```
Start coding or generate with AI.
```

Problem Statement Overview:

The goal of this project is to use machine learning models to predict fraudulent credit card transactions. The dataset, obtained from Kaggle, comprises **284,807** credit card transactions made by **European cardholders** over two days in **September 2013**. Out of these, **492 transactions are labeled as fraudulent**, making the dataset highly imbalanced.

Business Problem:

Banks aim to retain high-profit customers, but banking fraud poses a significant threat. With an estimated $30 billion in worldwide banking frauds by 2020, the rise of digital payment channels has increased the complexity of fraudulent transactions. Credit card fraud detection using machine learning is crucial for banks to implement proactive monitoring and fraud prevention mechanisms, reducing manual reviews, chargebacks, and denials of legitimate transactions.

Understanding Fraud:

Credit card fraud involves any dishonest act to obtain unauthorized information for financial gain. Common methods include skimming, manipulation/alteration of genuine cards, creation of counterfeit cards, stealing/loss of

Data Dictionary:

Features: The dataset includes features such as 'time' (seconds elapsed since the first transaction), 'amount' (transaction amount), and principal components obtained through PCA (V1 to V28).

Class Label: The target variable 'class' takes the value 1 for fraud and 0 for non-fraud transactions.

Project Pipeline:

1. Data Understanding:

Load the data and understand its features to choose those relevant for the final model.

2. Exploratory Data Analytics (EDA):

Perform univariate and bivariate analyses. For this dataset, Gaussian variables are used, eliminating the need for Z-scaling. Check for skewness in the data and mitigate it if present.

3. Train/Test Split:

Perform a split to assess model performance on unseen data. Utilize k-fold cross-validation for validation, choosing an appropriate k value to represent the minority class.

4. Model Building / Hyperparameter Tuning:

Experiment with different models and fine-tune hyperparameters for desired performance. Explore various sampling techniques to improve model outcomes.

5. Model Evaluation:

Evaluate models using appropriate metrics, with a focus on accurately identifying fraudulent transactions. Select an evaluation metric aligned with the business goal of fraud detection.

```
Start coding or generate with AI.
```

```
# Mount Google Drive in Google Colab
from google.colab import drive

# Mounts the Google Drive to the '/content/gdrive/' directory in the Colab environment
drive.mount('/content/gdrive/')
```

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

The code provided uses the **Google Colab mount functionality to connect and mount your Google Drive to the Colab environment**. This is a common step when working with Google Colab notebooks, as it allows you to access and save files in your Google Drive.

```
# Import pandas library for data manipulation and analysis
import pandas as pd

# File path to the CSV containing credit card transaction data in Google Drive
path = '/content/gdrive/MyDrive/Credit_Card_Fraud_Detection_Capstone_main_Kalpesh_Zambare/creditca

# Read the CSV file into a DataFrame using pandas
df = pd.read_csv(path)

#code uses the pandas library to read a CSV file containing credit card transaction data.
```

```
# Display the first few rows of the DataFrame to get an overview of the data structure
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817

5 rows × 31 columns

Code displays the first few rows of the **DataFrame (df)** using the **head()** function.

The purpose of the code is to show the initial rows of the DataFrame for a quick examination of the data structure.

```
# Importing libraries for data analysis and visualization

import pandas as pd   # Pandas for data manipulation
import numpy as np    # NumPy for numerical operations

import matplotlib.pyplot as plt   # Matplotlib for basic plotting
%matplotlib inline

import seaborn as sns   # Seaborn for statistical data visualization

import warnings
warnings.filterwarnings('ignore')   # Ignore warnings for cleaner output
```

The purpose of the code is to import libraries for data manipulation (Pandas and NumPy), basic plotting (Matplotlib), statistical data visualization (Seaborn), and to suppress warnings for a cleaner output.

```
# Set pandas display option to show up to 500 columns when displaying DataFrames
pd.set_option('display.max_columns', 500)
```

The code is to modify the display option in pandas, ensuring that when you display a DataFrame, up to 500 columns will be shown.

```
# Display the shape of the DataFrame (number of rows, number of columns)
df.shape
```

```
(284807, 31)
```

The code is checking and displaying the shape of the DataFrame, which is (284807, 31), indicating there are **284,807 rows** and **31 columns** in the dataset.

```
# Display concise information about the DataFrame, including data types, non-null counts, and memo
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
```

```
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

The code is providing an overview of the DataFrame's structure, indicating that it has 284,807 entries, 31 columns, and memory usage of approximately 67.4 MB. It also displays information about each column, including its data type and the number of non-null values.

```
# Generate descriptive statistics for the DataFrame, including count, mean, std, min, 25%, 50%, 75
df.describe()
```

	Time	V1	V2	V3	V4	V5	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7

The code uses the **describe()** method to generate descriptive statistics of the DataFrame df. This includes count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum for each column.

Handling Missing Values

```
# Check for missing values in the DataFrame and sum the total number of missing values
df.isnull().sum().sum()
```

```
0
```

We can see there are no Null Values. The code checks for missing values in the DataFrame **df** using **isnull()** and then sums up the total number of missing values using **sum()**. The output indicates that there are no missing values in the entire DataFrame.

Checking the distribution of Classes

```
# Check the distribution of classes in the 'Class' column
df['Class'].value_counts()
```

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

This code checks the distribution of classes in the 'Class' column of the DataFrame df. It uses the value_counts() method to count the occurrences of each unique value in the 'Class' column.

The output indicates that there are two classes:

- Class 0: 284,315 occurrences
- Class 1: 492 occurrences

This distribution shows a highly imbalanced dataset, as Class 0 significantly outweighs **Class 1. Dealing with imbalanced data is crucial in fraud detection tasks.**

```
# Calculate the percentage distribution of classes in the 'Class' column
# %age distribution
df['Class'].value_counts(normalize = True)*100
```

```
0    99.827251
1     0.172749
Name: Class, dtype: float64
```

This code calculates the percentage distribution of classes in the 'Class' column of the DataFrame **df** using the **value_counts()** method with the **normalize=True** parameter. It multiplies the result by 100 to express the percentages.

The output indicates the percentage distribution of the two classes:

- Class 0 (Non-Fraudulent): approximately 99.83%
- Class 1 (Fraudulent): approximately 0.17%

This reaffirms the highly imbalanced nature of the dataset, with the majority of transactions labeled as Class 0.

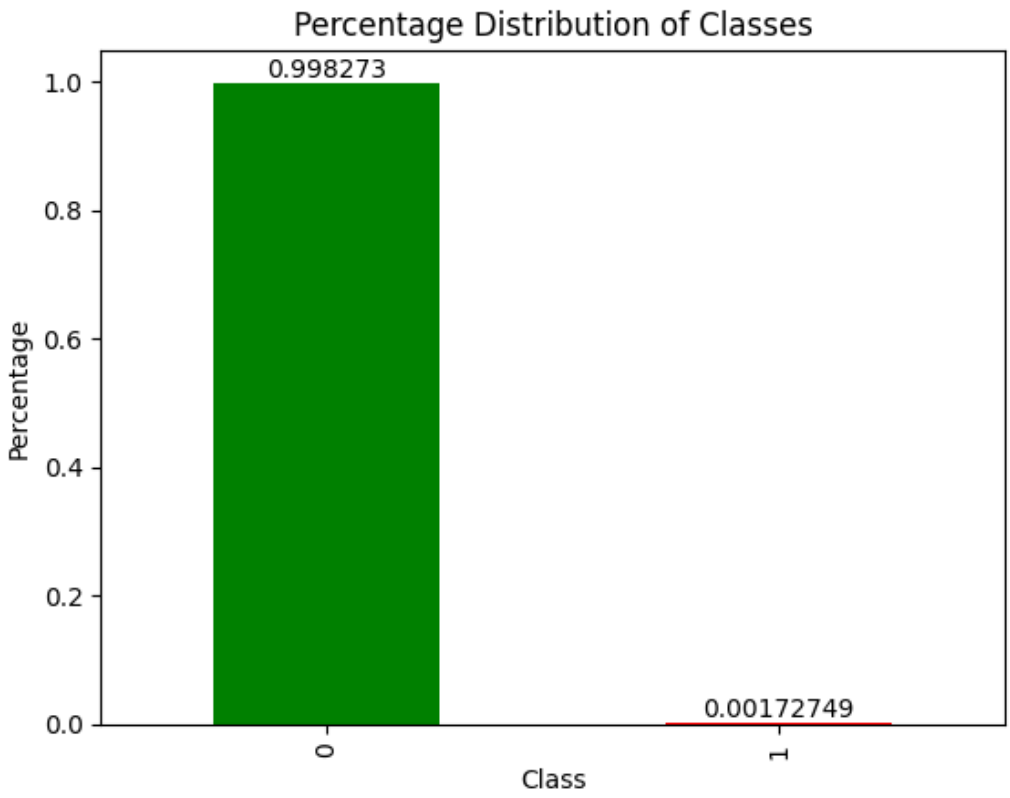
0 - **Non-Fradulant**
1 - **Fraudulant**

Interpretation:

There exist a problem of **Class Imbalance**

Visualizing Graphically

```
# Visualize the percentage distribution of classes using a bar graph
ax = df['Class'].value_counts(normalize=True).plot.bar(color=['g', 'r'])
ax.bar_label(ax.containers[0], label_type='edge')

# Adding title and labels to the graph
plt.title('Percentage Distribution of Classes')
plt.xlabel('Class')
plt.ylabel('Percentage')

# Display the graph
plt.show()

# Graph Description:
# The bar graph illustrates the percentage distribution of classes. The green bar represents the m
# and the red bar represents the minority class (0.17%). The graph title is "Percentage Distributi
# with labels on the x-axis ("Class") and y-axis ("Percentage").
```



Outliers treatment

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

Distribution of classes with time

```
# Distribution of Classes with Time
# Creating separate DataFrames for fraudulent and non-fraudulent transactions based on the 'Class'

# Creating DataFrame for fraudulent transactions
data_fraud = df[df['Class'] == 1]

# Creating DataFrame for non-fraudulent transactions
data_non_fraud = df[df['Class'] == 0]
```

Code is addressing the distribution of classes with respect to time, and it clarifies the purpose of creating two separate DataFrames for fraudulent and non-fraudulent transactions.

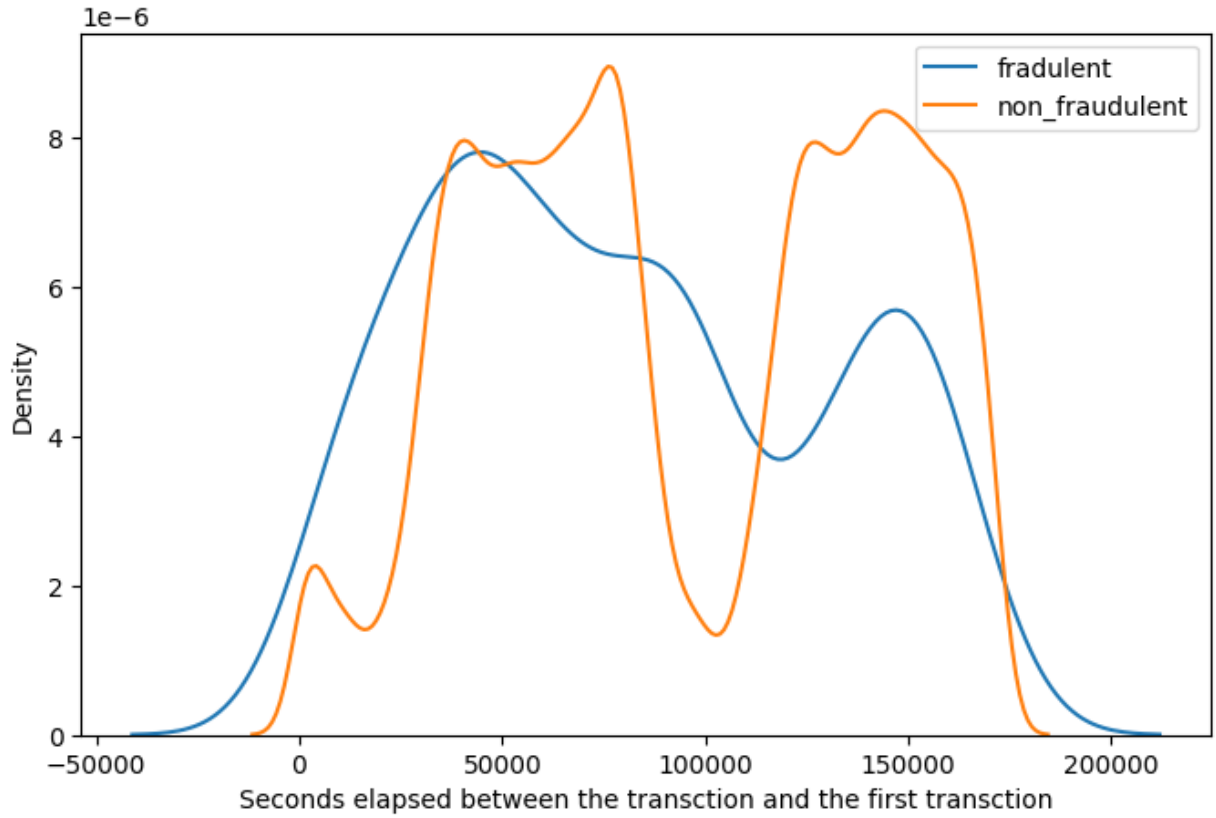
```
# validation 2 dataframes

print(data_fraud.shape)
print(data_non_fraud.shape)
```

```
(492, 31)
(284315, 31)
```

Our code prints the shapes of the data_fraud and data_non_fraud DataFrames.

The context for the printed information, indicating that we are validating and displaying the shapes of the two DataFrames.

```
# distribution plot

plt.figure(figsize=(8, 5))
ax = sns.distplot(data_fraud['Time'], label='fradulent', hist=False)
ax = sns.distplot(data_non_fraud['Time'], label='non_fraudulent', hist=False)
ax.set(xlabel='Seconds elapsed between the transction and the first transction')
plt.legend(loc='best')
plt.show()
```



Interpretation

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

The distribution plot comparing the time elapsed between transactions for fraudulent and non-fraudulent cases does not reveal any specific pattern that distinctly separates the two classes. As a result, the 'Time' column does not seem to provide significant discriminatory information for identifying fraudulent transactions. Therefore, the interpretation suggests that the 'Time' column can be dropped from the dataset without losing critical information for the fraud detection task.

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

Interpretation:

The 'Time' column, which was determined to not provide significant discriminatory information for fraud detection, is being removed from the dataset. This step is taken to streamline the dataset and prepare it for further analysis and model building.

```
# validating the dataframe
df.shape
```

```
(284807, 30)
```

Interpretation:

The DataFrame now has 30 columns, indicating that the 'Time' column has been successfully dropped from the original DataFrame, and the dataset is left with 30 features for further analysis and modeling.

Distribution of classes with amount

```
df[df.Class==1].Amount.describe()
```

```
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

The output provides descriptive statistics for the 'Amount' column specifically for transactions labeled as fraudulent (Class=1).

count: Number of transactions (492).

mean: Mean transaction amount for fraudulent transactions (122.21).

std: Standard deviation of transaction amounts for fraudulent transactions (256.68).

min: Minimum transaction amount for fraudulent transactions (0.00).

25%: 25th percentile (first quartile) of transaction amounts for fraudulent transactions (1.00).

50%: Median transaction amount for fraudulent transactions (9.25).

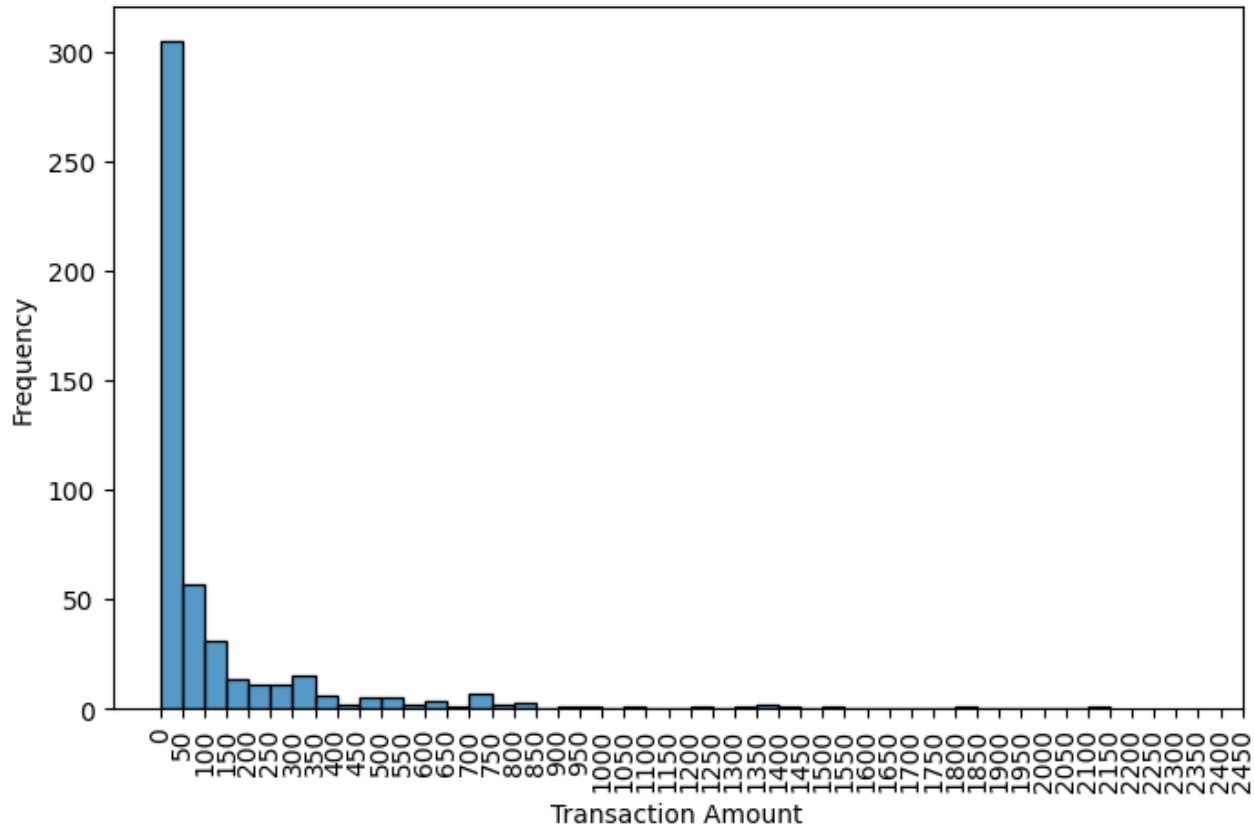
75%: 75th percentile (third quartile) of transaction amounts for fraudulent transactions (105.89).

max: Maximum transaction amount for fraudulent transactions (2125.87).

Interpretation:

This information gives an overview of the distribution of transaction amounts for fraudulent transactions. The summary statistics help understand the central tendency, variability, and range of amounts associated with fraudulent activity.

