Problem statement:-

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

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

Steps:-

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

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

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

import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
pd.set_option('display.max_columns', 500)
```

Reading and understanding the data

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

Out[ ]:

	Time	V1	V2	V3	V4	V5	V6	V7	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.2
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.3
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.2

In [ ]:
```
df.shape
```

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

In [ ]:
```
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
```

In [ ]:
```
df.describe()
```

Out[ ]:

	Time	V1	V2	V3	V4	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.6
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.3
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.1
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.9
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.4
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.1
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.4

Handling missing values

Handling missing values in columns

In [ ]:
```
# Cheking percent of missing values in columns
df_missing_columns = (round(((df.isnull().sum()/len(df.index))*100),2).to_frame(
df_missing_columns
```

Out[ ]:

	null
Time	0.0
V16	0.0
Amount	0.0
V28	0.0
V27	0.0
V26	0.0
V25	0.0
V24	0.0
V23	0.0
V22	0.0
V21	0.0
V20	0.0
V19	0.0
V18	0.0
V17	0.0
V15	0.0
V1	0.0
V14	0.0
V13	0.0
V12	0.0
V11	0.0
V10	0.0
V9	0.0
V8	0.0
V7	0.0
V6	0.0
V5	0.0
V4	0.0
V3	0.0
V2	0.0
Class	0.0

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

Checking the distribution of the classes

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

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

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

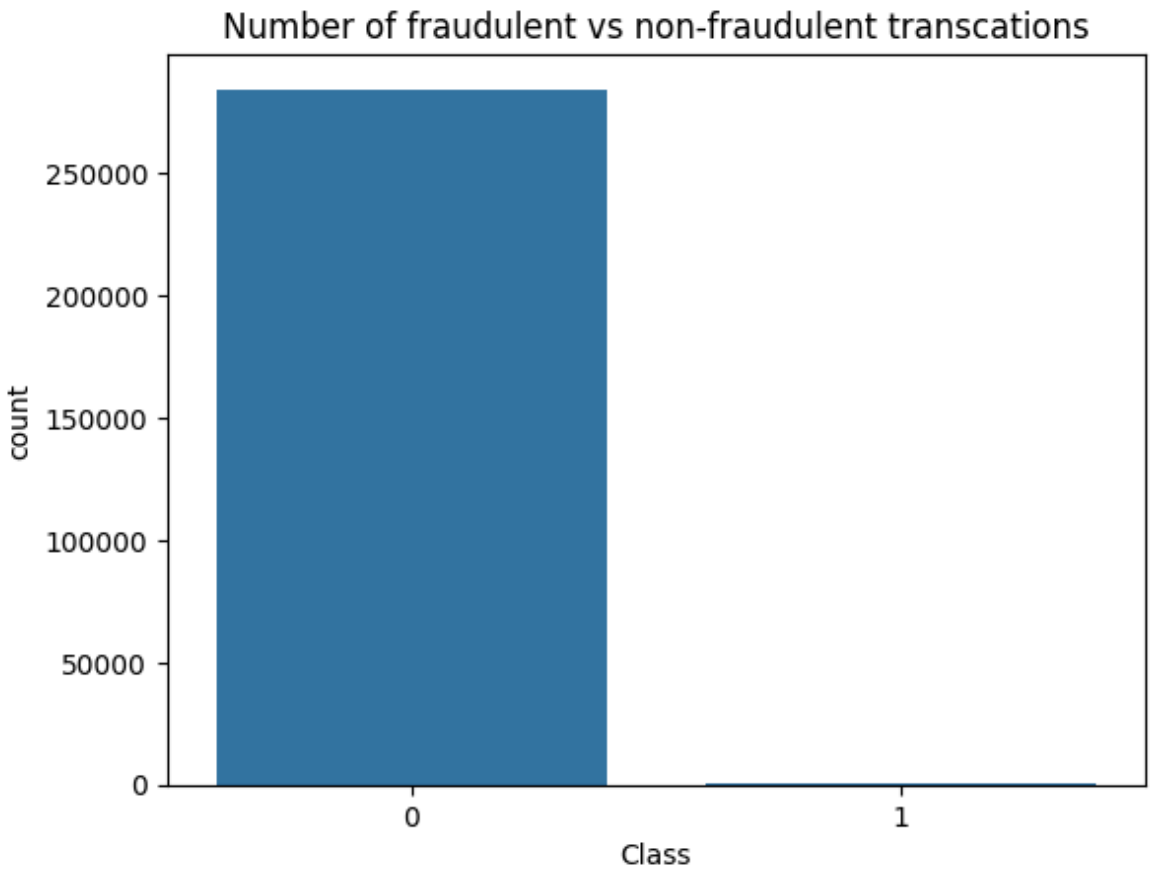
Out[ ]:
```
99.83
```

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

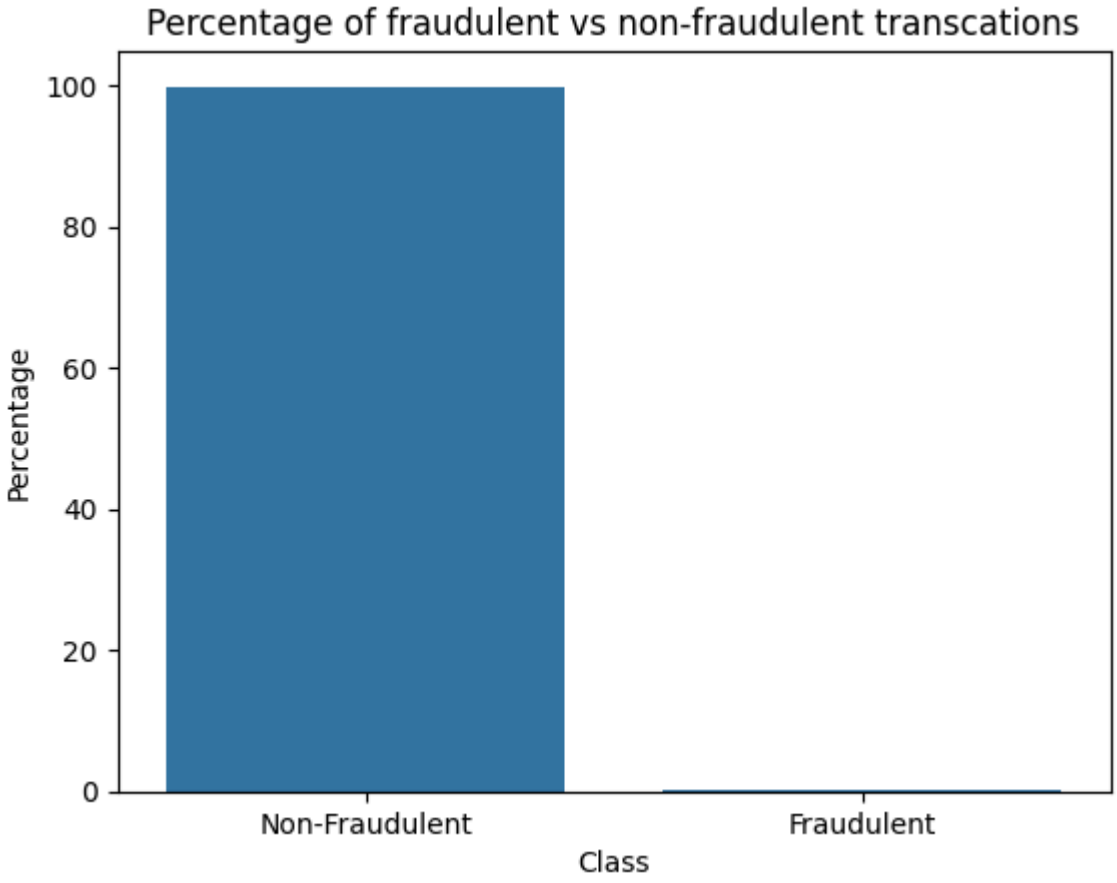
Out[ ]:
```
0.17
```

