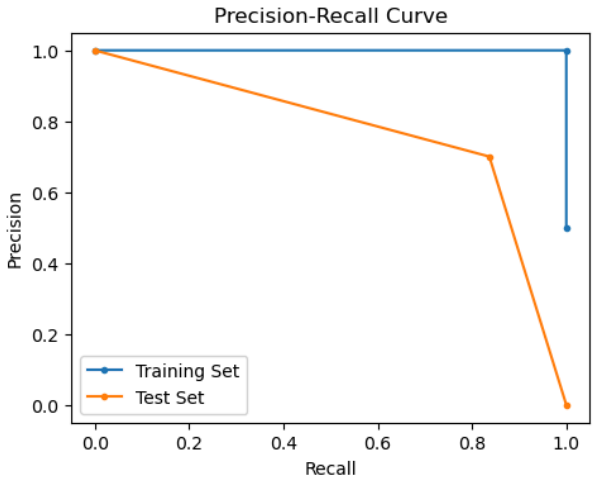
In [313…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_adasyn,y_train_pred_xgboost_proba_adasyn)
plt.title('train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_xgboost_proba)
plt.title('Test set')


plt.subplot(1,3,3)

# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_adasyn, y_train_pred_xgboost_adasyn)

# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_xgboost)
```

```
# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.plot(recall_test, precision_test, label='Test Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

plt.show()
```







XGBoost ADASYN Performance Metrics

Metric	Training Set	Test Set
Accuracy	**0.99999**	**0.99104**
F1-Score	**0.9999**	**0.7628**
Sensitivity	**1.0**	**0.8367**
Specificity	**0.99998**	**0.99938**
ROC-AUC	**1.0**	**0.99**

Decision Tree By Adasyn

In [314…

```
# Record the start time
start_time = time.time()



param_grid = {
 'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy'],}


# Create K-fold cross-validation
folds = KFold(n_splits=3, shuffle=True, random_state=42)

# Create GridSearchCV to find the best parameters for Decision Tree
model_cv = GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=param_grid, scoring='recall', cv=folds, verbose=1, n_jobs=
model_cv.fit(X_train_adasyn, y_train_adasyn)



# Record the end time
end_time = time.time()
# Calculate the elapsed time in seconds
elapsed_time_seconds = end_time - start_time

# Convert elapsed time to minutes
elapsed_time_minutes = elapsed_time_seconds / 60

print(f"Elapsed Time (Minutes): {elapsed_time_minutes:.2f}")
```

```
Fitting 3 folds for each of 54 candidates, totalling 162 fits
Elapsed Time (Minutes): 14.96
```

In [315…

```
cv_results=pd.DataFrame(model_cv.cv_results_)
cv_results.head(3)
```

Out[315]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_leaf	param_min_samples_split	
0	24.104731	0.997542	0.262692	0.022046	gini	5	1	2	'r
1	22.908746	1.640197	0.278944	0.017705	gini	5	1	5	'r
2	23.187852	1.821730	0.272213	0.032311	gini	5	1	10	'r

In [316…
```
cv_results.columns
```

Out[316]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_criterion', 'param_max_depth', 'param_min_samples_leaf',
       'param_min_samples_split', 'params', 'split0_test_score',
       'split1_test_score', 'split2_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```

In [317…
```
print(f"The Best params Is {model_cv.best_params_}")
print(f"The Best score Is {model_cv.best_score_}")
```