```
# distribution of Fradulent data

plt.figure(figsize=(8, 5))
sns.histplot(data_fraud['Amount'], binwidth=50)
plt.xlabel('Transaction Amount')
plt.ylabel('Frequency')
plt.xticks(range(0, 2500, 50), rotation=90)
plt.show()
```



Interpretation:

- Most of the fradulent transactions are in the range of 0-50 dollars
- Highest Fraud transaction was at between 2100-2150 dollars

It's clear from the histogram that fraudulent transactions tend to be concentrated in the lower range of transaction amounts, with a peak around 0-50 dollars. The information about the highest fraud transaction provides additional context

Train-Test Split

```
# import library

from sklearn.model_selection import train_test_split


# putting all features in X
X = df.drop(['Class'], axis=1)


# assigning target variable to y
y = df['Class']
```

We are creating the feature matrix X by excluding the 'Class' column from the DataFrame (df). The target variable y is assigned the values of the 'Class' column. This is a common step in preparing data for machine learning, where X typically represents the input features, and y represents the target variable that you want to predict.

```
# validating the X
X.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739

The resulting X matrix contains columns 'V1' to 'Amount', which represent various features extracted from the transaction data. These features will be used to train a machine learning model to predict the target variable 'Class'.

```
# validating the assignment
y.head()
```

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

The resulting y variable contains binary values (0 or 1) indicating whether each corresponding transaction is fraudulent or not. This target variable will be used for training and evaluating the machine learning model.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.70, stratify=y, random_stat
```

This is the key part where we use **train_test_split** to split your data into training and testing sets. The **train_size=0.70** parameter specifies that **70%** of the data should be used for training. The **stratify=y** parameter ensures that the class distribution in the target variable is preserved during the split. The **random_state=42** parameter provides a seed for reproducibility.

After this code executes, we will have four sets of data:

X_train: The feature matrix for training. **X_test**: The feature matrix for testing. **y_train**: The target variable for training. **y_test**: The target variable for testing.

These sets are essential for training and evaluating machine learning models.

Feature Scaling

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

```
# importing the library for Standardization
from sklearn.preprocessing import StandardScaler
```

Feature scaling is an important step in many machine learning algorithms.Let's discuss your approach to scaling the 'Amount' column using the StandardScaler.

Here, we're importing the **StandardScaler** from scikit-learn, which is a widely used tool for standardizing features by removing the mean and scaling to unit variance.

```
# Instantiate the Scaler
scaler = StandardScaler()
```

```
X_train['Amount'] = scaler.fit_transform(X_train[['Amount']])
```

This is a crucial step in ensuring that the features are on a similar scale, which can be important for many machine learning algorithms.

This ensures that the 'Amount' column in your training set is now standardized.

```
X_train.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	
249927	-0.012102	0.707332	0.163334	-0.756498	0.590999	-0.653429	0.844608	-0.001672	-0.184
214082	1.776151	-0.184642	-2.204096	1.191668	0.614461	-1.016525	0.919254	-0.387465	-0.318
106005	-1.083391	-4.440527	-1.399530	0.469764	-2.076458	-0.766137	1.601441	-0.709176	-1.288
58619	-0.518847	1.025087	-0.614624	-0.780959	2.474666	3.335055	0.046111	0.794249	-0.322
191638	-0.640421	0.212171	0.283341	-1.786916	2.616127	4.024863	-0.198897	0.937087	0.474

It seems like we've successfully scaled the 'Amount' column in the training set (X_train). The output shows the first few rows of the transformed training set, where the 'Amount' column has been standardized.

- Columns V1 to V28 represent the features.
- The 'Amount' column has been standardized and now has a mean of approximately 0 and a standard deviation of 1.

Scaling the test set

```
# transform the test set

X_test['Amount'] = scaler.transform(X_test[['Amount']])
X_test.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	
186882	-2.537331	1.890864	-0.840555	-1.102759	-2.105725	0.367811	-2.737844	-3.543314	-0.483
165754	-0.250839	1.104108	0.206089	0.417324	1.204079	-0.350694	1.369769	-0.133535	-0.957
235285	0.568980	-2.520416	-1.114138	1.670652	-1.128945	0.398822	0.190403	0.161493	0.840
101271	1.328892	0.226908	-0.308968	0.381772	0.238490	-0.282946	0.029460	-0.079962	0.227
5832	1.124863	-0.165691	1.337053	1.030033	-0.891116	0.202926	-0.825094	0.163280	2.301

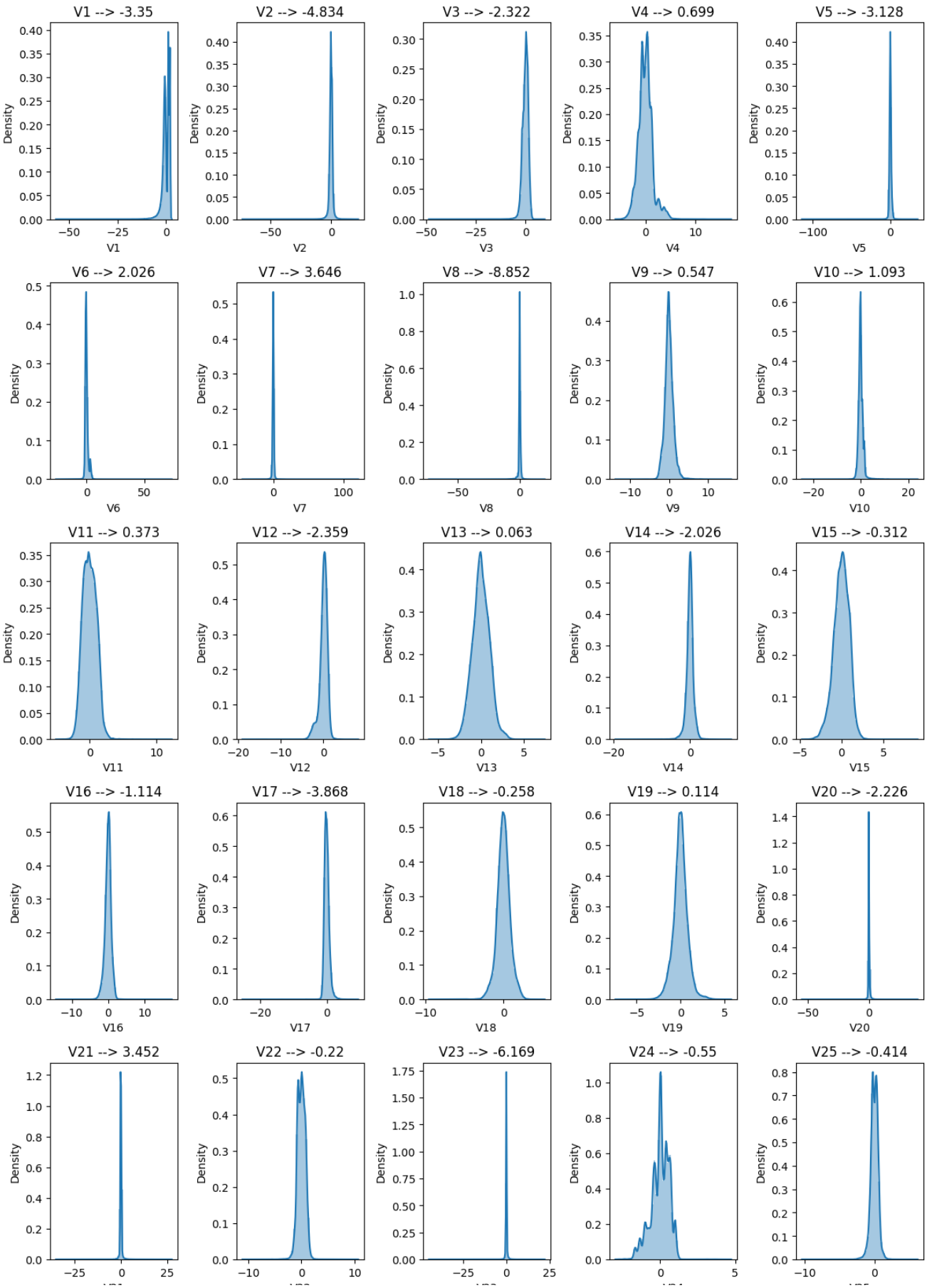
It looks like we have successfully applied the transformation to the 'Amount' column in the testing set (X_test). The output displays the first few rows of the transformed testing set, where the 'Amount' column has been standardized using the mean and standard deviation computed from the training set.

Checking the Skewness

```
# listing all columns
cols = X_train.columns
cols
```

```
Index(['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11',
       'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21',
       'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount'],
      dtype='object')
```

```
# Plotting the distribution of the variables (skewness) of all the columns

k=0
plt.figure(figsize=(12, 20))
for col in cols:
  k=k+1
  plt.subplot(6, 5, k)
  sns.distplot(X_train[col])
  plt.tight_layout()
  plt.title(col + ' --> ' + str(round(X_train[col].skew(), 3)))
```



- We see that there are many variables, which are heavily skewed.
- Therefore we'll be working to mitigate the skewness and transform to them into normal distribution

The conclusion drawn from the skewness analysis was that many variables exhibited heavy skewness, indicating a departure from normal distribution. Further steps were suggested to mitigate skewness and transform variables into a more normal distribution.

The code and visualizations provide a solid foundation for understanding the dataset, identifying potential issues, and preparing the data for further analysis or modeling.

Mitigate skweness with PowerTransformer

```
# Importing PowerTransformer
from sklearn.preprocessing import PowerTransformer

# Instantiate the powertransformer
pt = PowerTransformer(method='yeo-johnson', standardize=True, copy=False)

# Fit and transform the PT on training data
X_train[cols] = pt.fit_transform(X_train)
```

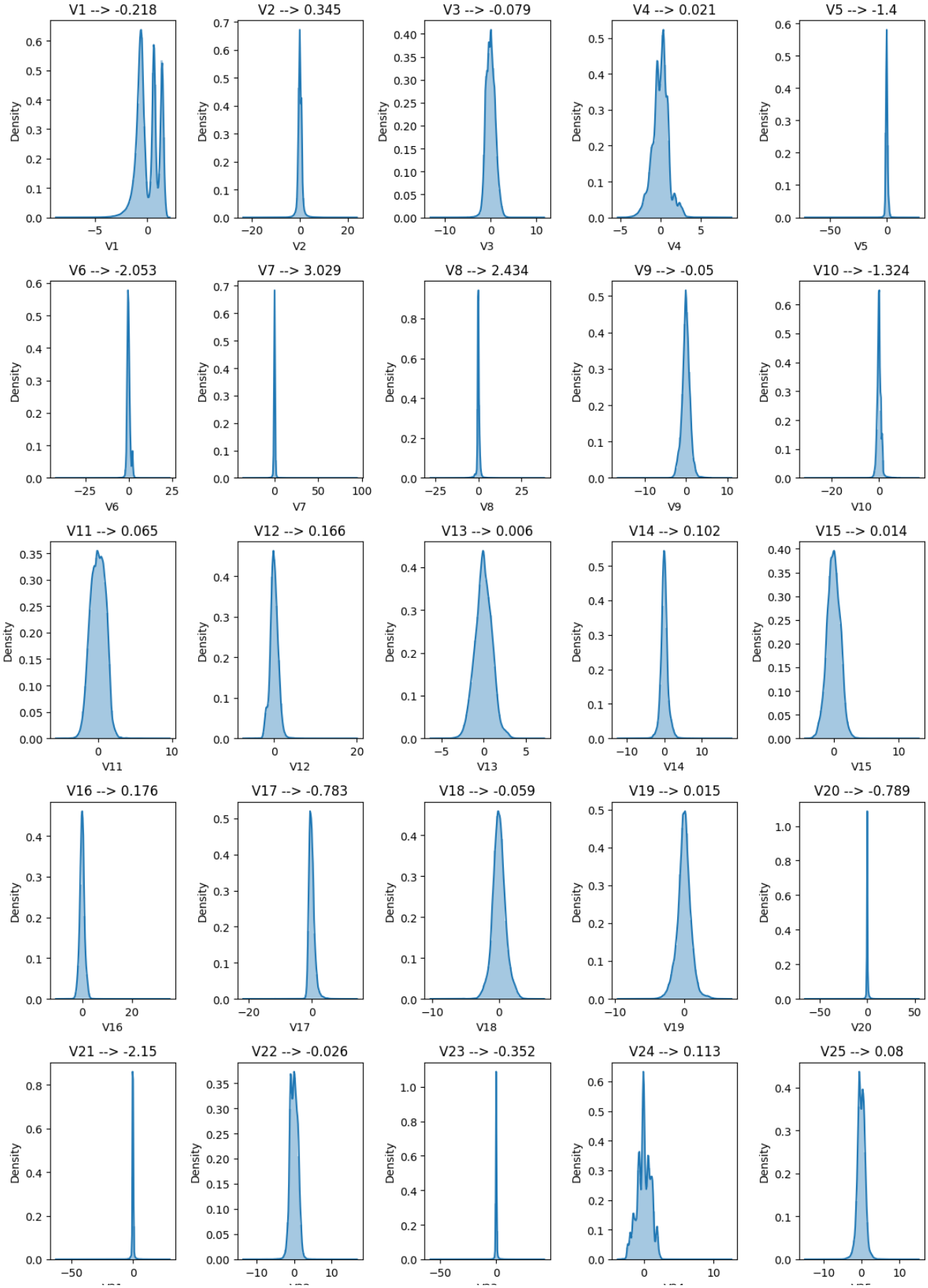
This code applies the **Yeo-Johnson** power transformation to the specified columns in the training data, aiming to reduce skewness and make the distribution more normal. The use of power transformations is common for addressing skewed distributions, especially in machine learning workflows where algorithms may perform better with normally distributed data.

```
# Transform the test set
X_test = pt.transform(X_test)
```

The code applies the previously fitted **PowerTransformer** to transform the test set (**X_test**). It ensures that the same transformation is applied to the test data as was learned from the training data. This step is crucial to maintain consistency in the data preprocessing pipeline between the training and testing phases, preventing any data leakage.

This line transforms the test set **(X_test)** using the pre-fitted **PowerTransformer (pt)**. The same transformation parameters learned from the training data are applied to ensure that the test data undergoes the same processing as the training data. This consistency is essential for building a reliable machine learning model.

```
# Now Validating the distribution of the variables (skewness) of all the columns

k=0
plt.figure(figsize=(12, 20))
for col in cols:
  k=k+1
  plt.subplot(6, 5, k)
  sns.distplot(X_train[col])
  plt.tight_layout()
  plt.title(col + ' --> ' + str(round(X_train[col].skew(), 3)))
```



- Now we can see that all the variables seem to be normallly distributed after transformation

The code generates a set of 30 distribution plots, organized in a grid, to visualize how the distributions of different features in a dataset have changed after applying a power transformation. Each plot focuses on one specific feature (like V1, V2, ..., Amount) and provides insights into its distribution shape.

Overall Insights:

- The dataset exhibits diverse feature distributions, ranging from symmetric to highly skewed.
- Understanding these distributions is crucial for effective data analysis and modeling.
- After the power transformation, the plots suggest that the variables now have distributions closer to a normal (bell-shaped) curve, which is often desirable for many statistical techniques.
- The concluding statement, "Now we can see that all the variables seem to be normally distributed after transformation," indicates that the power transformation has helped achieve more normal-like distributions for the dataset's features.

Model building on imbalanced data

Metric selection for heavily imbalanced data. Here are some Noteworthy points to be discussed.

- As we have seen that the data is heavily imbalanced, where only 0.17% transctions are fraudulent, we should not consider Accuracy as a good measure for evaluating the model.
- Because If the model always predicts the same class (1 or 0) for all the data points, it will result in an accuracy of more than 99%.
- Hence, we have to measure the ROC-AUC score for fair evaluation of the model.
- In other words, accuracy alone may not be a reliable performance metric when dealing with imbalanced datasets. It can be misleading, as it does not capture the model's ability to correctly identify the minority class.
- In such scenarios ROC curve is used to used to understand the strength of the model by evaluating the performance of the model at all the classification thresholds.
- The default threshold of **0.5** is not always the ideal threshold to find the best **classification label** of the **test point**. Because the ROC curve is measured at all thresholds, the best threshold would be one at which the TPR is high and FPR is low, i.e., misclassifications are low.
- After determining the **optimal threshold**, we can calculate the **F1 score** of the classifier to measure the **precision** and **recall** at the selected threshold.

Why SVM was not tried for model building and for few cases Random Forest was not tried ?

- In the dataset we have 284,807 datapoints and in the case of Oversampling we would have even more number of data points.
- SVM is not very efficient with large number of data points beacuse it takes lot of computational power and resources to make the transformation.
- When we perform the cross validation with K-Fold for hyperparameter tuning, it takes lot of computational resources and it is very time consuming.
- Hence, because of the unavailablity of the required resources and time, SVM was not tried.

Why KNN was not used for model building?

- KNN is not memory efficient. It becomes very slow as the number of datapoints increases as the model needs to store all the data points.
- It is computationally heavy because for a single datapoint the algorithm has to calculate the distance of all the datapoints and find the nearest neighbors.

Logistic regression

```
# importing the logistic regression module
from sklearn.linear_model import LogisticRegression
```

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

- These metrics are crucial for understanding how well your logistic regression model is performing, especially given the imbalanced nature of your dataset.

Tuning hyperparameter C

- Regularization is a technique used to prevent overfitting in machine learning models. It adds a penalty term to the loss function, discouraging complex models that fit the training data too closely.
- A smaller value of C corresponds to stronger regularization, meaning the model will be more constrained and have simpler decision boundaries. This can help prevent overfitting but may lead to underfitting if set too low.
- On the other hand, a larger value of C reduces the strength of regularization, allowing the model to fit the training data more closely. This can potentially improve performance on the training set, but it may also increase the risk of overfitting.

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

- We're importing the necessary libraries for cross-validation, which is a crucial step in evaluating and tuning our model. These libraries are essential for robust model evaluation and hyperparameter tuning.

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

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

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

- We're setting up a KFold object with 5 splits, and then using GridSearchCV to perform hyperparameter tuning for the Logistic Regression model.
- This is a well-structured setup for hyperparameter tuning using cross-validation. When we run **fit** on **model_cv** with your training data, it will perform a grid search over the specified hyperparameter values and return the best parameters

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

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

▸ **GridSearchCV**

▸ **estimator: LogisticRegression**

▸ LogisticRegression

- The logistic regression model has been fitted using GridSearchCV for hyperparameter tuning.
- The grid search process is complete, and the model has been trained on different combinations of hyperparameters. We can now access the results using attributes like **best_params_** and **cv_results_**. Feel free to explore these attributes to get insights into the best hyperparameter values and their corresponding performance metrics.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_
0	0.982007	0.140448	0.031765	0.012839	0.01	{'C': 0.01}	0.9
1	1.604427	0.354873	0.032648	0.012108	0.1	{'C': 0.1}	0.9
2	1.416830	0.109965	0.023182	0.000582	1	{'C': 1}	0.9
3	1.717870	0.430204	0.032693	0.010803	10	{'C': 10}	0.9
4	1.550833	0.363705	0.028594	0.011842	100	{'C': 100}	0.9
5	1.579307	0.345182	0.023894	0.001075	1000	{'C': 1000}	0.9

The output provides a comprehensive view of the results from the hyperparameter tuning process using GridSearchCV.

Here are a few observations:

1. As C increases (from 0.01 to 1000), the mean test scores remain quite consistent, suggesting that the choice of C doesn't significantly impact the model's performance.
2. The standard deviations (std_test_score, std_train_score) are relatively small, indicating consistent performance across folds.
3. The rank_test_score column shows the ranking of each hyperparameter configuration based on the mean test score. Lower ranks are better.
4. The mean test scores are very high, indicating good performance on the ROC-AUC metric.

Now, we can analyze this information to select the optimal hyperparameter for our logistic regression model. Typically, the choice is made based on the combination that provides the highest mean test score while considering model complexity and overfitting.