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

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



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



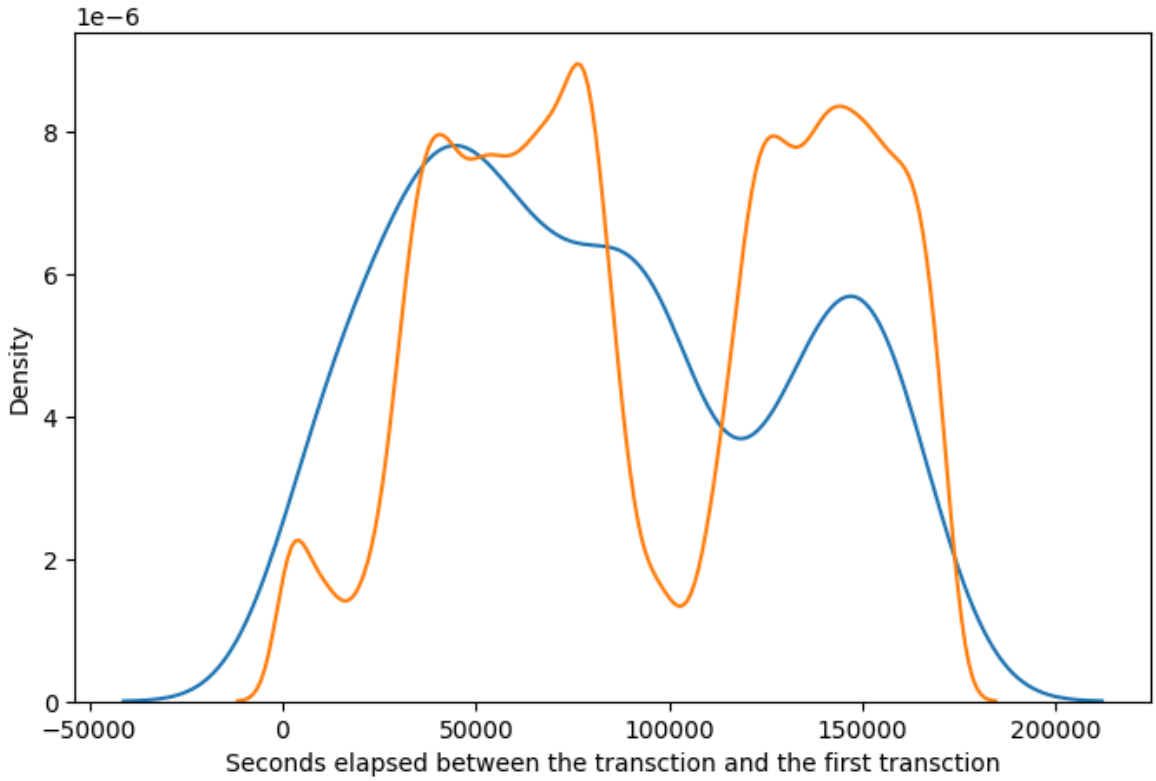
Outliers treatment

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

Observe the distribution of classes with time

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

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



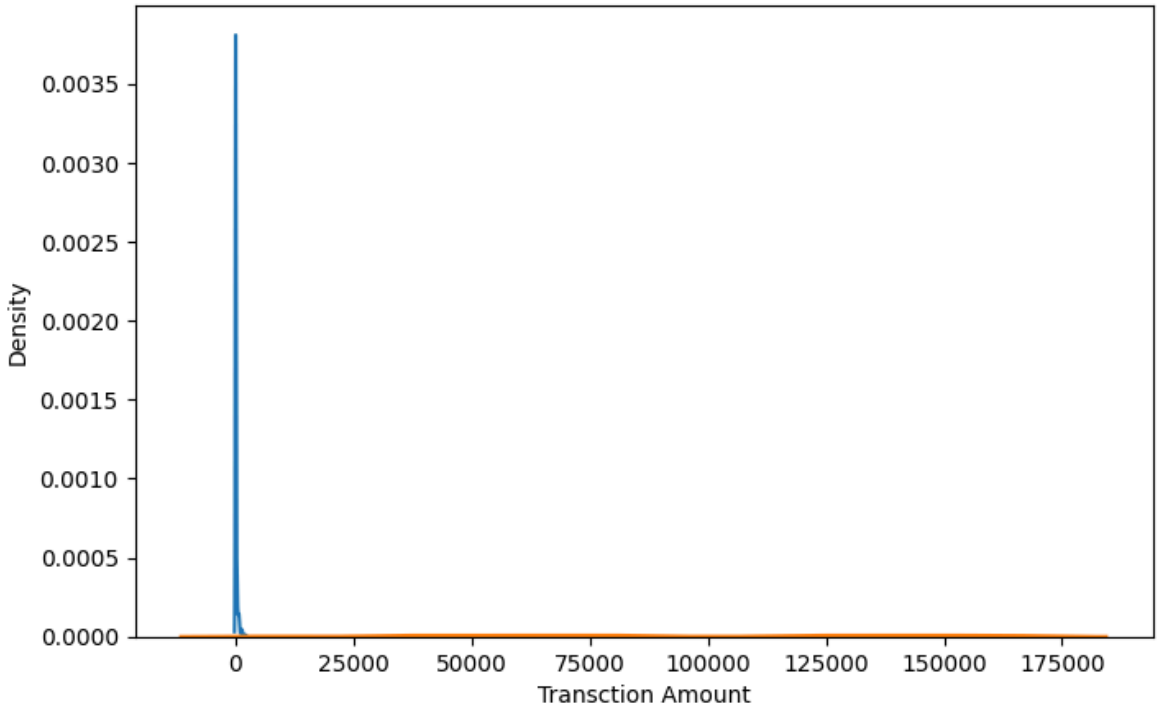
Analysis

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

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

Observe the distribution of classes with amount

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



Analysis

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

Train-Test Split

In [ ]:
```
# Import library
from sklearn.model_selection import train_test_split
```

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

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

In [ ]:
```
# Splitting data into train and test set 80:20
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_s
```

Feature Scaling

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