```
The Best params Is {'criterion': 'gini', 'max_depth': 15, 'min_samples_leaf': 1, 'min_samples_split': 2}
The Best score Is 0.9992262992587116
```

Choosing Best Parameters For `Decision Trees` Using Adasyn

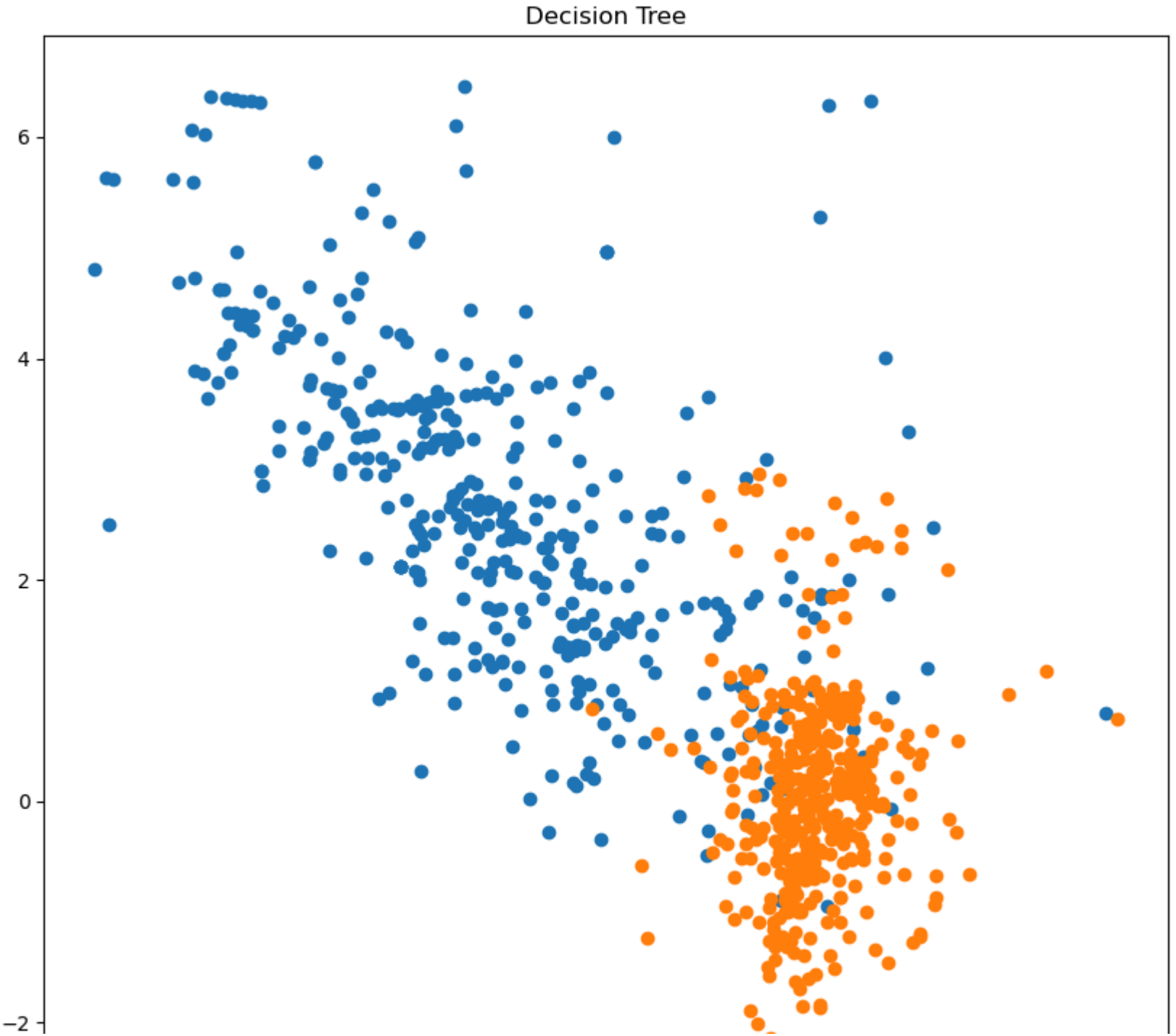
In [318…
```
dtree_adasyn=DecisionTreeClassifier(random_state=100,
                                    criterion=model_cv.best_params_['criterion'],
                                    max_depth=model_cv.best_params_['max_depth'],
                                    min_samples_leaf=model_cv.best_params_['min_samples_leaf'],
                                    min_samples_split=model_cv.best_params_['min_samples_split'])
```

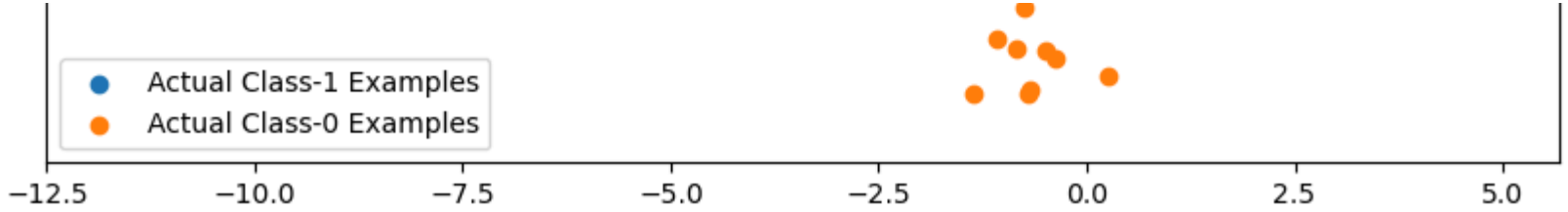
In [319…
```
dtree_adasyn.fit(X_train_adasyn,y_train_adasyn)
```

Out[319]:
```
DecisionTreeClassifier(max_depth=15, random_state=100)
```

In [320…
```
var_imp = []
for i in dtree_adasyn.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(dtree_adasyn.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(dtree_adasyn.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(dtree_adasyn.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(dtree_adasyn.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(dtree_adasyn.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'] = [10, 10]