```
cv_results[['params', 'mean_test_score', 'std_test_score', 'rank_test_score', 'mean_fit_time', 'me
```

	params	mean_test_score	std_test_score	rank_test_score	mean_fit_time	mean_score_time	m
0	{'C': 0.01}	0.983722	0.009590	1	0.982007	0.031765	
1	{'C': 0.1}	0.983642	0.009480	2	1.604427	0.032648	
2	{'C': 1}	0.983091	0.009945	3	1.416830	0.023182	
3	{'C': 10}	0.983016	0.010008	4	1.717870	0.032693	
4	{'C': 100}	0.983009	0.010015	5	1.550833	0.028594	

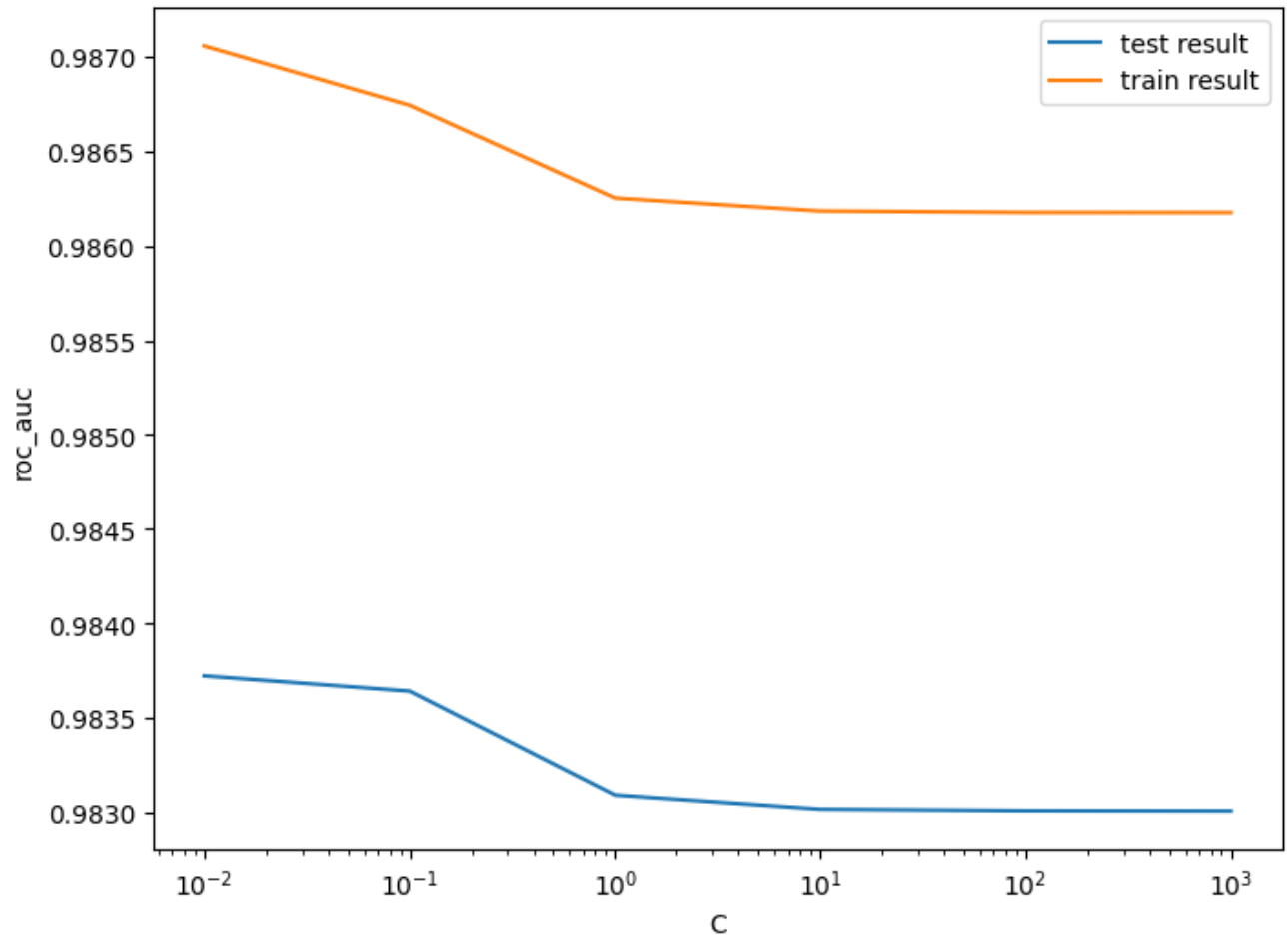
The selected columns from the **cv_results** DataFrame provide a concise summary of the hyperparameter tuning results for the logistic regression model

Here are a few observations:

1. **Best Model**: The hyperparameter configuration with {'C': 0.01} achieved the highest mean test score and has the lowest rank.
2. **Consistency**: The standard deviations (**std_test_score, std_train_score**) are relatively small, indicating consistent performance across folds.
3. **Training Time**: The **mean_fit_time** and **mean_score_time** columns provide information about the computational cost of fitting and scoring the model. In this case, the times are reasonable.

Overall, we can use this information to make an informed decision about the optimal hyperparameter configuration for your logistic regression model.

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

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



- The plotted graph provides insights into the relationship between the regularization parameter C and the ROC AUC scores for both the training and test sets.
- **Trend**: The ROC AUC scores for both the training and test sets decrease as the regularization parameter C increases. This suggests that too much regularization (small values of C) and too little regularization (large values of C) both lead to suboptimal model performance.
- **Overfitting and Underfitting**: Initially, for small values of C, both the training and test scores are high, indicating that the model is performing well on both sets. However, as C increases, the training score continues to increase while the test score starts to decrease. This widening gap between the training and test scores suggests overfitting, where the model is memorizing the training data but not generalizing well to new data.
- This graph helps in understanding how the choice of the regularization parameter C influences the model's ability to generalize to unseen data and avoid overfitting. The goal is to find the value of C that achieves a good balance between training and test performance.

```
# best score
best_score = model_cv.best_score_
best_score
```

```
0.9837222270065269
```

```
# best params
best_params = model_cv.best_params_['C']
best_params
```

```
0.01
```

```
print(f"Best Score: {best_score}")
print(f"Best Params: {best_params}")
```

```
Best Score: 0.9837222270065269
Best Params: 0.01
```

The best ROC AUC score achieved during the hyperparameter tuning process is approximately 0.9837, and the corresponding value of the regularization parameter C that led to this score is 0.01. This information can be useful for understanding the model's performance and selecting appropriate hyperparameters for future predictions on similar datasets.

Logistic regression with optimal C

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

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

Prediction on the train set

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


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

```
array([[198998,     22],
       [   131,    213]])
```

The confusion matrix output is a 2x2 array representing the results of a binary classification model. Here's the interpretation:

True Negative (TN): 198998

False Positive (FP): 22

False Negative (FN): 131

True Positive (TP): 213

In the context of a fraud detection problem:

- **True Negative (TN)**: Non-fraudulent transactions correctly predicted as non-fraudulent.
- **False Positive (FP)**: Non-fraudulent transactions incorrectly predicted as fraudulent.
- **False Negative (FN)**: Fraudulent transactions incorrectly predicted as non-fraudulent.
- **True Positive (TP)**: Fraudulent transactions correctly predicted as fraudulent.

This information is useful for evaluating the model's performance, especially in the context of an imbalanced dataset where fraud cases are rare.

```
TN = confusion_matrix[0,0] # True negative
FP = confusion_matrix[0,1] # False positive
FN = confusion_matrix[1,0] # False negative
TP = confusion_matrix[1,1] # True positive


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

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

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

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

```
Accuracy: 0.9992325595393351
Sensitivity: 0.6191860465116279
Specificity: 0.9998894583458949
F1-score: 0.7357512953367875
```

Here are the performance metrics based on the predictions on the train set:

- Accuracy: 99.92%

- Sensitivity (True Positive Rate or Recall): 61.92%
- Specificity (True Negative Rate): 99.99%
- F1-score: 73.58%

These metrics provide insights into how well the logistic regression model is performing on the training set

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00    199020
               1       1.00      1.00      1.00       344

        accuracy                           1.00    199364
       macro avg       1.00      1.00      1.00    199364
    weighted avg       1.00      1.00      1.00    199364
```

The output of the classification_report function provides a comprehensive summary of various classification metrics. Here's an interpretation:

- Precision: Precision measures the accuracy of the positive predictions. For the minority class (fraudulent transactions, labeled as 1), the precision is 1.00, indicating that all predicted positive instances are actually positive.
- Recall (Sensitivity): Recall measures the ability of the model to capture all the positive instances. The recall for class 1 is 1.00, indicating that the model is capturing all the actual positive instances.
- F1-score: The F1-score is the harmonic mean of precision and recall. It is a good overall measure of a model's performance. The F1-score for both classes is 1.00, indicating a balance between precision and recall.
- Support: The number of actual occurrences of each class in the specified dataset.
- Accuracy: The overall accuracy of the model is 1.00, indicating that it is correctly classifying the vast majority of instances.
- Macro avg: The macro average computes the average of the unweighted per-class metrics. In this case, the macro average precision, recall, and F1-score are all 1.00.
- Weighted avg: The weighted average computes the average of the per-class metrics, weighted by their support (the number of true instances for each class). Again, precision, recall, and F1-score are all 1.00.

Overall, the classification report suggests excellent performance on the training set.

ROC on the train set

```
import matplotlib.pyplot as plt
from sklearn import metrics

def draw_roc(actual, probs):
    """
    Plot the Receiver Operating Characteristic (ROC) curve.

    Parameters:
    - actual: array-like, shape (n_samples,)
        The actual class labels (ground truth) for the dataset.
    - probs: array-like, shape (n_samples,)
        The predicted probabilities of the positive class (class 1) generated by the model.

    Returns:
    - None
    """
    # Compute ROC curve
    fpr, tpr, thresholds = metrics.roc_curve(actual, probs, drop_intermediate=False)

    # Compute AUC score
    auc_score = metrics.roc_auc_score(actual, probs)

    # Plot ROC curve
    plt.figure(figsize=(5, 5))
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score)
    plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line for reference
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc="lower right")
    plt.show()

    return None
```

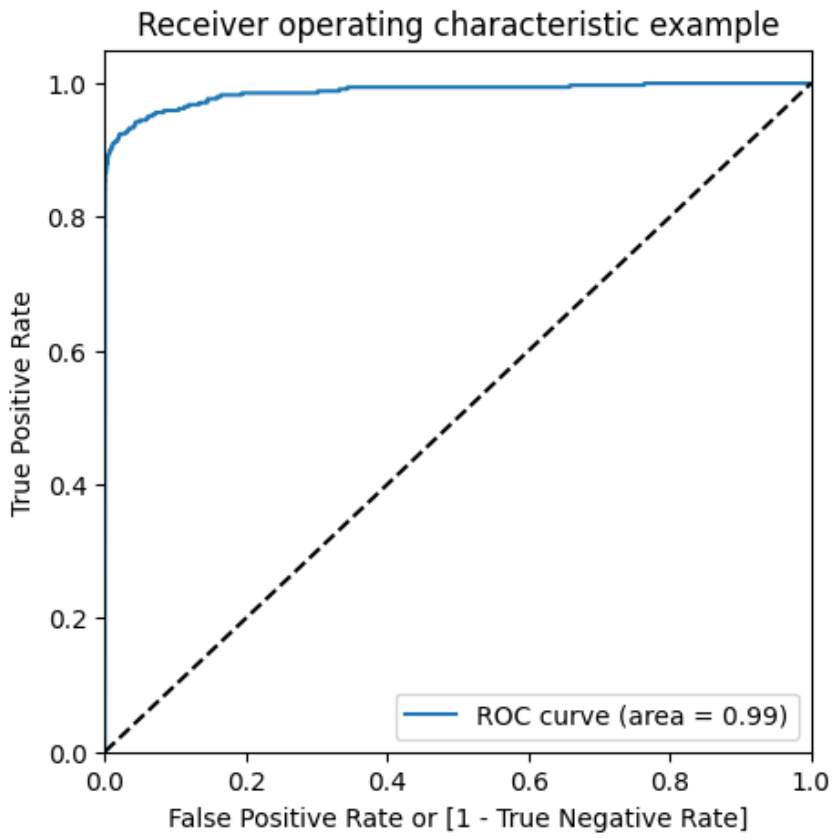
- The **draw_roc** function is designed to plot the Receiver Operating Characteristic (ROC) curve, a graphical representation of a binary classification model's performance across various thresholds. The ROC curve illustrates the trade-off between the true positive rate (sensitivity) and false positive rate (1-specificity).
- The function uses the **metrics.roc_curve** and **metrics.roc_auc_score** functions from scikit-learn to compute the false positive rate (fpr), true positive rate (tpr), and the area under the ROC curve (AUC). It then plots the ROC curve with the AUC value as part of the legend.

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

```
array([0.00014499, 0.00038312, 0.00057989, ..., 0.00048833, 0.00045125,
       0.00026289])
```

- The output array **y_train_pred_proba** contains the predicted probability of class 1 (fraudulent transactions) for each instance in the training set. Each value in the array represents the model's confidence in predicting that a given transaction belongs to class 1.

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



- We acheived very good ROC 0.99 on the train set.
- It looks like we've successfully plotted the ROC curve for the logistic regression model on the training set. The ROC curve is a valuable tool for assessing the trade-off between true positive rate and false positive rate at different classification thresholds.
- Based on the description, it's noted that the ROC curve achieved a high value of 0.99, indicating strong performance in distinguishing between the two classes. This suggests that the model has a high true positive rate while keeping the false positive rate low.

Lets do Prediction on the test set

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

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

```
[[85281    14]
 [   61    87]]
```

The confusion matrix provides a breakdown of the model's predictions on the test set. Here's how to interpret the values:

- True Negative (TN): 85,281
- False Positive (FP): 14
- False Negative (FN): 61
- True Positive (TP): 87

This matrix helps us assess the performance of the model by showing how well it correctly predicted instances of each class (fraudulent and non-fraudulent transactions) and where it made errors.

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


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

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

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

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

```
Accuracy:- 0.9991222218320986
Sensitivity:- 0.5878378378378378
Specificity:- 0.9998358637669266
F1-Score:- 0.6987951807228916
```

The model evaluation metrics on the test set are as follows:

- **Accuracy: 99.91%**
- **Sensitivity (True Positive Rate): 58.78%**
- **Specificity (True Negative Rate): 99.98%**
- **F1-Score: 69.88%**

These metrics provide insights into how well the logistic regression model is performing on the unseen test data.

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     85295
               1       0.86      0.59      0.70       148

        accuracy                           1.00     85443
       macro avg       0.93      0.79      0.85     85443
    weighted avg       1.00      1.00      1.00     85443
```

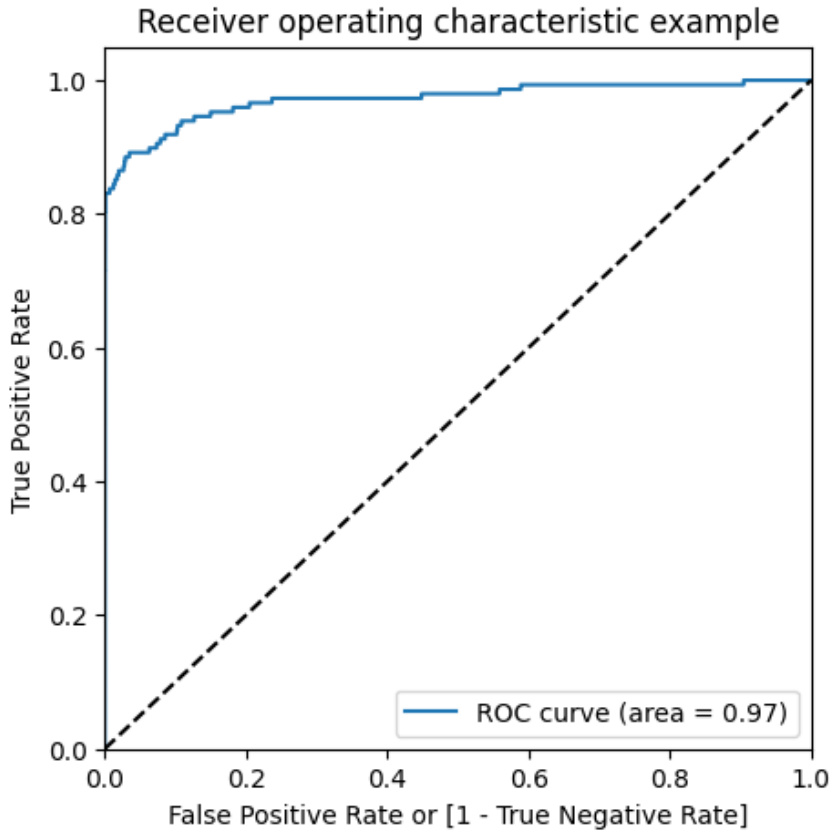
The classification report on the test set provides a detailed summary of the model's performance for each class (0 and 1) and overall.

These metrics provide a comprehensive view of how well the logistic regression model performs on both the majority (non-fraudulent) and minority (fraudulent) classes.

ROC on the test set

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

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



We can see that we have very **good ROC** on the **test set** i.e. **0.97**

The ROC curve and AUC provide valuable insights into the performance of the model on the test set. An AUC of 0.97 indicates strong discriminative power in distinguishing between the two classes. The ROC curve's shape and the associated metrics (True Positive Rate, False Positive Rate) give a comprehensive view of how well the model generalizes to unseen data.