In [ ]:
```
# Standardization method
from sklearn.preprocessing import StandardScaler
```

In [ ]:
```
# Instantiate the Scaler
scaler = StandardScaler()
```

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

In [ ]:
```
X_train.head()
```

Out[ ]:

	V1	V2	V3	V4	V5	V6	V7	
201788	2.023734	-0.429219	-0.691061	-0.201461	-0.162486	0.283718	-0.674694	0.1
179369	-0.145286	0.736735	0.543226	0.892662	0.350846	0.089253	0.626708	-0.0
73138	-3.015846	-1.920606	1.229574	0.721577	1.089918	-0.195727	-0.462586	0.9
208679	1.851980	-1.007445	-1.499762	-0.220770	-0.568376	-1.232633	0.248573	-0.5
206534	2.237844	-0.551513	-1.426515	-0.924369	-0.401734	-1.438232	-0.119942	-0.4

Scaling the test set

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

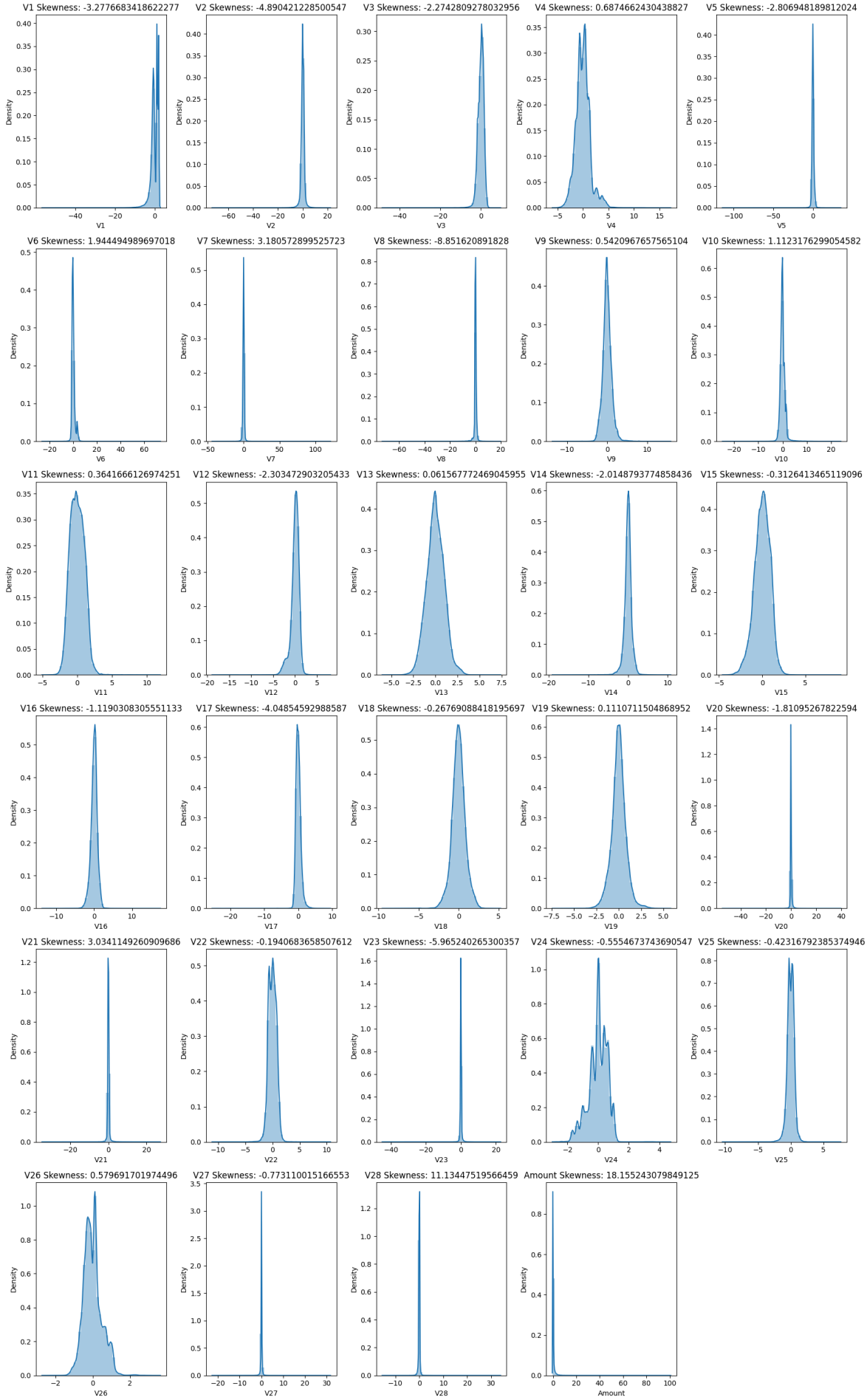
In [ ]:
```
# Transform the test set
X_test['Amount'] = scaler.transform(X_test[['Amount']])
X_test.head()
```

Out[ ]:

	V1	V2	V3	V4	V5	V6	V7	
49089	1.229452	-0.235478	-0.627166	0.419877	1.797014	4.069574	-0.896223	1.03
154704	2.016893	-0.088751	-2.989257	-0.142575	2.675427	3.332289	-0.652336	0.75