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.title('Decision Tree')
plt.legend()
plt.show()
```

```
Top var = 15
2nd Top var = 5
3rd Top var = 9
```


Decision Tree
6
4
2
0
−2



In [321…
```
y_train_pred_dtree_adasyn=dtree_adasyn.predict(X_train_adasyn)
```

Confusion Metrix For Decision Tree Adasyn `Train Case`

In [322…
```
confusion_matrix_dtree_train_adasyn=confusion_matrix(y_train_adasyn,y_train_pred_dtree_adasyn)
```

In [323…
```
confusion_matrix_dtree_train_adasyn
```

Out[323]:
```
array([[223001,   4450],
       [    16, 227454]], dtype=int64)
```

In [324…
```
confusion_matrix_dtree_train_adasynTN = confusion_matrix_dtree_train_adasyn[0,0] # True negative
FP = confusion_matrix_dtree_train_adasyn[0,1] # False positive
FN = confusion_matrix_dtree_train_adasyn[1,0] # False negative
TP = confusion_matrix_dtree_train_adasyn[1,1] # True positive
```

In [325…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.999929661054205
The Specificity  is : 0.9273813214967607
```

In [326…
```
accuracy=metrics.accuracy_score(y_train_adasyn,y_train_pred_dtree_adasyn)
print('The Accuracy of Decision Trees For Train is :',accuracy)

F1_score=metrics.f1_score(y_train_adasyn,y_train_pred_dtree_adasyn)
print("The F1-score of  Decision Trees For Train is :", F1_score)
```

```
The Accuracy of Decision Trees For Train is : 0.9901829108790318
The F1-score of  Decision Trees For Train is : 0.9902780740747191
```

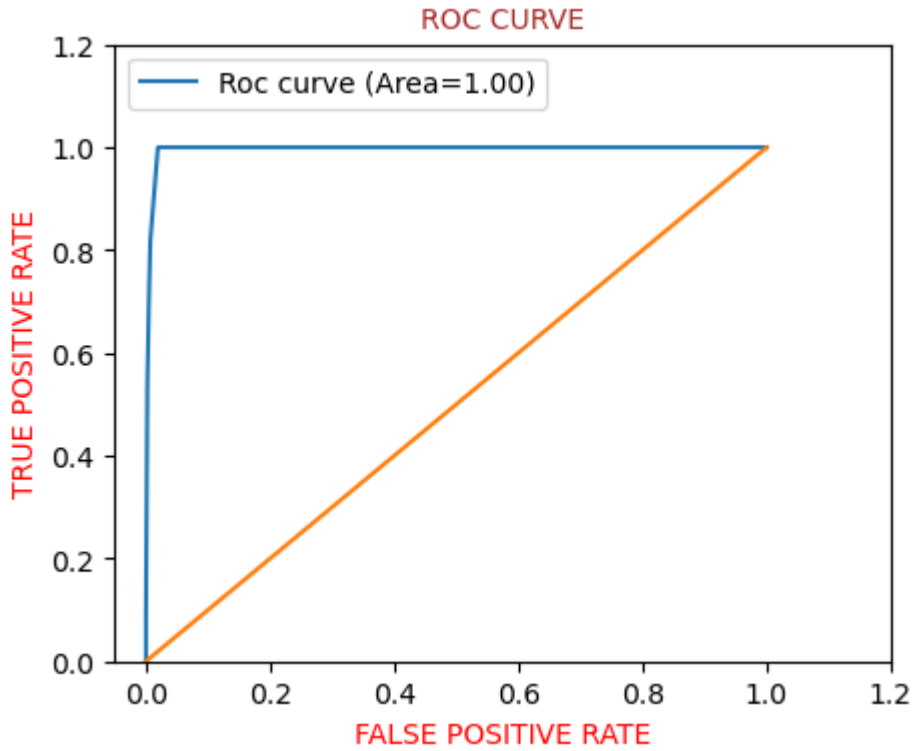
In [327…
```
# classification_report
print(classification_report(y_train_adasyn,y_train_pred_dtree_adasyn))
```