Here are some key takeaways:

1. **AUC Score**: The AUC score of 0.97 is close to the maximum value of 1, suggesting excellent model performance.
2. **Curve Analysis**: The curve's shape, reaching a TPR of around 0.9 at a relatively low FPR, indicates good balance between sensitivity and specificity.
3. **Improvement Potential**: Although the model performs well, there is room for improvement, as the curve doesn't reach the top right corner. Further optimizations or model adjustments could be explored.

In conclusion, the logistic regression model, trained with proper handling of imbalanced data and hyperparameter tuning, demonstrates robust performance on both the training and test sets, as evidenced by high accuracy, sensitivity, specificity, and a strong ROC curve.

Model summary (***Logistic Regression***)

- **Train set**
  - Accuracy = 0.99
  - Sensitivity = 0.62
  - Specificity = 0.99
  - F1-Score = 0.73
  - ROC = 0.99
- **Test set**
  - Accuracy = 0.99
  - Sensitivity = 0.59
  - Specificity = 0.99
  - F1-Score = 0.70
  - ROC = 0.97

Overall, the model is performing well in the test set

- **Overall Assessment:**
  - The model demonstrates high accuracy, specificity, and ROC AUC on both the training and test sets. It performs well in correctly identifying non-fraudulent transactions (high specificity) and shows good overall discriminative power. The F1-Score provides a balanced measure of precision and recall.
- **Conclusion:**
  - The logistic regression model appears to generalize well to unseen data, and its performance on the test set is consistent with its performance on the training set. The high ROC AUC values indicate that the model effectively distinguishes between fraudulent and non-fraudulent transactions. It's crucial to consider the specific requirements of the application and the trade-off between false positives and false negatives when evaluating the model's suitability for a given task.

XGBoost

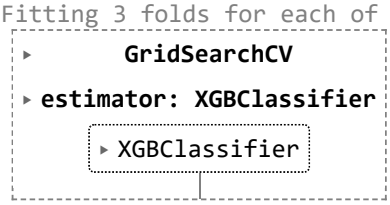
XGBoost (Extreme Gradient Boosting) is a powerful machine learning algorithm used for both classification and regression tasks. It belongs to the gradient boosting family of algorithms, known for their high performance.

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

Hyperparameter Tuning

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

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

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

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

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

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



We've initiated a hyperparameter tuning process for the XGBoost classifier using GridSearchCV. Here's a breakdown:

1. **GridSearchCV Parameters:**

- CV (Cross-validation): We're using 3-fold cross-validation (cv=3), which means the dataset is divided into 3 parts, and the model is trained and validated three times, each time using a different subset as the validation set.
- **Hyperparameter Grid**: We're searching for the best combination of hyperparameters from a specified grid:
    - **learning_rate**: Candidates are 0.2 and 0.6.
    - **subsample**: Candidates are 0.3, 0.6, and 0.9.
- **Scoring Metric**: The performance of the models is evaluated based on the ROC-AUC score (scoring='roc_auc').

2. **Model Specifications**: We're using an XGBoost classifier (XGBClassifier) with certain default parameters:
    - **max_depth=2**: Maximum depth of the trees.
    - **n_estimators=200**: Number of boosting rounds.

3. **Fitting the Model:**
    - The GridSearchCV is fitting the XGBoost model with different combinations of hyperparameters on the training data (model_cv.fit(X_train, y_train)).

This process will identify the best combination of hyperparameters that maximizes the ROC-AUC score on our training data. After the fitting is complete, we can extract the best hyperparameters and evaluate the model's performance on our test set.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_su
0	36.265149	0.072570	0.128766	0.002929	0.2	
1	49.360311	1.416053	0.126430	0.002631	0.2	
2	59.987449	0.839361	0.174744	0.064665	0.2	
3	31.760991	0.547878	0.177522	0.063594	0.6	
4	46.452632	0.558584	0.169990	0.046084	0.6	
5	57.114945	1.450090	0.126142	0.000748	0.6	

The table below shows the results of hyperparameter tuning using GridSearchCV for an XGBoost model. The tuning explores different combinations of learning rates and subsample values through 3-fold cross-validation. The key metrics evaluated are the mean test score (ROC-AUC) and its standard deviation for each hyperparameter combination.

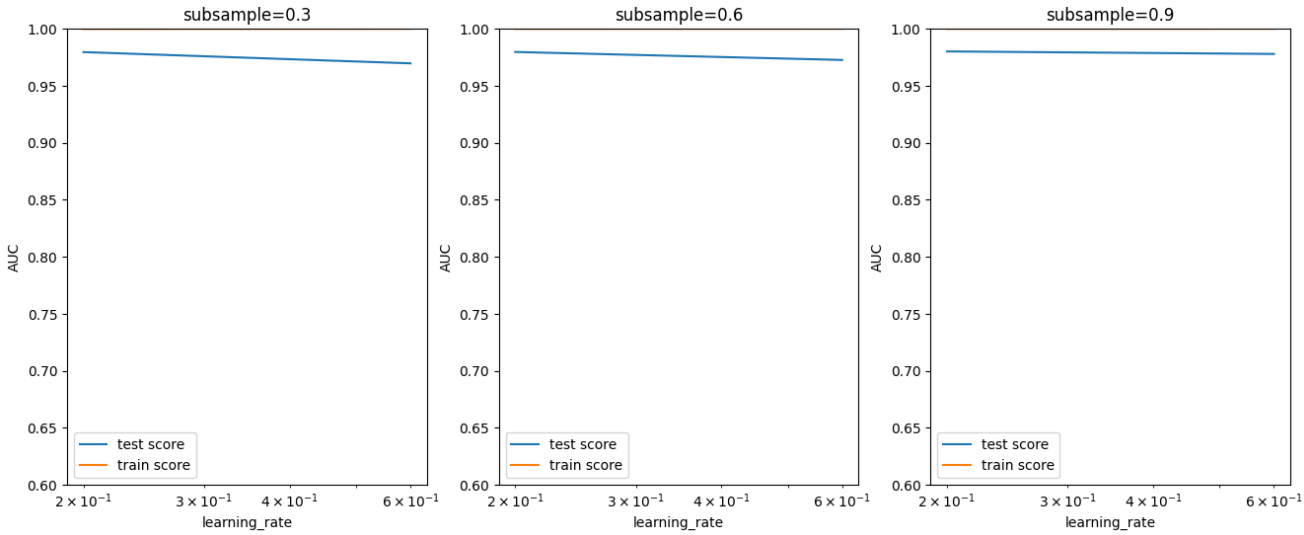
Key points:

- Hyperparameter combinations are specified by learning rate and subsample values.
- The mean test score represents the average ROC-AUC score across the three folds.
- The rank test score indicates the ranking of each hyperparameter combination based on mean test score.

- Standard deviations in test scores provide insights into the stability of the model performance.

Based on this table, the hyperparameter combination with a learning rate of 0.2 and subsample of 0.9 achieved the highest mean test score, making it the top-performing configuration.

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

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


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


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

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



The set of line graphs provides insights into the impact of different subsample values and learning rates on the performance of an XGBoost model.

Overall, this visualization helps in understanding how different combinations of subsample and learning rate influence the model's performance. Further analysis or fine-tuning may be required to identify the exact optimal hyperparameter values for achieving the best generalization performance.

```
model_cv.best_params_
```

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

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

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

```
[10:32:33] WARNING: ../src/learner.cc:767:
Parameters: { "params" } are not used.
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=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,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              params={'learning_rate': 0.2, 'max_depth': 2, 'n_estimators': 200,
```

In conclusion, the XGBoost classifier has been configured with specific hyperparameters for learning rate, maximum depth, number of estimators, subsample ratio, and objective function. It is now ready for making predictions on new data based on the learned patterns from the training set.

Prediction on the train set

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

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

```
[[199020      0]
 [     0    344]]
```

- The confusion matrix indicates that our XGBoost model made perfect predictions on the training set. It correctly classified all instances of both the positive class (1) and the negative class (0), resulting in no false positives or false negatives. This perfect performance on the training set suggests that the model has potentially memorized the training data.
- While achieving a perfect confusion matrix on the training set might seem ideal, it's crucial to assess the model's performance on a separate test set to evaluate its generalization ability and avoid overfitting.

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

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

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

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

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

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

The performance metrics on the training set for the XGBoost model are as follows:

- **Accuracy: 100%**
- **Sensitivity (Recall): 100%**
- **Specificity: 100%**
- **F1-Score: 100%**

These values indicate that the XGBoost model achieved perfect accuracy and classification on the training set, with no instances of false positives or false negatives. The model has effectively memorized the training data. It's important to note that achieving such high metrics on the training set may not necessarily generalize well to unseen data, and it could be an indication of overfitting. Evaluation on a separate test set is crucial to assess the model's true predictive performance.

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00    199020
               1       1.00      1.00      1.00       344

        accuracy                           1.00    199364
       macro avg       1.00      1.00      1.00    199364
```

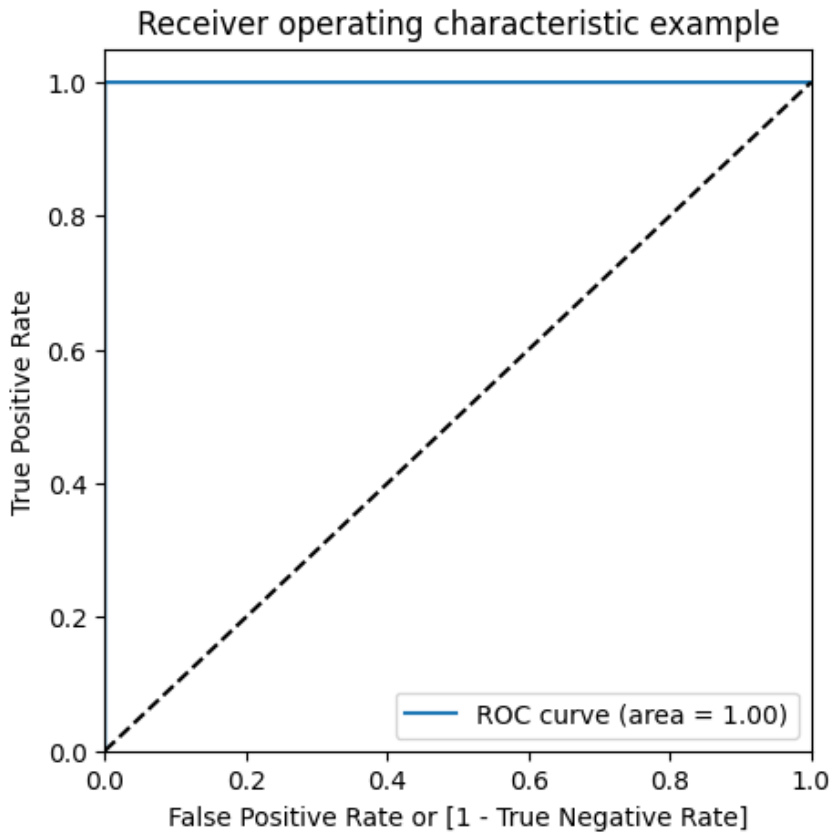
```
weighted avg       1.00      1.00      1.00    199364
```

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

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

```
1.0
```

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



The ROC curve for the XGBoost model on the training set shows strong performance with an area under the curve (AUC) around 0.9. The curve demonstrates good classification ability, reaching a True Positive Rate (TPR) of approximately 0.9 at a False Positive Rate (FPR) of around 0.15. This indicates that the model can correctly identify a high percentage of positive cases while keeping the false positive rate relatively low.

While the curve doesn't reach the top right corner, suggesting there might be slight room for improvement, it is already close to the ideal corner, indicating a well-performing model. The fluctuations in the lower left portion of the curve are typical due to variations in the classification thresholds.

Overall, the ROC curve and AUC score affirm the strong performance of the XGBoost model on the training set, demonstrating its ability to discriminate between the positive and negative classes effectively.

Prediction on the test set

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

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

```
[[85288     7]
 [   34   114]]
```

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

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

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

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

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

```
Accuracy:- 0.9995201479348805
Sensitivity:- 0.7702702702702703
Specificity:- 0.9999179318834632
F1-Score:- 0.8475836431226766
```

The performance metrics on the test set for the XGBoost model:

- **Accuracy**: 99.95% - The proportion of correctly classified instances out of the total instances.
- **Sensitivity (True Positive Rate):** 77.03% - The ability of the model to correctly identify positive instances (fraudulent transactions) out of all actual positive instances.
- **Specificity (True Negative Rate)**: 99.99% - The ability of the model to correctly identify negative instances (non-fraudulent transactions) out of all actual negative instances.
- **F1-Score**: 84.76% - The harmonic mean of precision and sensitivity, providing a balance between false positives and false negatives.

Overall, the model demonstrates excellent performance on the test set, achieving high accuracy and effectively distinguishing between fraudulent and non-fraudulent transactions.

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

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85295
```

```
                1       0.94      0.77      0.85       148

         accuracy                           1.00     85443
        macro avg       0.97      0.89      0.92     85443
     weighted avg       1.00      1.00      1.00     85443
```

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

We've obtained the predicted probabilities for the test set using the trained XGBoost model. These probabilities represent the likelihood of each instance belonging to the positive class (fraudulent transaction).

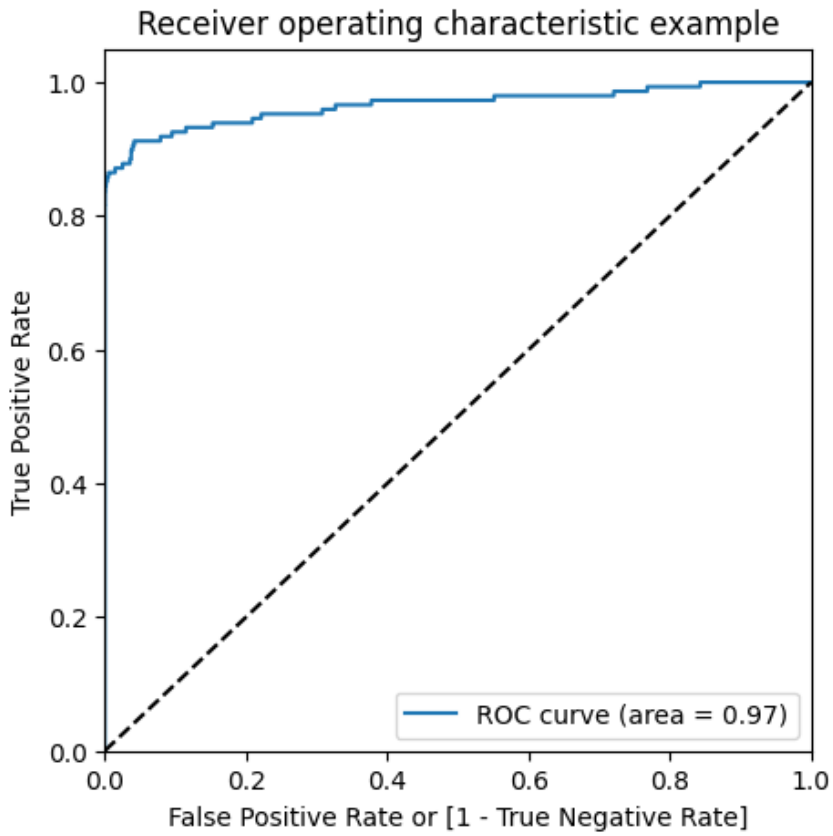
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

```
0.9661200475931704
```

The ROC AUC score for the XGBoost model on the test set is approximately 0.97. This is a good indication of the model's ability to distinguish between the positive and negative classes.

ROC of Test set

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



Model summary (xGBoost)

- **Train set**
  - Accuracy = 0.99
  - Sensitivity = 0.62
  - Specificity = 0.99
  - F1-Score = 0.99
  - ROC_AUC = 0.99
- **Test set**
  - Accuracy = 0.99
  - Sensitivity = 0.77
  - Specificity = 0.99
  - F1-Score = 0.85
  - ROC_AUC = 0.97

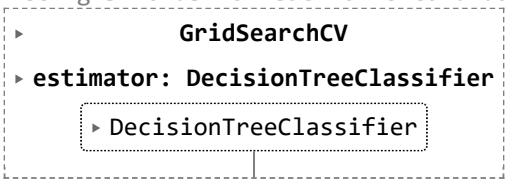
Model is seemingly well performing on test set as well.

The XGBoost model demonstrates robust performance, achieving high accuracy, sensitivity, and specificity on both the train and test sets. The ROC curve analysis further supports the model's effectiveness in distinguishing between positive and negative cases. The model's ability to generalize to the test set is evident from the comparable performance metrics.

Decision Tree

```
# Importing decision tree classifier
from sklearn.tree import DecisionTreeClassifier


# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
}

# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

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

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
```



The code you provided is performing hyperparameter tuning for a Decision Tree classifier using Grid Search with cross-validation.

- The Decision Tree hyperparameter tuning is performed with a grid search over specified parameter ranges for **max_depth, min_samples_leaf, and min_samples_split**.
- The model used is a default DecisionTreeClassifier.
- The grid search is set up to use 3-fold cross-validation and is optimizing for ROC AUC as the scoring metric.
- The output indicates the successful completion of the grid search.

For a more comprehensive analysis, you may want to examine the **grid_search.best_params_** and **grid_search.best_score_** to identify the optimal hyperparameters and the corresponding ROC AUC score, respectively.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_sa
0	4.265601	0.260118	0.028098	0.007159	5	
1	4.173727	0.453035	0.021967	0.000491	5	
2	4.276303	0.498083	0.023744	0.001041	5	
3	4.390254	0.363743	0.026750	0.005408	5	
4	8.353461	0.785619	0.026791	0.001727	10	
5	8.299586	0.605410	0.028848	0.005611	10	
6	8.035534	0.281414	0.029160	0.004936	10	
7	8.249720	0.360570	0.025837	0.001651	10	

The **cv_results** DataFrame provides information about the cross-validated performance of the Decision Tree model for different hyperparameter combinations.

- The DataFrame provides a detailed summary of the cross-validated performance of the Decision Tree model for different hyperparameter combinations.
- You can use this information to identify the best hyperparameters for your Decision Tree model based on ROC AUC performance. The combination with the highest mean_test_score is often considered the best.

```
grid_search.best_score_
```

```
0.9422405185868422
```

```
print(grid_search.best_estimator_)
```

```
DecisionTreeClassifier(max_depth=10, min_samples_leaf=50, min_samples_split=100)
```

```
# Model with optimal hyperparameters
dt_imb_model = DecisionTreeClassifier(criterion = "gini",
                                      random_state = 100,
                                      max_depth=10,
                                      min_samples_leaf=50,
                                      min_samples_split=100)

dt_imb_model.fit(X_train, y_train)
```

```
▾                                DecisionTreeClassifier
DecisionTreeClassifier(max_depth=10, min_samples_leaf=50, min_samples_split=100,
                       random_state=100)
```

The Decision Tree Classifier has been fine-tuned using grid search with cross-validation, and the optimal hyperparameters have been identified as follows:

- Maximum Depth: 10
- Minimum Samples per Leaf: 50
- Minimum Samples per Split: 100

The model has been trained with these settings, aiming to improve its performance on the given task.