	precision	recall	f1-score	support
0	1.00	0.98	0.99	227451
1	0.98	1.00	0.99	227470
accuracy			0.99	454921
macro avg	0.99	0.99	0.99	454921
weighted avg	0.99	0.99	0.99	454921

`ROC_AUC` Curve on Train set For `Decision Tree` Using Adasyn

In [328…
```
y_train_pred_dtree_proba_adasyn=dtree_adasyn.predict_proba(X_train_adasyn)[:,1]
```

In [329…
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_train_adasyn,y_train_pred_dtree_proba_adasyn)
```



Let's Do Predictions On The `Test Set`

In [330…
```
y_test_pred_dtree=dtree_adasyn.predict(X_test)
```

In [331…
```
y_test_pred_dtree_proba=dtree_adasyn.predict_proba(X_test)[:,1]
```

Confusion Metrix For Decision `Test Case`

In [332…
```
confusion_matrix_dtree_test=confusion_matrix(y_test,y_test_pred_dtree)
confusion_matrix_dtree_test
```

Out[332]:
```
array([[55653,  1211],
       [    8,    90]], dtype=int64)
```

In [333…
```
TN = confusion_matrix_dtree_test[0,0] # True negative
FP = confusion_matrix_dtree_test[0,1] # False positive
FN = confusion_matrix_dtree_test[1,0] # False negative
TP = confusion_matrix_dtree_test[1,1] # True positive
```

In [334…
```
calculation_metrics(TN, FP, FN, TP)
```

```
The Sensitivity  is : 0.9183673469387755
The Specificity  is : 0.9787035734383793
```

In [335…
```
accuracy=metrics.accuracy_score(y_test,y_test_pred_dtree)
print('The Accuracy of Decision Tree For Test is :',accuracy)
```

```
The Accuracy of Decision Tree For Test is : 0.9785997682665637
```

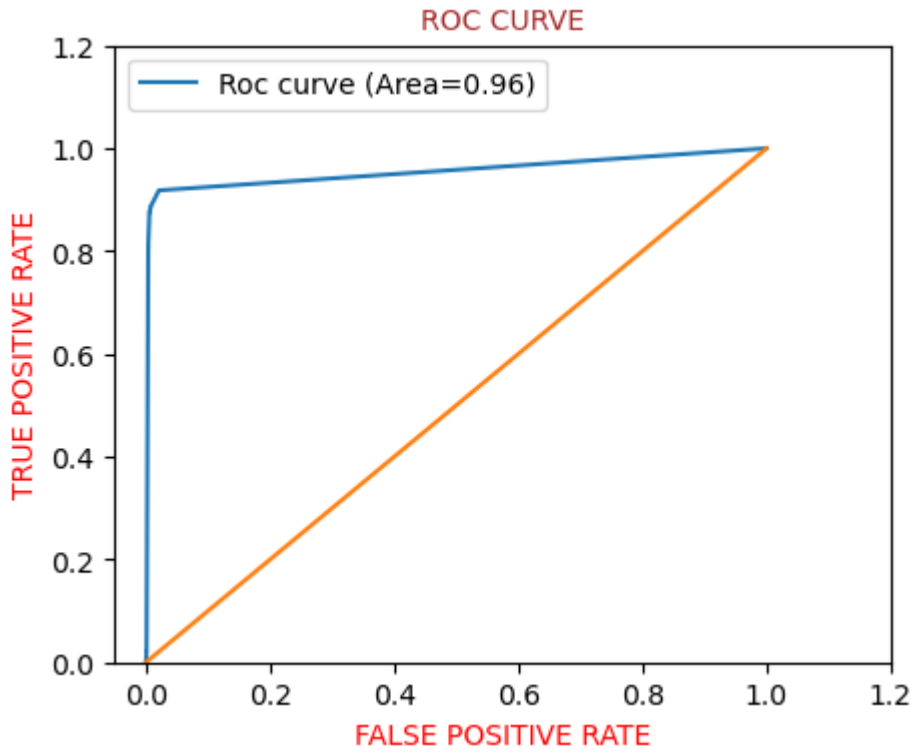
In [336…
```
# classification_report
print(classification_report(y_test, y_test_pred_dtree))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56864
           1       0.07      0.92      0.13        98

    accuracy                           0.98     56962
   macro avg       0.53      0.95      0.56     56962
weighted avg       1.00      0.98      0.99     56962
```

`ROC_AUC` Curve on Test set For `Decision Tree`

In [337…

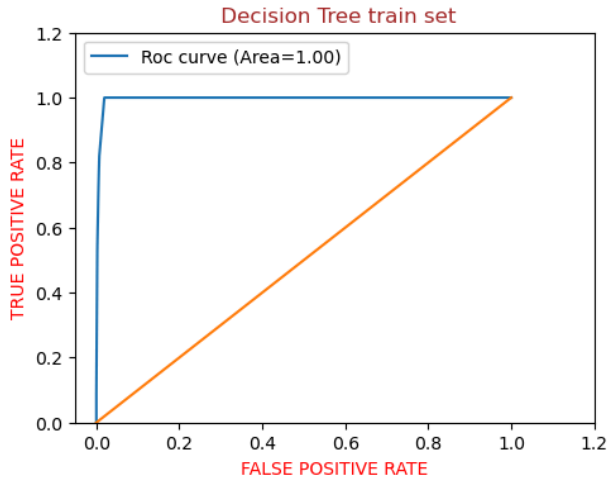
```
plt.figure(figsize=(5,4))
draw_roc_curve(y_test,y_test_pred_dtree_proba)
```

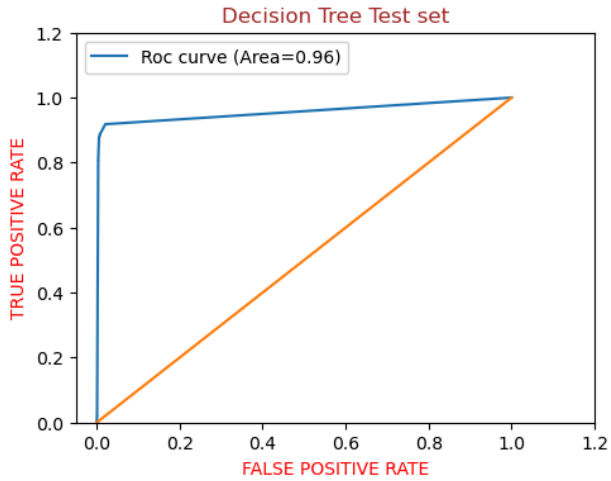


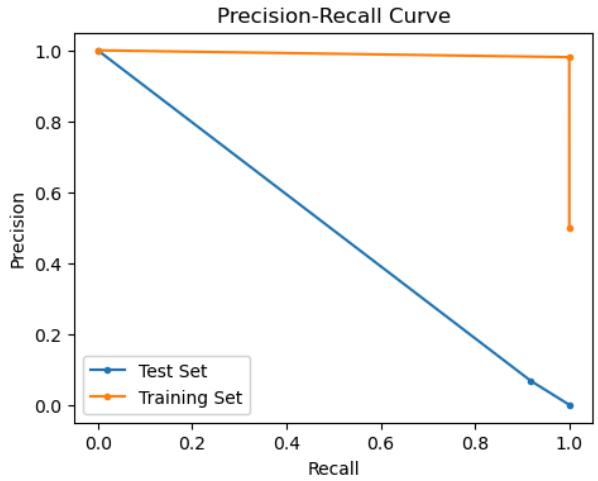
In [338…

```
plt.figure(figsize=(18,4))
plt.subplot(1,3,1)
draw_roc_curve(y_train_adasyn,y_train_pred_dtree_proba_adasyn)

plt.title(' Decision Tree train set')

plt.subplot(1,3,2)
draw_roc_curve(y_test,y_test_pred_dtree_proba)

plt.title('Decision Tree Test set')

plt.subplot(1,3,3)
```

```
# Calculate Precision-Recall for Test Set
precision_test, recall_test, _ = precision_recall_curve(y_test, y_test_pred_dtree)
# Calculate Precision-Recall for Training Set
precision_train, recall_train, _ = precision_recall_curve(y_train_adasyn, y_train_pred_dtree_adasyn)