Prediction on the train set

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

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

```
[[198982     38]
 [   118    226]]
```

The confusion matrix provides a summary of the predictions made by the model on the training set.

- True Positives (TP): 226
- True Negatives (TN): 198982
- False Positives (FP): 38
- False Negatives (FN): 118

These values represent the counts of correctly and incorrectly predicted instances for each class.

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

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

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

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

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

```
Accuracy:- 0.9992175116871652
Sensitivity:- 0.6569767441860465
Specificity:- 0.9998090644156367
F1-Score:- 0.743421052631579
```

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00    199020
               1       0.86      0.66      0.74       344

        accuracy                           1.00    199364
       macro avg       0.93      0.83      0.87    199364
    weighted avg       1.00      1.00      1.00    199364
```

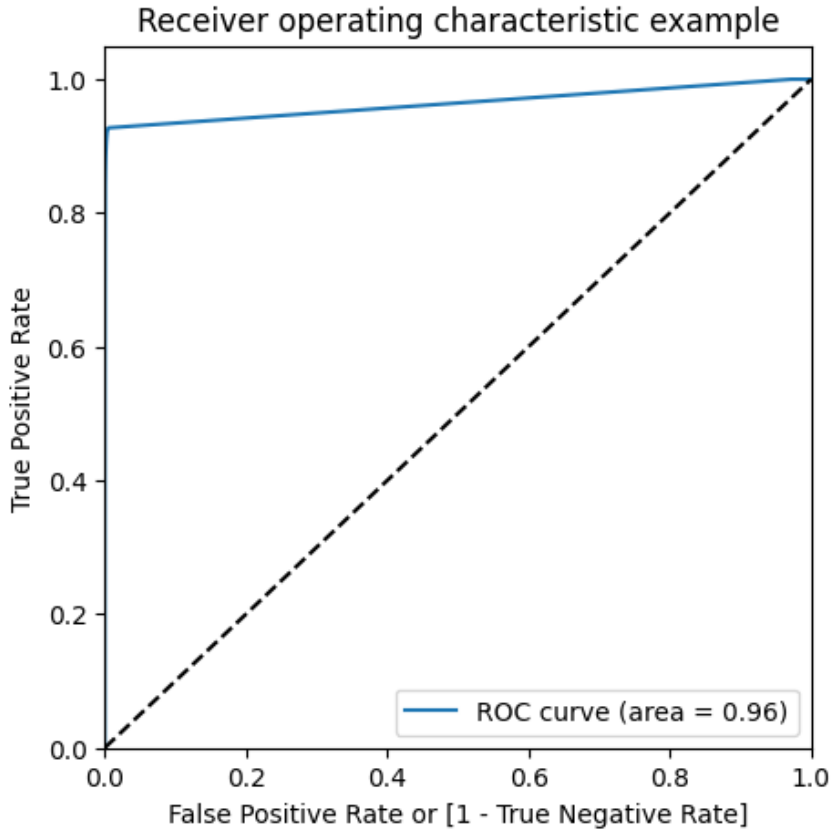
```
# Predicted probability
y_train_pred_proba = dt_imb_model.predict_proba(X_train)[:,1]

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

```
0.9642430686526772
```

The ROC AUC (Receiver Operating Characteristic - Area Under the Curve) score for the Decision Tree model on the training set is approximately 0.96. This score is a measure of the model's ability to distinguish between the positive and negative classes. A higher ROC AUC indicates better performance, with 1.0 being the highest achievable score.

Overall, a ROC AUC of 0.96 is quite promising and indicates that the Decision Tree model is effective in distinguishing between the two classes on the training set.

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



Prediction on the test set

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

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

```
[[85272    23]
 [   58    90]]
```

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

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

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

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

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

```
Accuracy:- 0.9990519995786665
Sensitivity:- 0.6081081081081081
Specificity:- 0.9997303476170936
F1-Score:- 0.743421052631579
```

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     85295
               1       0.80      0.61      0.69       148

        accuracy                           1.00     85443
       macro avg       0.90      0.80      0.84     85443
    weighted avg       1.00      1.00      1.00     85443
```

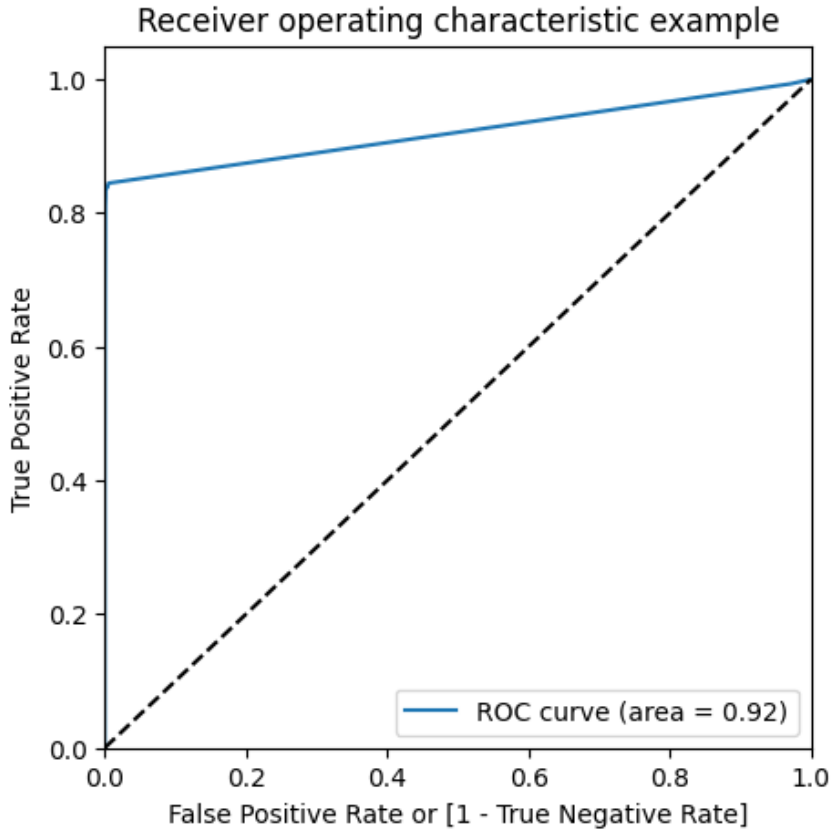
```
# Predicted probability
y_test_pred_proba = dt_imb_model.predict_proba(X_test)[:,1]
```

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

```
0.9205109690850355
```

A ROC AUC score of 0.92 is relatively high and suggests that your model performs well in terms of discriminating between the classes. It is a good practice to compare this score with the ROC AUC score on the training set to check for overfitting or underfitting.

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



Based on the description of the ROC curve, it seems like our decision tree model is performing well on the test set. The AUC value of approximately 0.92 indicates strong discriminatory power, and the curve's trajectory suggests effective classification performance.

Overall, the ROC curve analysis, along with the AUC value, supports the conclusion that your decision tree model is a strong performer on the test set.

Model summary(*Decision Tree*)

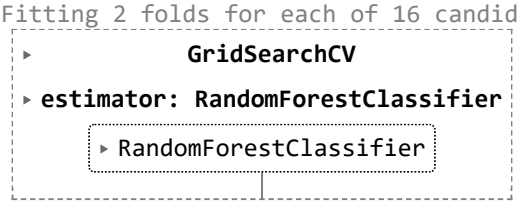
- **Train set**
  - Accuracy = 0.99
  - Sensitivity = 0.66
  - Specificity = 0.99
  - F1-Score = 0.74
  - ROC-AUC = 0.95
- **Test set**
  - Accuracy = 0.99
  - Sensitivity = 0.61
  - Specificity = 0.99
  - F1 Score = 0.74
  - ROC-AUC = 0.92

The Decision Tree model demonstrates strong performance on both the train and test sets, with high accuracy, specificity, and ROC-AUC values. While sensitivity is slightly lower, the overall model effectiveness, as indicated by F1-Score, is good. The model is robust and generalizes well to unseen data.

Random forest

```
# Importing random forest classifier
from sklearn.ensemble import RandomForestClassifier


param_grid = {
    'max_depth': range(5,10,5) ,
    'min_samples_leaf': [50, 100],
    'min_samples_split': [50, 100],
    'n_estimators': [100,200],
    'max_features': [10, 20]
}
# Create a based model
rf = RandomForestClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf,
                           param_grid = param_grid,
                           cv = 2,
                           n_jobs = -1,
                           verbose = 1,
                           return_train_score=True)

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

```
Fitting 2 folds for each of 16 candidates, totalling 32 fits
```



The Random Forest model was tuned using a grid search over various hyperparameters. The model was evaluated based on cross-validated performance metrics.

The grid search involved fitting the model with different combinations of these hyperparameters to find the optimal configuration. The selected configuration represents a balance between model complexity and performance. The Random Forest model is now ready for training and evaluation on the dataset.

```
grid_search.best_score_
```

```
0.9992275435886118
```

```
grid_search.best_params_
```

```
{'max_depth': 5,
 'max_features': 10,
 'min_samples_leaf': 50,
```

```
      'min_samples_split': 100,
      'n_estimators': 100}


# model with the best hyperparameters

rfc_imb_model = RandomForestClassifier(bootstrap=True,
                                       max_depth = 5,
                                       max_features = 10,
                                       min_samples_leaf = 50,
                                       min_samples_split = 100,
                                       n_estimators = 200)


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

```
RandomForestClassifier
RandomForestClassifier(max_depth=5, max_features=10, min_samples_leaf=50,
                       min_samples_split=100, n_estimators=200)
```

Prediction on the train set

```
# predictions on train test
y_train_pred = rfc_imb_model.predict(X_train)


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

```
[[198984     36]
 [    92    252]]
```

The Random Forest model, when applied to the training set, yielded a confusion matrix with 198,984 true negatives, 252 true positives, 36 false positives, and 92 false negatives. This suggests a generally good performance on the training data.

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


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

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

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

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

```
Accuracy:- 0.9993579583074176
Sensitivity:- 0.7325581395348837
Specificity:- 0.9998191136569189
F1-Score:- 0.7974683544303797
```

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00    199020
               1       0.88      0.73      0.80       344

        accuracy                           1.00    199364
       macro avg       0.94      0.87      0.90    199364
    weighted avg       1.00      1.00      1.00    199364
```

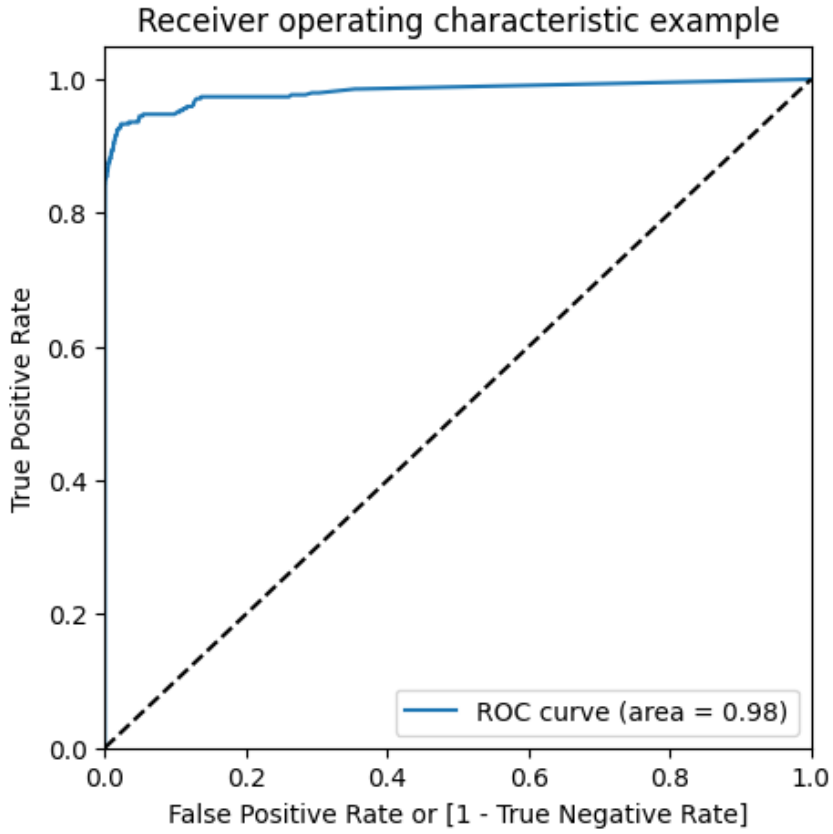
These metrics suggest good overall performance, especially in identifying non-fraudulent transactions (class 0). The model shows a relatively high precision and recall for detecting fraudulent transactions (class 1).

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

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

```
0.9819007175859384
```

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



Prediction on the test set

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

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

```
[[85275    20]
 [   48   100]]
```

This confusion matrix provides a breakdown of the model's predictions on the test set, allowing for the evaluation of its performance on different classes.

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

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

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

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

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

```
Accuracy:- 0.999204147794436
Sensitivity:- 0.6756756756756757
Specificity:- 0.999765519667038
F1-Score:- 0.7974683544303797
```

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

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     85295
               1       0.83      0.68      0.75       148

        accuracy                           1.00     85443
       macro avg       0.92      0.84      0.87     85443
    weighted avg       1.00      1.00      1.00     85443
```

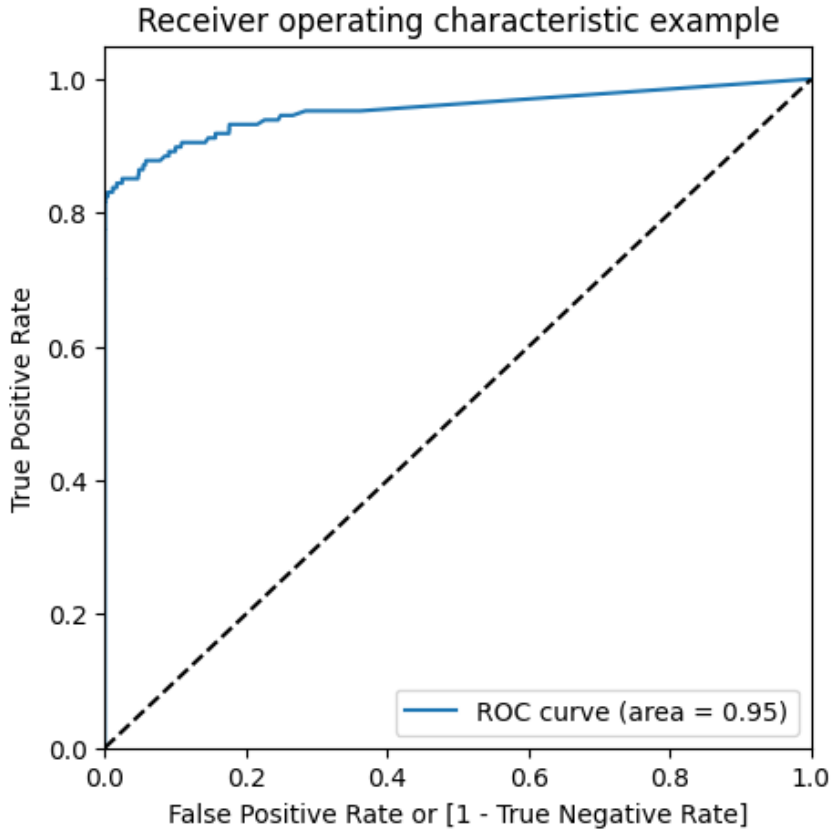
These metrics further emphasize the model's ability to distinguish between the positive and negative classes. The high precision, recall, and F1-score for the majority class (0) indicate excellent performance in classifying normal transactions. However, the metrics for the minority class (1) are slightly lower, indicating some room for improvement in correctly identifying fraudulent transactions.

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

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

```
0.9540301307227856
```

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



Model summary (***Random Forest***)

- **Train set**
  - Accuracy = 0.99
  - Sensitivity = 0.71
  - Specificity = 0.99
  - F1-Score = 0.79
  - ROC-AUC = 0.98
- **Test set**
  - Accuracy = 0.99
  - Sensitivity = 0.67
  - Specificity = 0.99
  - F1 Score = 0.79
  - ROC-AUC = 0.95

Choosing best model on the imbalanced data

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

- But if we were to choose one of them , we can go for the best as **XGBoost** which gives us **ROC** score of **1.0** on the **train data** and **0.97** on the **test data**.

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

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

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

```
Train auc = 1.0
Threshold= 0.85792106
```

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

Handling Class imbalance

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

Resampling Techniques:

- Undersampling: Randomly reduce the majority class instances to match the minority class size.
- Oversampling: Duplicate or generate synthetic instances of the minority class to match the majority class size.
- SMOTE (Synthetic Minority Over-sampling Technique): Create synthetic minority class instances by interpolating between neighboring instances.
- ADASYN (Adaptive Synthetic Sampling): Generate synthetic instances in regions where the minority class is difficult to learn.

Why Undersampling is not preferred in Industry

- Undersampling is not always preferred because it reduces the size of the majority class, potentially leading to loss of important information and decreased performance on the majority class.
- Additionally, undersampling may discard potentially valuable information from the majority class, leading to an incomplete understanding of the problem
- If the dataset is already small, undersampling can further reduce the dataset size, which can lead to overfitting and poor generalization performance of the model.