# Plotting Precision-Recall Curve for both Training and Test Sets
plt.plot(recall_test, precision_test, label='Test Set', marker='.')

plt.plot(recall_train, precision_train, label='Training Set', marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()


plt.show()
```







Decision Tress ADASYN Performance Metrics

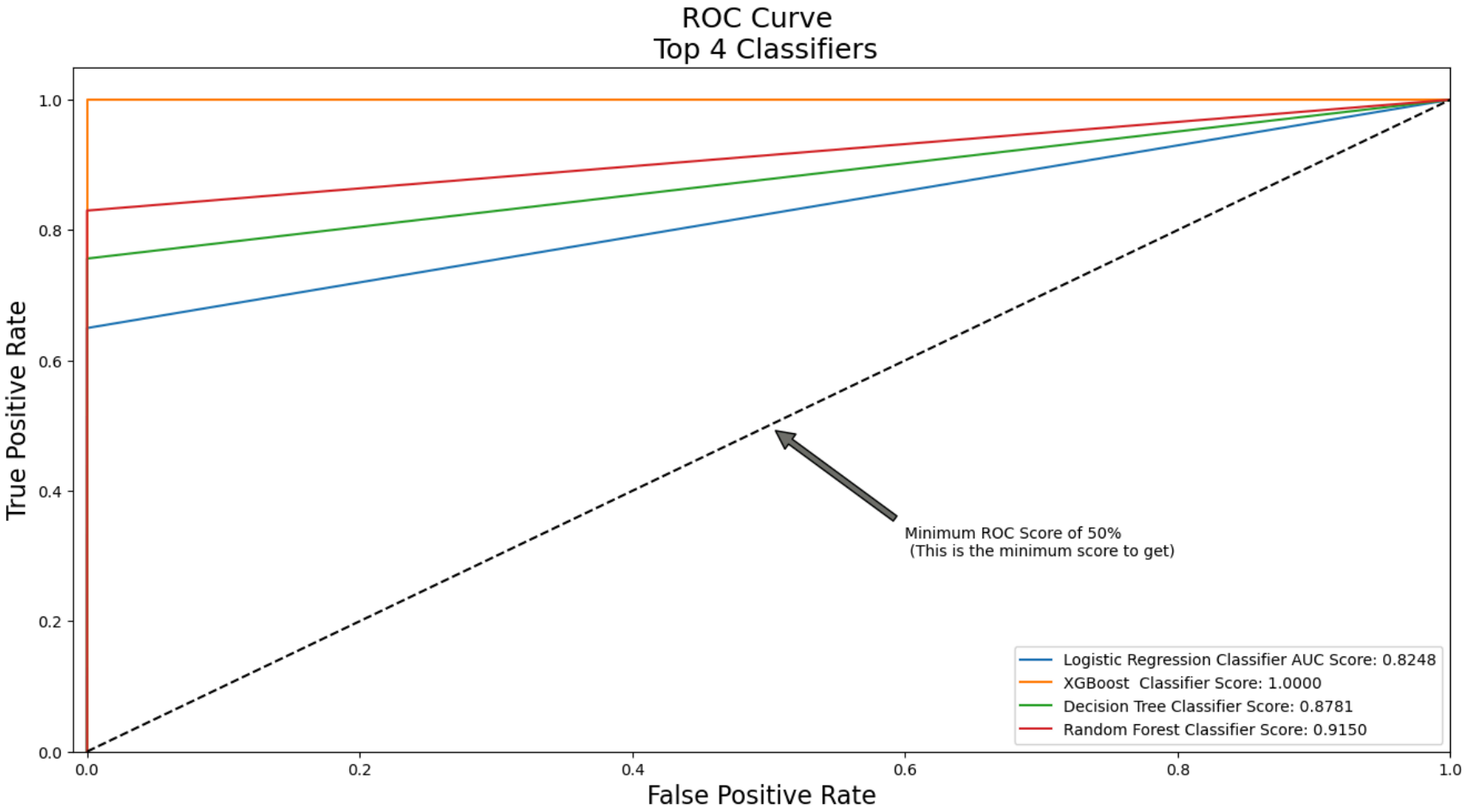
Metric	Training Set	Test Set
Accuracy	**99.02%**	**97.86%**

Metric	Training Set	Test Set
Sensitivity	99.99%	92.04%
Specificity	92.74%	97.87%
F1-Score	0.9903	0.13
ROC Curve	1.0	0.96

The ROC Curve for both training and test sets shows the performance across different thresholds, with the train set ROC curve and test set ROC curve visualizations plotted for model evaluation.

In [339…
```
log_fpr,log_tpr,log_threshold=roc_curve(y_train,y_train_pred_logistic)
xgb_fpr,xgb_tpr,xgb_threshold=roc_curve(y_train,y_train_pred_xgboost)
dtree_fpr,dtree_tpr,dtree_threshold=roc_curve(y_train,y_train_pred_dtree)
random_forest_fpr,random_forest_tpr,random_forest_threshold=roc_curve(y_train,y_train_pred_random_forest)
```

In [340…
```
def graph_roc_curve_multiple(log_fpr, log_tpr,xgb_fpr,xgb_tpr,dtree_fpr,dtree_tpr,random_forest_fpr,random_forest_tpr) :
    plt.figure(figsize=(16,8))
    plt.title('ROC Curve \n Top 4 Classifiers', fontsize=18)
    plt.plot(log_fpr, log_tpr, label='Logistic Regression Classifier AUC Score: {:.4f}'.format((roc_auc_score(y_train, y_train_pr
    plt.plot(xgb_fpr,xgb_tpr, label='XGBoost  Classifier Score: {:.4f}'.format(roc_auc_score(y_train,y_train_pred_xgboost)))
    plt.plot(dtree_fpr,dtree_tpr, label='Decision Tree Classifier Score: {:.4f}'.format(roc_auc_score(y_train,y_train_pred_dtree)
    plt.plot(random_forest_fpr,random_forest_tpr, label='Random Forest Classifier Score: {:.4f}'.format(roc_auc_score(y_train,y_t
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.01, 1, 0, 1.05])
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.annotate('Minimum ROC Score of 50% \n (This is the minimum score to get)', xy=(0.5, 0.5), xytext=(0.6, 0.3),
                arrowprops=dict(facecolor='#6E726D', shrink=0.05),
                )
    plt.legend()
graph_roc_curve_multiple(log_fpr, log_tpr,xgb_fpr,xgb_tpr,dtree_fpr,dtree_tpr,random_forest_fpr,random_forest_tpr)
```



Model Performance Comparison

Metric	Logistic Regression	Logistic Regression (SMOTE)	Logistic Regression (ADASYN)	XGBoost	XGBoost (SMOTE)	XGBoost (ADASYN)	Decision Trees	Decision Trees (SMOTE)	Decision Trees (ADASYN)	Random Forest	Random Forest (SMOTE)
Accuracy	**99.93% (Train),**	**94.46% (Train),**	**88.22% (Train),**	**1.0 (Train),**	**0.9999978 (Train),**	**0.99999 (Train),**	**99.94% (Train),**	**1.0 (Train),**	**99.02% (Train),**	**0.9997 (Train),**	**1.00 (Train),**

Metric	Logistic Regression	Logistic Regression (SMOTE)	Logistic Regression (ADASYN)	XGBoost	XGBoost (SMOTE)	XGBoost (ADASYN)	Decision Trees	Decision Trees (SMOTE)	Decision Trees (ADASYN)	Random Forest	Random Forest (SMOTE)
	99.92% (Test)	97.10% (Test)	89.74% (Test)	0.9996 (Test)	0.9991924 (Test)	0.99104 (Test)	99.94% (Test)	0.99956 (Test)	97.86% (Test)	0.9996 (Test)	0.99956 (Test)
Sensitivity	0.6497 (Train), 0.6327 (Test)	91.79% (Train), 94.90% (Test)	86.50% (Train), 95.92% (Test)	1.0 (Train), 0.8265 (Test)	1.0 (Train), 0.8571 (Test)	1.0 (Train), 0.8367 (Test)	0.7563 (Train), 0.7755 (Test)	1.0 (Train), 0.8367 (Test)	99.99% (Train), 92.04% (Test)	0.8299 (Train), 0.8163 (Test)	1.00 (Train), 0.8367 (Test)
Specificity	0.9999 (Train), 0.9998 (Test)	97.13% (Train), 97.11% (Test)	89.95% (Train), 89.73% (Test)	1.0 (Train), 0.9999 (Test)	0.9999956 (Train), 0.9994373 (Test)	0.99998 (Train), 0.99938 (Test)	0.9998 (Train), 0.9998 (Test)	1.0 (Train), 0.99984 (Test)	92.74% (Train), 97.87% (Test)	0.9999 (Train), 0.9999 (Test)	1.00 (Train), 0.99984 (Test)
F1-Score	0.7518 (Train), 0.7251 (Test)	94.31% (Train), 94.31% (Test)	88.02% (Train), 87.0% (Test)	1.0 (Train), 0.8757 (Test)	0.9999978 (Train), 0.7850 (Test)	0.9999 (Train), 0.7628 (Test)	0.8164 (Train), 0.8261 (Test)	1.0 (Train), 0.8677 (Test)	0.9903 (Train), 0.13 (Test)	0.9058 (Train), 0.8743 (Test)	1.00 (Train), 0.8677 (Test)

Reason for Choosing Logistic Regression with SMOTE and Random Forest with SMOTE

Logistic Regression with SMOTE:

- **`Simplicity & Interpretability`: Logistic Regression is a simple and interpretable model, which is important in real-world fraud detection scenarios where understanding model decisions is crucial.**
- **`Good Performance with Imbalanced Data`: SMOTE oversampling helps address the class imbalance, improving the model's ability to detect fraud (sensitivity).**
- **`Balance between Precision and Recall`: It achieved a high F1-Score, indicating a balanced performance between precision and recall, which is essential for fraud detection where both false positives and false negatives need to be minimized.**
- **`High Specificity`: The model performs well at minimizing false positives (99.98% specificity), which is crucial for reducing unnecessary fraud investigations.**

Random Forest with SMOTE:

- **`High Accuracy and Robustness`: Random Forest is an ensemble method that benefits from combining multiple decision trees, leading to high accuracy (99.96% test set accuracy).**