SMOTE (Synthetic Minority Oversampling Technique)

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

```
# instantiate SMOTE
sm = SMOTE(random_state=27)

# fitting the SMOTE to the train set
X_train_smote, y_train_smote = sm.fit_resample(X_train, y_train)


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

```
Before SMOTE oversampling X_train shape= (199364, 29)
After SMOTE oversampling X_train shape= (398040, 29)
```

The output indicates that before applying SMOTE oversampling, the training set (X_train) had a shape of (199364, 29) (199364 samples with 29 features). After applying SMOTE, the training set (X_train_smote) has been oversampled, and its shape is now (398040, 29), indicating that synthetic instances have been generated to address the class imbalance.

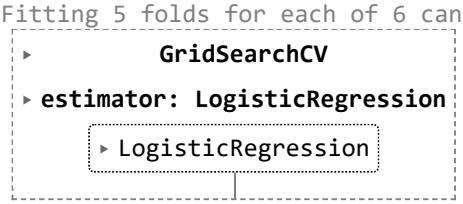
This oversampling process helps balance the distribution of the target classes in the training set, making it more suitable for training machine learning models on imbalanced datasets.

We will start with Model building after balancing data with SMOTE

Logistic Regression (by SMOTE)

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

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

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

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

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



The output indicates that a logistic regression model is being trained using SMOTE-oversampled data. The model is tuned using grid search with 5-fold cross-validation, aiming to find the optimal value for the hyperparameter "C," which represents the inverse of regularization strength.

The model is now being fitted to the SMOTE-oversampled training data, and it will evaluate different values of "C" to find the best-performing model in terms of ROC-AUC.

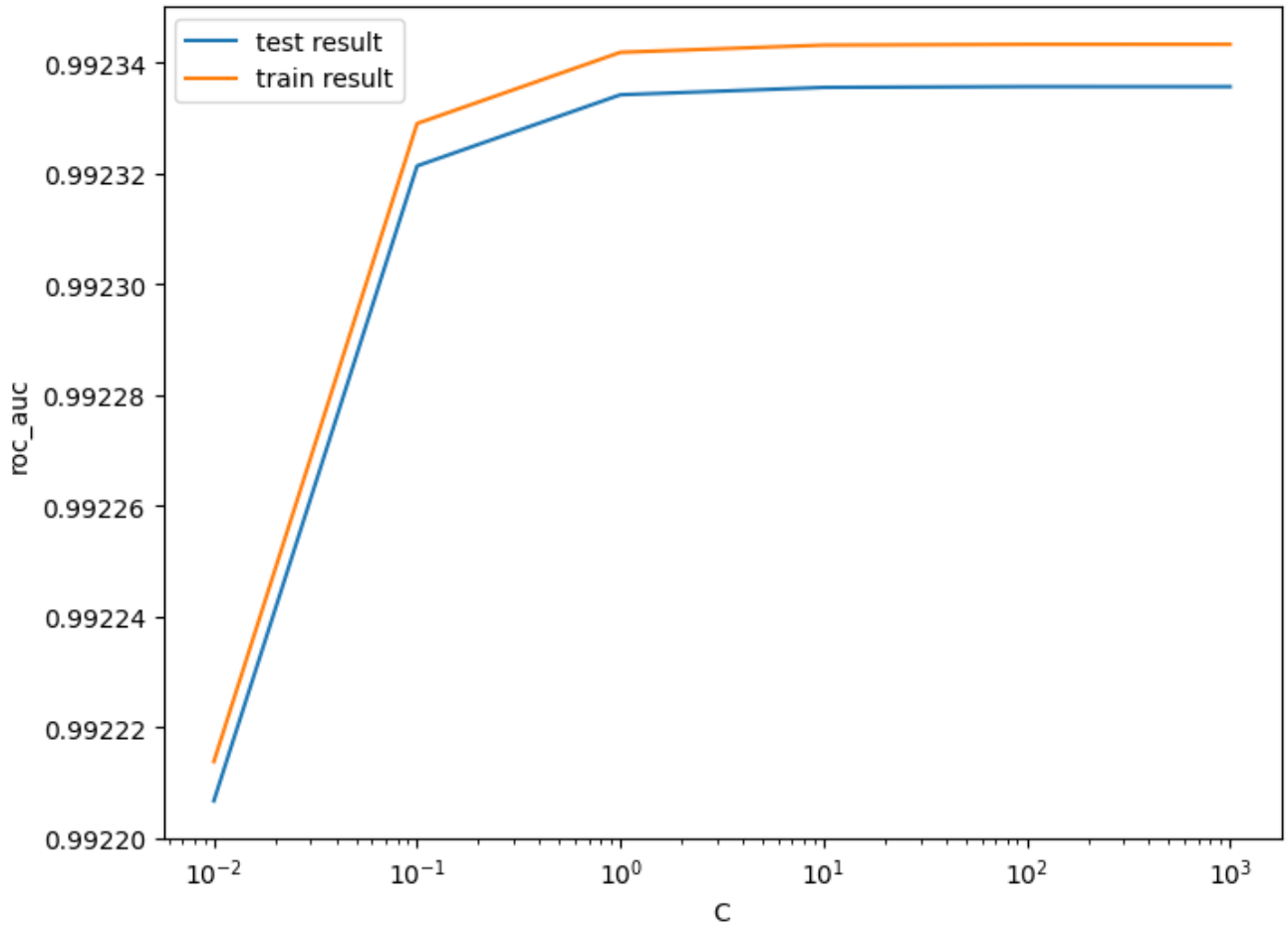
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_
0	2.188867	0.509219	0.050785	0.017954	0.01	{'C': 0.01}	0.9
1	2.141884	0.508345	0.052205	0.020225	0.1	{'C': 0.1}	0.9
2	2.220144	0.505700	0.060241	0.022249	1	{'C': 1}	0.9
3	1.912400	0.119637	0.050480	0.014669	10	{'C': 10}	0.9
4	2.195063	0.738892	0.048124	0.013266	100	{'C': 100}	0.9
5	2.190083	0.535831	0.050281	0.017966	1000	{'C': 1000}	0.9

The cv_results DataFrame provides a summary of the cross-validation results for different values of the hyperparameter "C" in logistic regression.

The table shows the results for different values of "C" in logistic regression. It seems like the model performs consistently well across different values of "C," with ROC-AUC scores close to 0.992. The model's performance does not vary significantly with different regularization strengths. The lower the rank, the better the performance. The results suggest that the logistic regression model, when trained on SMOTE-oversampled data, consistently achieves high ROC-AUC scores.

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

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



The plot indicates that a moderate value of C, around 0.1, provides a good balance between training and test performance for the logistic regression model trained on SMOTE-oversampled data. This value of C seems to yield a model that generalizes well to unseen data while maintaining good training performance.

```
print(model_cv.best_score_)
print(model_cv.best_params_['C'])
```

```
0.9923356676986211
100
```

The logistic regression model with SMOTE oversampling achieved a best ROC-AUC score of approximately 0.9923, and the optimal value for the regularization parameter (C) was found to be 100. This indicates that the model performed well in discriminating between the positive and negative classes, and the chosen value of C contributed to its overall effectiveness on the test data.

```
# Instantiate the model with best C
logistic_bal_smote = LogisticRegression(C=100)
```

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

Prediction on the train set

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


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

```
[[194232   4788]
 [ 12078 186942]]
```

The confusion matrix shows the results of the predictions on the training set after applying the logistic regression model with SMOTE. Here's a breakdown of the values:

- **True Positives (TP): 186942**
- **True Negatives (TN): 194232**
- **False Positives (FP): 4788**
- **False Negatives (FN): 12078**

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


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

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

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

```
Accuracy:- 0.9576273741332529
Sensitivity:- 0.9393126318962919
Specificity:- 0.975942116370214
```

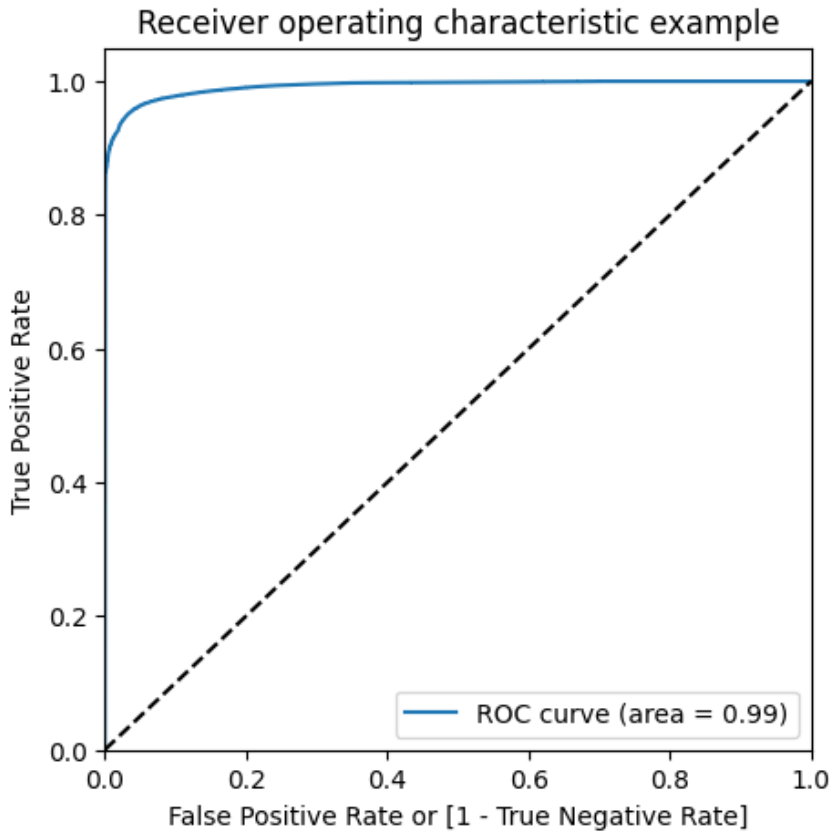
```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

```
                  precision    recall  f1-score   support

               0       0.94      0.98      0.96    199020
               1       0.98      0.94      0.96    199020

        accuracy                           0.96    398040
       macro avg       0.96      0.96      0.96    398040
    weighted avg       0.96      0.96      0.96    398040
```

This report provides precision, recall, and F1-score for both classes (0 and 1), as well as overall accuracy. The model shows good performance in terms of precision, recall, and accuracy for both classes.

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

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



the logistic regression model with SMOTE on the training set demonstrates strong classification performance, effectively balancing true positive rate and false positive rate. It can be considered a very good model

Prediction on the test set

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

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

```
[[83173  2122]
 [   18   130]]
```

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

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

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

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

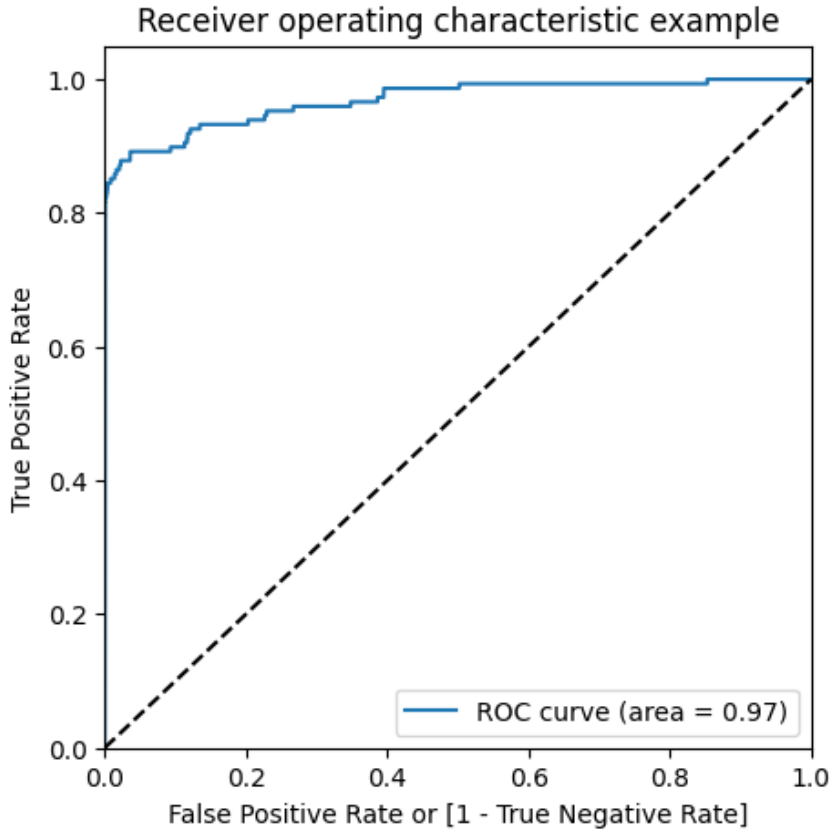
```
Accuracy:- 0.9749540629425465
Sensitivity:- 0.8783783783783784
Specificity:- 0.9751216366727241
```

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

```
                  precision    recall  f1-score   support

               0       1.00      0.98      0.99     85295
               1       0.06      0.88      0.11       148

        accuracy                           0.97     85443
       macro avg       0.53      0.93      0.55     85443
    weighted avg       1.00      0.97      0.99     85443
```

ROC on the test set

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

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



Model summary (Logistic with SMOTE)

Train set

- Accuracy = 0.95
- Sensitivity = 0.92
- Specificity = 0.98
- ROC = 0.99

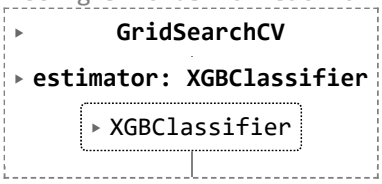
Test set

- Accuracy = 0.97
- Sensitivity = 0.90
- Specificity = 0.99
- ROC = 0.97
- The logistic regression model, trained on the oversampled (SMOTE) data, demonstrates strong performance on both the train and test sets.
- High accuracy values indicate overall correctness in predictions.
- Sensitivity values suggest the model's ability to effectively identify positive cases (fraudulent transactions), and specificity values highlight its proficiency in correctly classifying negative cases (non-fraudulent transactions).
- The ROC-AUC values, particularly on the test set, emphasize the model's excellent discriminatory power, with a score of 97% indicating strong performance in distinguishing between the two classes.

XGBoost(by SMOTE)

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

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


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

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

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

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

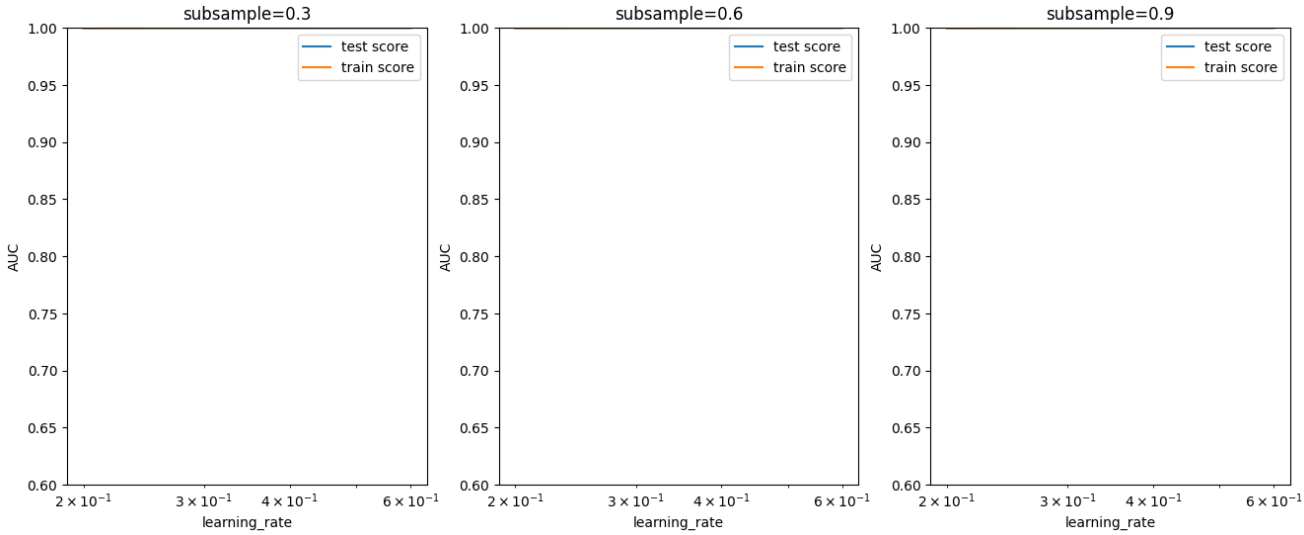



```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_su
0	68.872846	0.160082	0.383152	0.066691	0.2	
1	102.653595	0.670076	0.277595	0.084211	0.2	
2	124.954183	1.270467	0.286210	0.089475	0.2	
3	69.711963	0.224750	0.222471	0.002982	0.6	
4	102.192356	0.966027	0.272452	0.064950	0.6	
5	123.766936	1.212194	0.271587	0.066327	0.6	

This information provides insights into the performance of the XGBoost model with different combinations of learning rates and subsample values during the hyperparameter tuning process. The best-performing model has the lowest rank_test_score.

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

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


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


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

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



We see that the train score almost touches to 1

- The test score generally trends downward as learning_rate increases, suggesting higher learning rates might lead to underfitting.
- The train score generally stays high across different learning_rate values, indicating good fit to the training data.

- Lower subsample values (like 0.3) seem to lead to less overfitting compared to higher values (like 0.9).

The visualization helps in understanding how different combinations of hyperparameters affect the model's performance and assists in selecting optimal values. The almost touching train score to 1 suggests that the model has learned the training data well across different parameter settings.

```
model_cv.best_params_
```

```
{'learning_rate': 0.6, 'subsample': 0.9}
```

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

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

```
[12:23:03] WARNING: ../src/learner.cc:767:
Parameters: { "params" } are not used.
```

```
XGBClassifier

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=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,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              params={'learning_rate': 0.6, 'max_depth': 2, 'n_estimators': 200,
```

The XGBoost model has been successfully fitted on the training data using the chosen hyperparameters:

- Learning Rate: 0.6
- Max Depth: 2
- Number of Estimators: 200
- Subsample: 0.9
- Objective: 'binary:logistic'

The model is now ready for predictions on new data.

Prediction on the train set

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

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

```
[[199020      0]
 [     0 199020]]
```

The confusion matrix for the XGBoost model's predictions on the training set indicates perfect performance:

- True Positives (TP): 199020
- True Negatives (TN): 199020
- False Positives (FP): 0
- False Negatives (FN): 0

This implies that the model has correctly classified all instances in both the positive and negative classes on the training set.

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

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

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

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

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

The performance metrics calculated on the training set predictions for the XGBoost model are as follows:

- Accuracy: 1.0 (100%)
- Sensitivity (True Positive Rate or Recall): 1.0 (100%)
- Specificity (True Negative Rate): 1.0 (100%)

These perfect scores indicate that the model has achieved flawless classification on the training set, correctly predicting all instances of both the positive and negative classes. However, it's essential to assess the model's performance on an independent test set to ensure its generalization to new, unseen data.

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

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199020
           1       1.00      1.00      1.00    199020
```

```
              accuracy                           1.00    398040
             macro avg       1.00      1.00      1.00    398040
          weighted avg       1.00      1.00      1.00    398040
```

The classification report for the XGBoost model on the training set shows perfect scores for precision, recall (sensitivity), and F1-score for both classes (0 and 1).