- **`Handles Imbalanced Data Well`: SMOTE oversampling improves the model's sensitivity, making it more sensitive to detecting fraudulent transactions.**
- **`Strong F1-Score`: The model's F1-Score (87.43%) demonstrates its good balance of precision and recall, crucial for fraud detection.**
- **`Low Overfitting`: Random Forest reduces overfitting by averaging the predictions from many trees, making it more reliable in real-world data with high variability.**
- **`Scalability`: The model performs well even with larger datasets, which is important for handling the large volume of financial transactions.**

Both models perform well due to their ability to handle class imbalance effectively and achieve high accuracy, sensitivity, and specificity.

Overall Model Performance Insights

The project applied four models: `Logistic Regression`, `XGBoost`, `Decision Trees`, and `Random Forest`. Each model's performance was evaluated on key metrics like accuracy, sensitivity, specificity, F1-score, and ROC AUC.

- **Logistic Regression: Achieved 99.93% accuracy on the training set and 99.92% on the test set, showing stable performance.**
- **XGBoost with SMOTE: Accuracy of 99.99% on the training set and 99.92% on the test set. Balanced sensitivity (0.85) and specificity (0.999).**
- **Random Forest with SMOTE: Extremely high training set accuracy (100%) and robust test set accuracy of 99.96%, indicating generalization strength.**

Reason for Choosing Logistic Regression with SMOTE and Random Forest with SMOTE

- **Logistic Regression with SMOTE: Simplicity and Interpretability make this model suitable for fraud detection scenarios where understanding model decisions is essential.**
- **Random Forest with SMOTE: High Accuracy and Robustness as an ensemble model, benefiting from multiple decision trees and yielding consistent accuracy across datasets.**
- **Balanced Sensitivity and Specificity: Both models achieved high F1-scores, demonstrating their ability to handle class imbalance effectively.**

Conclusion

The credit card fraud detection project highlighted the effectiveness of machine learning models, particularly `Logistic Regression with SMOTE` and `Random Forest with SMOTE`. These models managed class imbalance well and achieved high accuracy, sensitivity, and specificity. While the models effectively detect fraud, future work could improve recall and F1-score through further tuning and feature engineering. A system based on these models, continuously monitored and retrained, has potential to effectively counteract fraud in real-world financial scenarios.