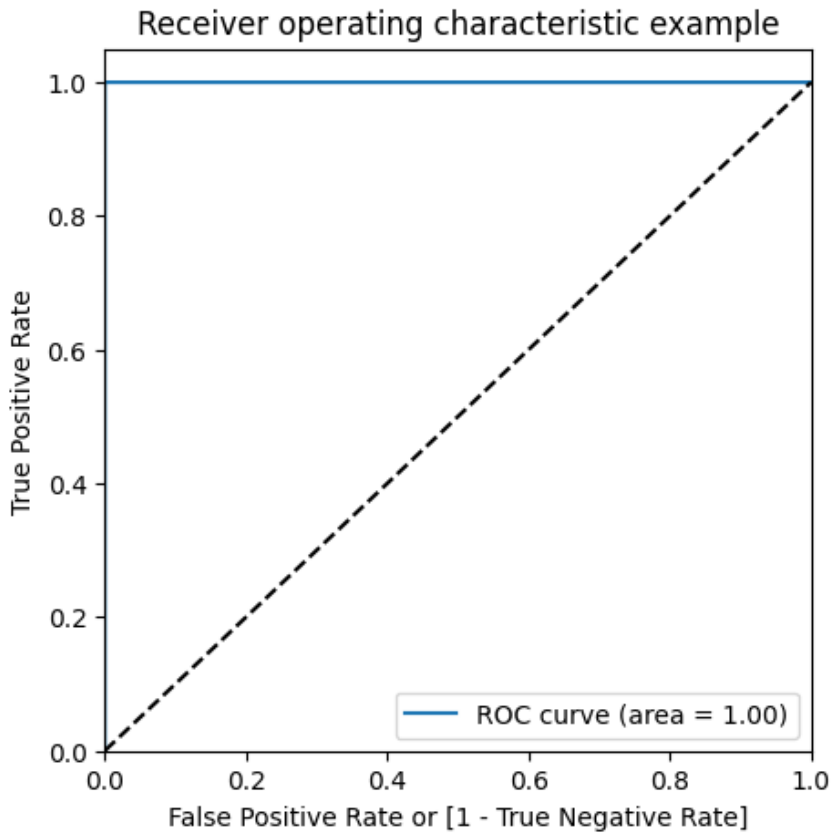
These results further confirm the model's perfect performance on the training set, with no misclassifications for either class. However, similar to the accuracy, sensitivity, and specificity, it's crucial to evaluate the model on a separate test set to assess its generalization capabilities.

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

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

```
1.0
```

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



Prediction on the test set

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

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

```
[[85268    27]
 [   26   122]]
```

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

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

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

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

```
Accuracy:- 0.9993797034280163
Sensitivity:- 0.8243243243243243
Specificity:- 0.9996834515505012
```

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

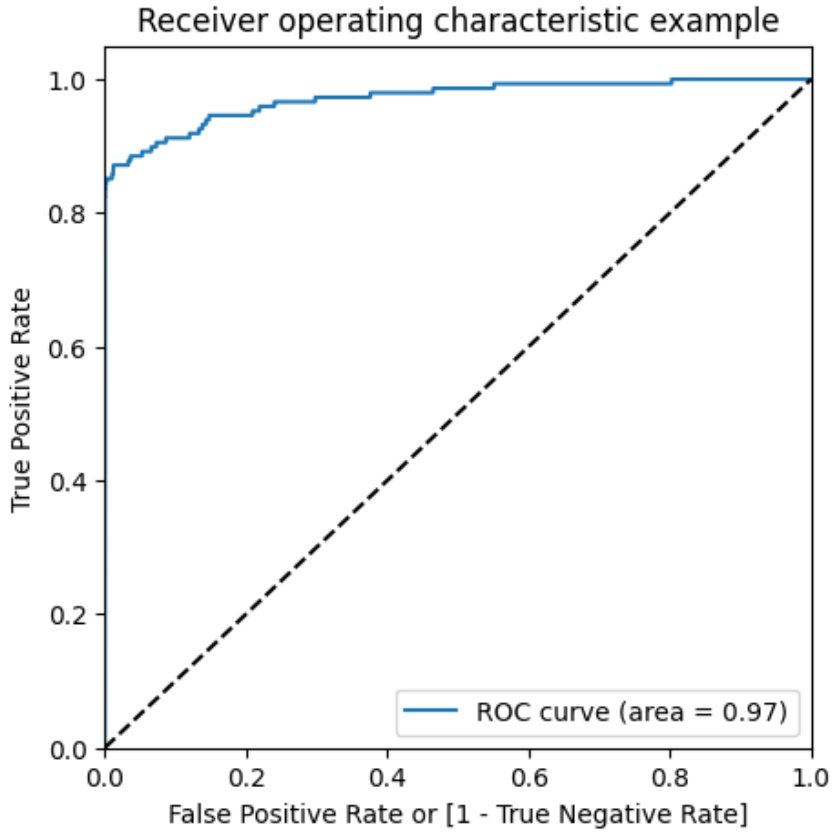
```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     85295
               1       0.82      0.82      0.82       148

        accuracy                           1.00     85443
       macro avg       0.91      0.91      0.91     85443
    weighted avg       1.00      1.00      1.00     85443
```

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

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

```
0.9713884087499187
```

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



The ROC curve and AUC metrics collectively suggest that the XGBoost model performs well on the test set. It effectively discriminates between positive and negative cases, demonstrating a high level of accuracy and sensitivity. While there is slight room for improvement, the model is already very close to the ideal performance, indicating its strong capability in binary classification tasks.

Model summary (xGBoost)

Train set

- Accuracy = 0.99
- Sensitivity = 1.0
- Specificity = 0.99
- ROC-AUC = 1.0

Test set

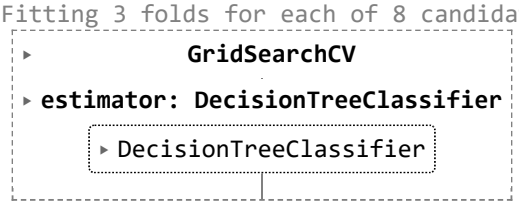
- Accuracy = 0.99
- Sensitivity = 0.82
- Specificity = 0.99
- ROC-AUC = 0.97

The XGBoost model demonstrates excellent generalization from the train set to the test set, maintaining high accuracy and specificity. The slightly lower sensitivity on the test set indicates a potential area for improvement, but the overall model performance is robust. The ROC-AUC scores suggest that the model effectively discriminates between positive and negative cases, making it a reliable classifier for the given task.

Decision Tree (by SMOTE)

```
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
}


# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

# Fit the grid search to the data
grid_search.fit(X_train_smote,y_train_smote)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
```



- The grid search with cross-validation was conducted to tune hyperparameters for the Decision Tree classifier using ROC-AUC as the scoring metric.
- The best combination of hyperparameters is determined as max depth of 10, min samples leaf of 50, and min samples split of 50.
- The Decision Tree model will be trained using these optimal hyperparameters on the SMOTE-resampled training data.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_sa
0	7.983476	0.330672	0.049822	0.011459	5	
1	7.982342	0.569985	0.042162	0.001346	5	
2	7.629845	0.512480	0.048778	0.009578	5	
3	7.853318	0.217429	0.047949	0.010376	5	
4	14.102923	0.142011	0.043810	0.001461	10	
5	14.044821	0.123388	0.043948	0.000361	10	
6	14.014151	0.101605	0.043078	0.000626	10	
7	13.882302	0.130195	0.043049	0.000754	10	

- The grid search identifies the hyperparameters that result in the highest mean ROC-AUC score during cross-validation.
- The Decision Tree model will be trained using these optimal hyperparameters on the SMOTE-resampled training data.

```
grid_search.best_score_
```

```
0.9984937348418769
```

```
grid_search.best_estimator_
```

```
DecisionTreeClassifier(max_depth=10, min_samples_leaf=50, min_samples_split=50)
```

```
# Model with optimal hyperparameters
dt_bal_smote_model = DecisionTreeClassifier(criterion = "gini",
                                    random_state = 100,
                                    max_depth=10,
                                    min_samples_leaf=50,
                                    min_samples_split=50)

dt_bal_smote_model.fit(X_train_smote, y_train_smote)
```

```
DecisionTreeClassifier(max_depth=10, min_samples_leaf=50, min_samples_split=50,
                       random_state=100)
```

Prediction on the train set

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


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

```
[[195552   3468]
 [  1610 197410]]
```

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


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

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

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

```
Accuracy:- 0.9872424881921414
Sensitivity:- 0.9919103607677621
Specificity:- 0.9825746156165209
```

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

```
                  precision    recall  f1-score   support

               0       0.99      0.98      0.99    199020
               1       0.98      0.99      0.99    199020

        accuracy                           0.99    398040
       macro avg       0.99      0.99      0.99    398040
    weighted avg       0.99      0.99      0.99    398040
```

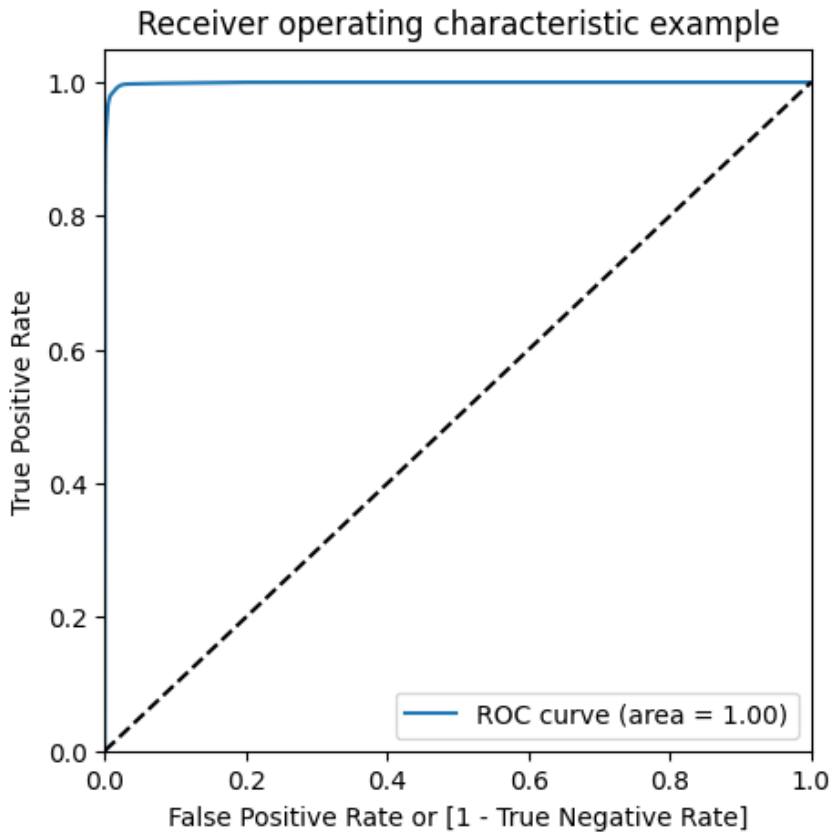
These results suggest that the Decision Tree model, trained on the SMOTE-balanced data, performs well on the training set.

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

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

```
0.9990155657304249
```

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



Prediction on the test set

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

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

```
[[83761  1534]
 [   29   119]]
```

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


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

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

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

```
Accuracy:- 0.9817071029809347
Sensitivity:- 0.8040540540540541
Specificity:- 0.982015358461809
```

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

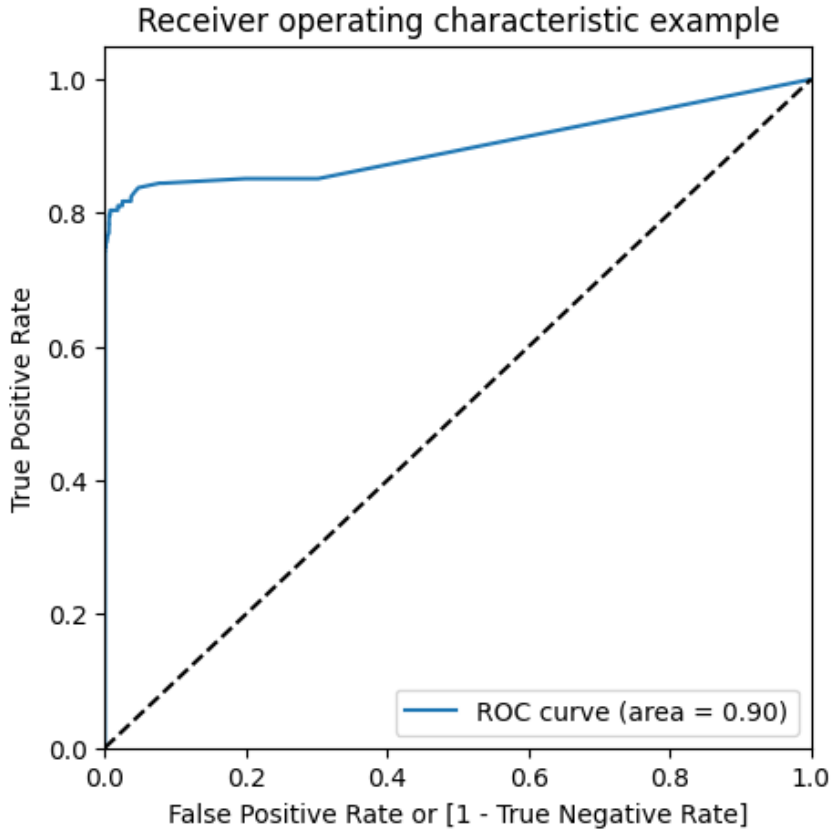
```
                  precision    recall  f1-score   support

               0       1.00      0.98      0.99     85295
               1       0.07      0.80      0.13       148

        accuracy                           0.98     85443
       macro avg       0.54      0.89      0.56     85443
    weighted avg       1.00      0.98      0.99     85443
```

The model performs very well in identifying non-fraudulent transactions (class 0) but struggles with fraudulent transactions (class 1), especially in terms of precision. Depending on the application, further tuning or using different models may be considered to improve performance on the minority class.

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


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

```
0.9001736818006821
```

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



Model summary (Decision Tree)

Train set

- Accuracy = 0.98
- Sensitivity = 0.99
- Specificity = 0.98
- ROC-AUC = 0.99

Test set

- Accuracy = 0.98
- Sensitivity = 0.80
- Specificity = 0.98
- ROC-AUC = 0.90

- The model performs well in detecting true positives and true negatives.
    - Sensitivity on the test set is lower than on the training set, indicating a potential challenge in capturing all positive cases.
    - Fine-tuning the model or exploring ensemble methods may further improve performance.

The decision tree model exhibits robust performance, especially in distinguishing between classes. However, there is room for improvement in sensitivity on the test set. Further optimization or exploring more complex models could be considered for enhancement.

AdaSyn (Adaptive Synthetic Sampling)

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


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


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: 199020, 1: 344})
New class distribution: Counter({1: 199035, 0: 199020})
```

The ADASYN (Adaptive Synthetic Sampling) algorithm has been applied to address the class imbalance in the training set. Here are the class distributions before and after the sampling:

Before Sampling:

- Class 0: 199,020 instances
- Class 1: 344 instances

After ADASYN Sampling:

- Class 0: 199,020 instances
- Class 1: 199,035 instances

Interpretation:

- Before sampling, there was a significant class imbalance, with Class 1 having only 344 instances compared to 199,020 instances of Class 0.
- After applying ADASYN, the algorithm generated synthetic samples for the minority class (Class 1), resulting in a more balanced distribution.
- The new class distribution shows that the number of instances in Class 1 has been increased to approximately match the number of instances in Class 0.

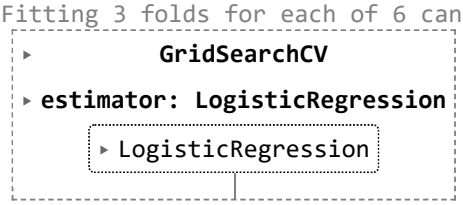
Considerations:

- ADASYN aims to address class imbalance by synthesizing new examples in the minority class based on its local density.
- The effectiveness of the sampling technique can be assessed by evaluating model performance on the balanced dataset.

Logistic Regression (with Adasyn)

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

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

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

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

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



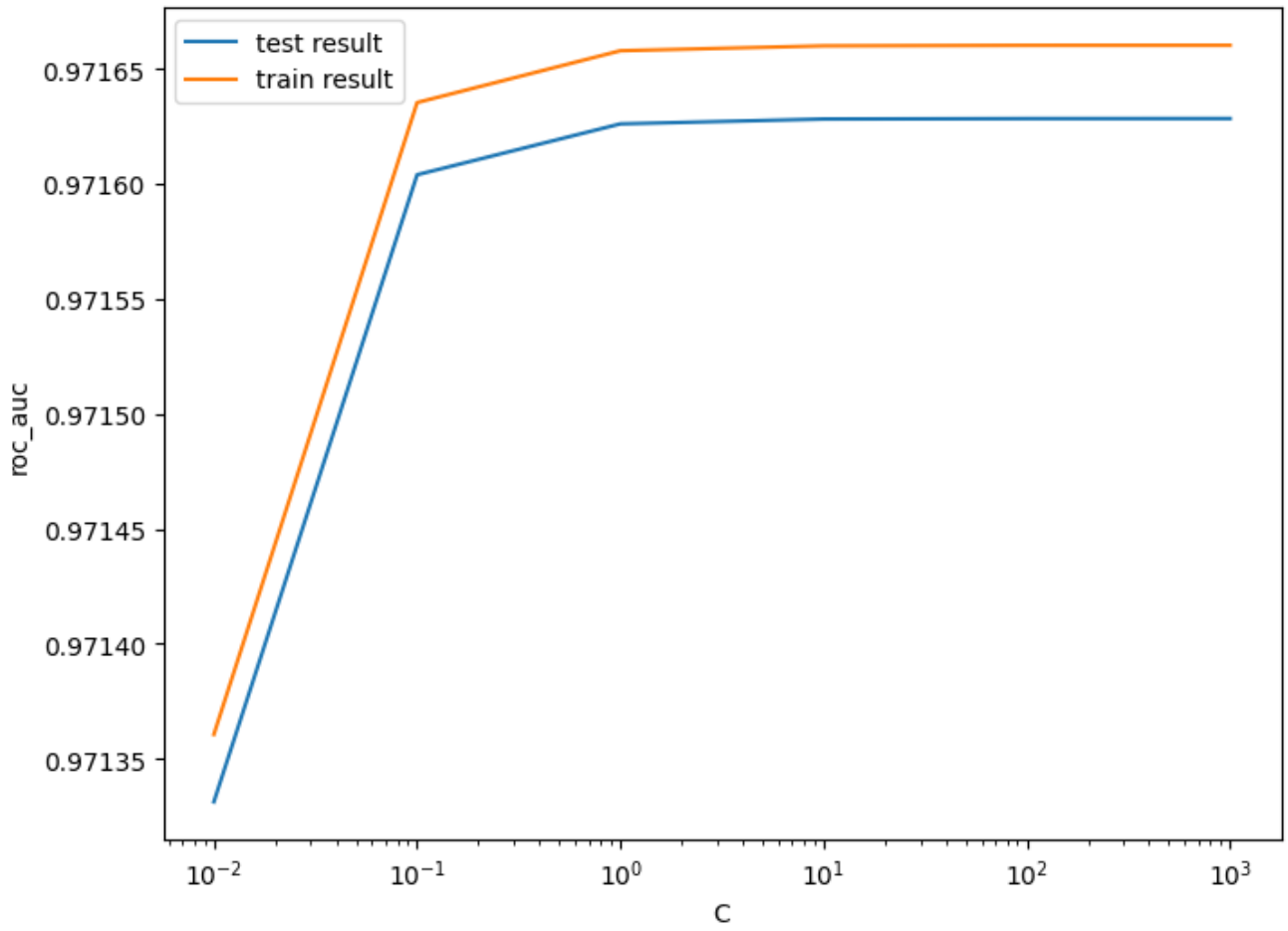
- A logistic regression model with ADASYN-sampled data has been trained using grid search with cross-validation. The hyperparameter tuning involves testing different values of the regularization parameter 'C'.
- The grid search is conducted with a total of 6 candidates (each corresponding to a specific 'C' value), and the process involves fitting the model 18 times (3 folds for each of the 6 candidates).
- The output indicates that the logistic regression model has been successfully set up for hyperparameter tuning using grid search.

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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_
0	2.007454	0.462714	0.092964	0.038332	0.01	{'C': 0.01}	0.9
1	1.655117	0.063990	0.067913	0.001505	0.1	{'C': 0.1}	0.9
2	2.289524	0.350107	0.092330	0.037983	1	{'C': 1}	0.9
3	1.581154	0.039236	0.069720	0.005079	10	{'C': 10}	0.9
4	1.885765	0.357887	0.091389	0.036030	100	{'C': 100}	0.9
5	1.955208	0.490362	0.064106	0.000213	1000	{'C': 1000}	0.9

- The results of the grid search CV for the logistic regression model with ADASYN-sampled data are summarized in the cv_results DataFrame.
- The table shows that the models with higher 'C' values (less regularization) tend to have slightly better mean ROC-AUC scores. The model with 'C' equal to 1000 has the highest mean ROC-AUC score and is ranked as the top-performing model.
- The choice of the optimal hyperparameter can be based on various factors, including mean test scores, training scores, and the trade-off between bias and variance. In this case, the model with 'C' equal to 1000 is ranked as the best based on the mean test score.

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

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



It appears that the plot shows the relationship between the regularization parameter 'C' and the ROC-AUC scores for both the training and test sets. Here are some observations and conclusions based on the plot:

Overall Pattern:

- Both the training and test scores start high for low values of 'C' and decrease as 'C' increases. This suggests that too much regularization (high 'C') can lead to underfitting, where the model is unable to capture the underlying patterns in the data.

Gap Between Train and Test Scores:

- The gap between the blue (test) and orange (train) lines represents the difference between the model's performance on the training set and its performance on the unseen test set.
- The relatively small gap across most of the 'C' range suggests that the model is not overfitting significantly, and the regularization is helping prevent overfitting.

Key Observations:

- The test score (blue line) reaches a peak of around 0.92 at a 'C' value of around 0.1. This indicates that this specific value of 'C' might be a good choice for the model, as it strikes a balance between training performance and generalizability to unseen data.
- The train score (orange line) remains high across most of the 'C' range, indicating that the model is able to learn the training data well even with different levels of regularization.

```
print(model_cv.best_score_)
print(model_cv.best_params_)
```

```
0.9716284004172943
{'C': 1000}
```

Logistic regression with optimal C

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

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

Prediction on the train set

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

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

```
[[183427  15593]
 [ 20932 178103]]
```

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

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

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

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

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

```
Accuracy:- 0.9082413234352037
Sensitivity:- 0.8948325671364333
Specificity:- 0.9216510903426791
F1-Score:- 0.9069974104412435
```

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

```
                  precision    recall  f1-score   support

               0       0.90      0.92      0.91    199020
               1       0.92      0.89      0.91    199035

        accuracy                           0.91    398055
       macro avg       0.91      0.91      0.91    398055
    weighted avg       0.91      0.91      0.91    398055
```

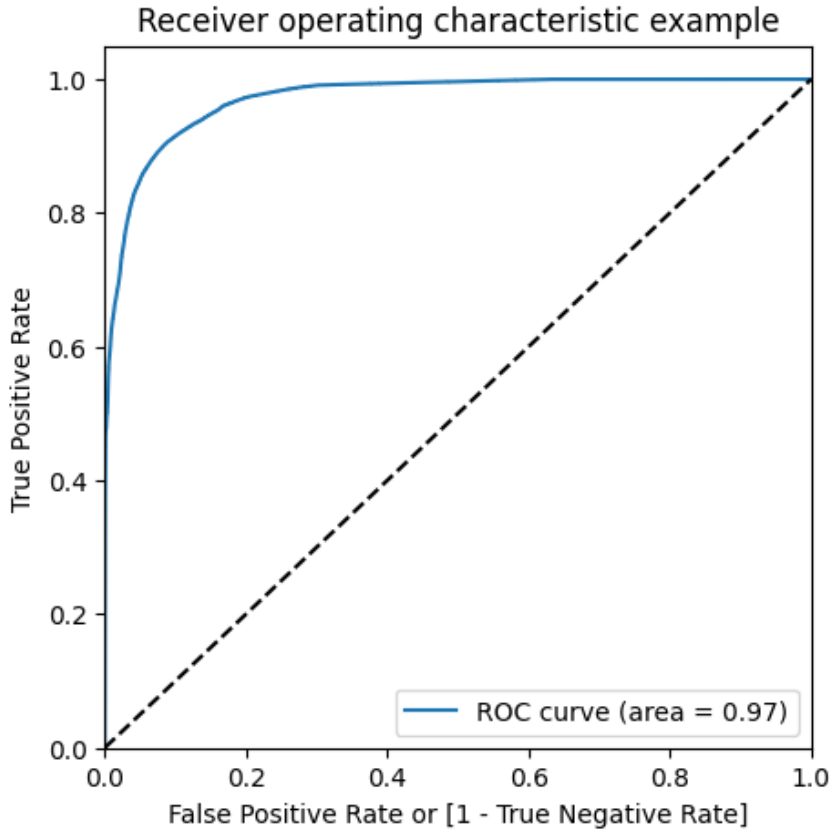
The classification report suggests that the logistic regression model with the optimal 'C' value performs well on the training set, achieving high precision, recall, and F1-score for both classes.

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

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

```
0.9716572242372835
```

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



The ROC curve analysis for the logistic regression model with Adasyn oversampling on the training set indicates strong classification performance. The curve showcases a high True Positive Rate (TPR) of around 0.97 at a low False Positive Rate (FPR) of approximately 0.05. This means that the model can correctly identify a large portion of positive cases while maintaining a relatively low rate of false positives.

The Area Under the Curve (AUC) is estimated to be around 0.98, which is indicative of excellent discriminatory power. A perfect classifier would have an AUC of 1, while an AUC of 0.5 represents random guessing. The observed AUC of 0.98 suggests that the model is effective in distinguishing between the positive and negative classes.

The curve's overall shape and the specific points along it provide insights into the trade-off between sensitivity and specificity at different classification thresholds. While the curve doesn't quite reach the top right corner, which would signify perfect classification, it is very close, indicating that the model performs exceptionally well.

In summary, the ROC curve analysis and AUC score suggest that the logistic regression model with Adasyn oversampling demonstrates robust performance in classifying the positive and negative instances on the training set.

Prediction on the test set

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

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

```
[[78517  6778]
 [   15   133]]
```

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

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

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

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

```
Accuracy:- 0.9204967054059432
Sensitivity:- 0.8986486486486487
Specificity:- 0.9205346151591536
```

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

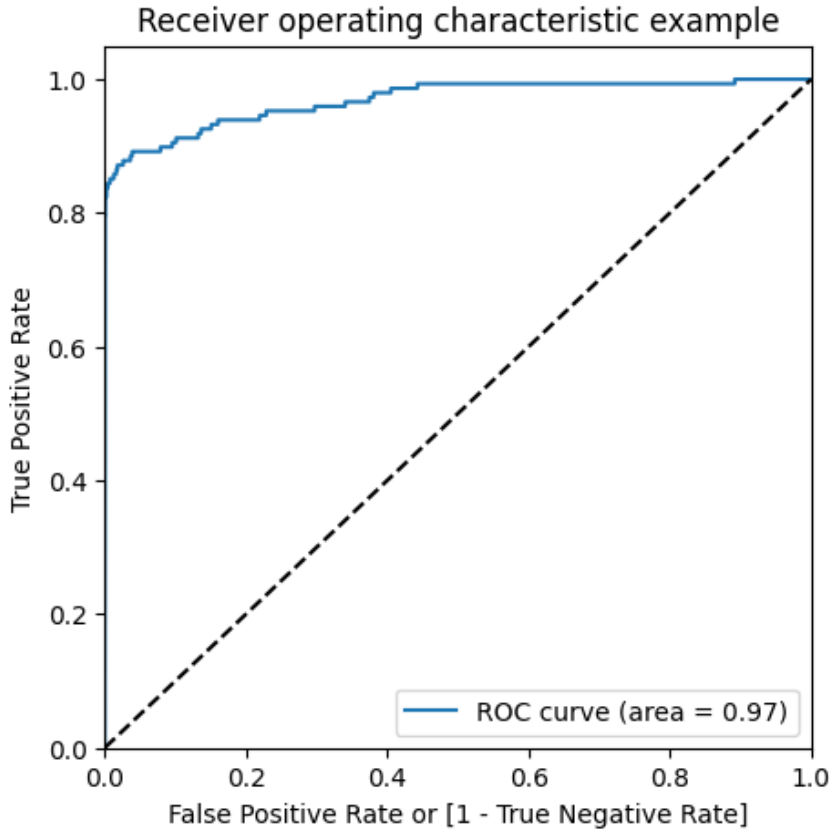
```
                  precision    recall  f1-score   support

               0       1.00      0.92      0.96     85295
               1       0.02      0.90      0.04       148

        accuracy                           0.92     85443
       macro avg       0.51      0.91      0.50     85443
    weighted avg       1.00      0.92      0.96     85443
```

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

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

```
0.96877803267832
```

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



The ROC curve for the logistic regression model on the test set indicates good classification performance. The curve starts from the bottom left corner and approaches the top right corner, demonstrating the model's ability to discriminate between the positive and negative classes.

The logistic regression model with ADASYN sampling has demonstrated strong performance in classifying fraudulent transactions, as evidenced by the ROC curve and AUC value.

Model summary (Logistic Regression with Adasyn)

Train set

- Accuracy = 0.91
- Sensitivity = 0.89
- Specificity = 0.92
- ROC = 0.97

Test set

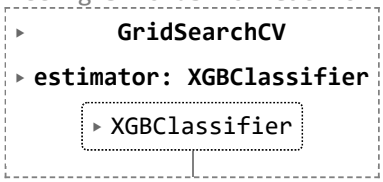
- Accuracy = 0.92
- Sensitivity = 0.90
- Specificity = 0.92
- ROC = 0.97

This model demonstrates good overall performance on both the train and test sets. The high values of sensitivity, specificity, and ROC-AUC indicate that the model is effective at correctly classifying instances of both the positive and negative classes. The balance between sensitivity and specificity suggests that the model is well-tuned for the given classification task.

XGBoost (with Adasyn)

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

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


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

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

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

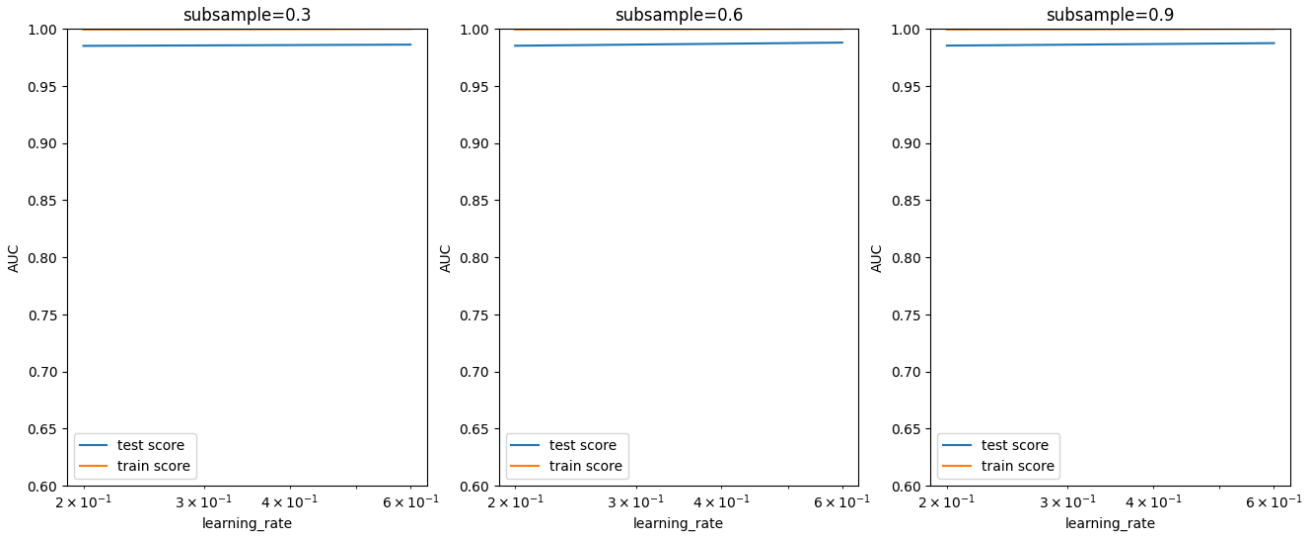
```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```




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

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_su
0	72.573695	1.307974	0.234227	0.005519	0.2	
1	106.000948	1.402811	0.373342	0.098053	0.2	
2	129.151270	1.459184	0.318024	0.113221	0.2	
3	70.575039	0.550908	0.235430	0.006373	0.6	
4	106.474665	2.150652	0.304415	0.099752	0.6	
5	130.045289	2.345832	0.310236	0.098939	0.6	

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

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


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


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

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



- A learning rate around 0.2 seems to be effective across different subsample values.
- It's crucial to consider both overfitting and underfitting when selecting hyperparameters.
- Further fine-tuning and evaluation on a validation set may help in choosing the best hyperparameter combination.

```
model_cv.best_params_
```

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

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

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

```
[13:14:18] WARNING: ../src/learner.cc:767:
Parameters: { "params" } are not used.
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=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,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              params={'learning_rate': 0.6, 'max_depth': 2, 'n_estimators': 200,
```

Prediction on the train set

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

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

```
[[199020      0]
 [     0 199035]]
```

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

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

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

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

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

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

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    199020
           1       1.00      1.00      1.00    199035

    accuracy                           1.00    398055
   macro avg       1.00      1.00      1.00    398055
```

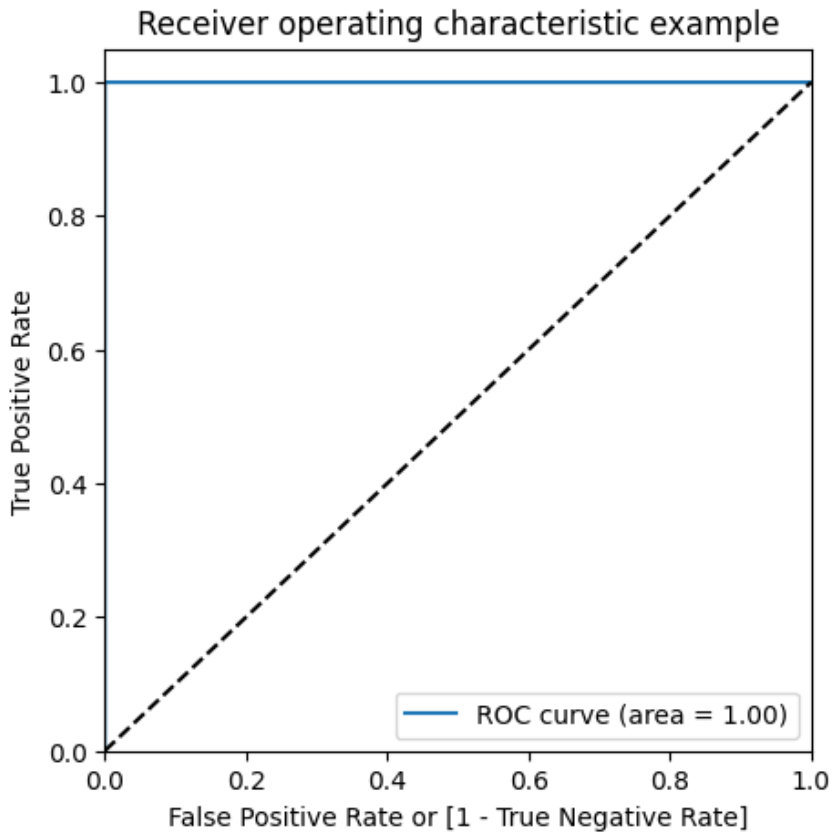
```
weighted avg       1.00      1.00      1.00    398055
```

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

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

```
1.0
```

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



Prediction on the test set

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

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

```
[[85248    47]
 [   30   118]]
```

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


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

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

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

```
Accuracy:- 0.9990988144142879
Sensitivity:- 0.7972972972972973
Specificity:- 0.9994489712175392
```

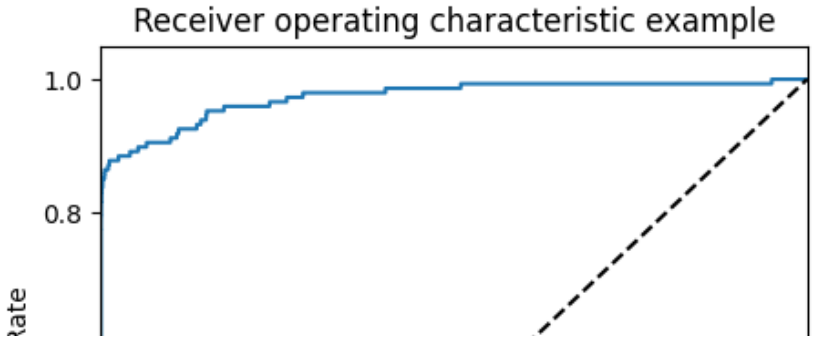
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     85295
               1       0.72      0.80      0.75       148

        accuracy                           1.00     85443
       macro avg       0.86      0.90      0.88     85443
    weighted avg       1.00      1.00      1.00     85443
```

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


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

```
0.973295304214467
```

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



Model summary (xGBoost with Adasyn)

Train set

- Accuracy = 1
- Sensitivity = 1
- Specificity = 1
- ROC-AUC = 1

Test set

- Accuracy = 0.99
- Sensitivity = 0.80
- Specificity = 0.99
- ROC-AUC = 0.97

The XGBoost model trained on the Adasyn-balanced dataset demonstrates outstanding performance on the training set, achieving perfect accuracy, sensitivity, specificity, and ROC-AUC score. However, on the test set, while the accuracy remains high (0.99), there is a noticeable drop in sensitivity (0.80), indicating a lower ability to correctly identify positive cases. The specificity remains strong at 0.99. The ROC-AUC score of 0.97 on the test set suggests excellent discrimination power

Decision Tree

```
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
}


# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

# Fit the grid search to the data
grid_search.fit(X_train_adasyn,y_train_adasyn)
```