67247	0.535093	-1.469185	0.868279	0.385462	-1.439135	0.368118	-0.499370	0.30
251657	2.128486	-0.117215	-1.513910	0.166456	0.359070	-0.540072	0.116023	-0.21
201903	0.558593	1.587908	-2.368767	5.124413	2.171788	-0.500419	1.059829	-0.25

Checking the Skewness

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

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

for col in cols:
    k = k + 1
    plt.subplot(6, 5, k)
    sns.distplot(X_train[col])
    plt.title(col + ' Skewness: ' + str(X_train[col].skew()))

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



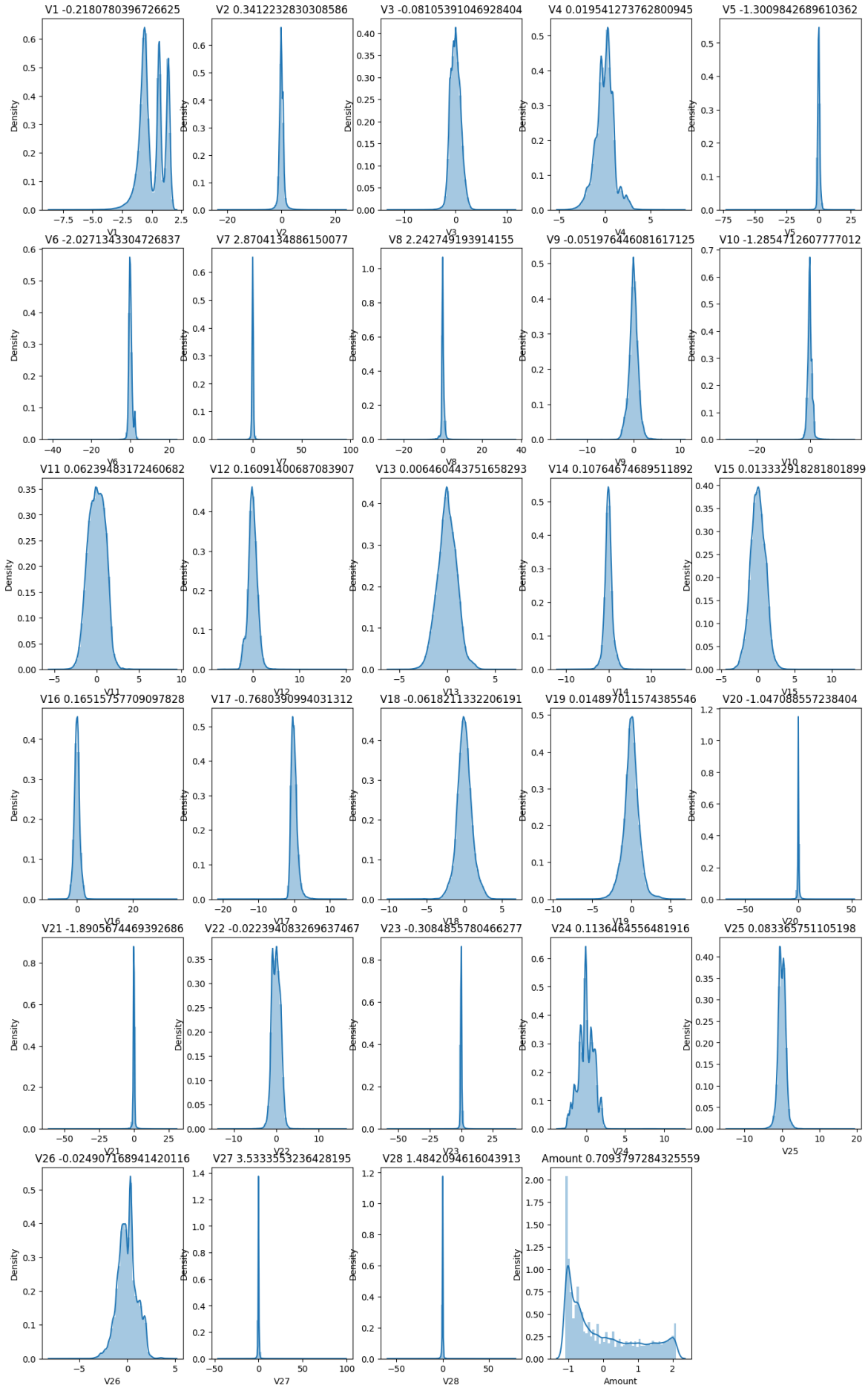
We see that there are many variables, which are heavily skewed. We will mitigate the skewness only for those variables for bringing them into normal distribution.

Mitigate skweness with PowerTransformer

In [ ]:
```
# 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)
```

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

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



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

Model building on imbalanced data

Metric Selection for Heavily Imbalanced Data

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

Reasons for Not Choosing SVM and Random Forest in Specific Cases

SVM

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

Random Forest

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

Exclusion of KNN in Model Building

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

Logistic regression

In [ ]:
```
# Importing scikit logistic regression module
from sklearn.linear_model import LogisticRegression
```

In [ ]:
```
# Impoting metrics
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import classification_report
```

Tuning hyperparameter C

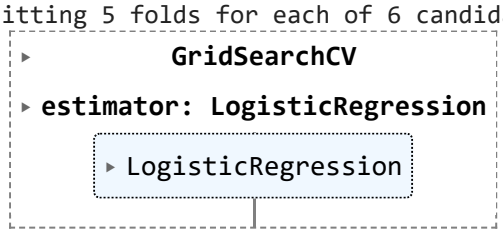
C is the the inverse of regularization strength in Logistic Regression. Higher values of C correspond to less regularization.

In [ ]:
```
# 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
```

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

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

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

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

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

Out[ ]:

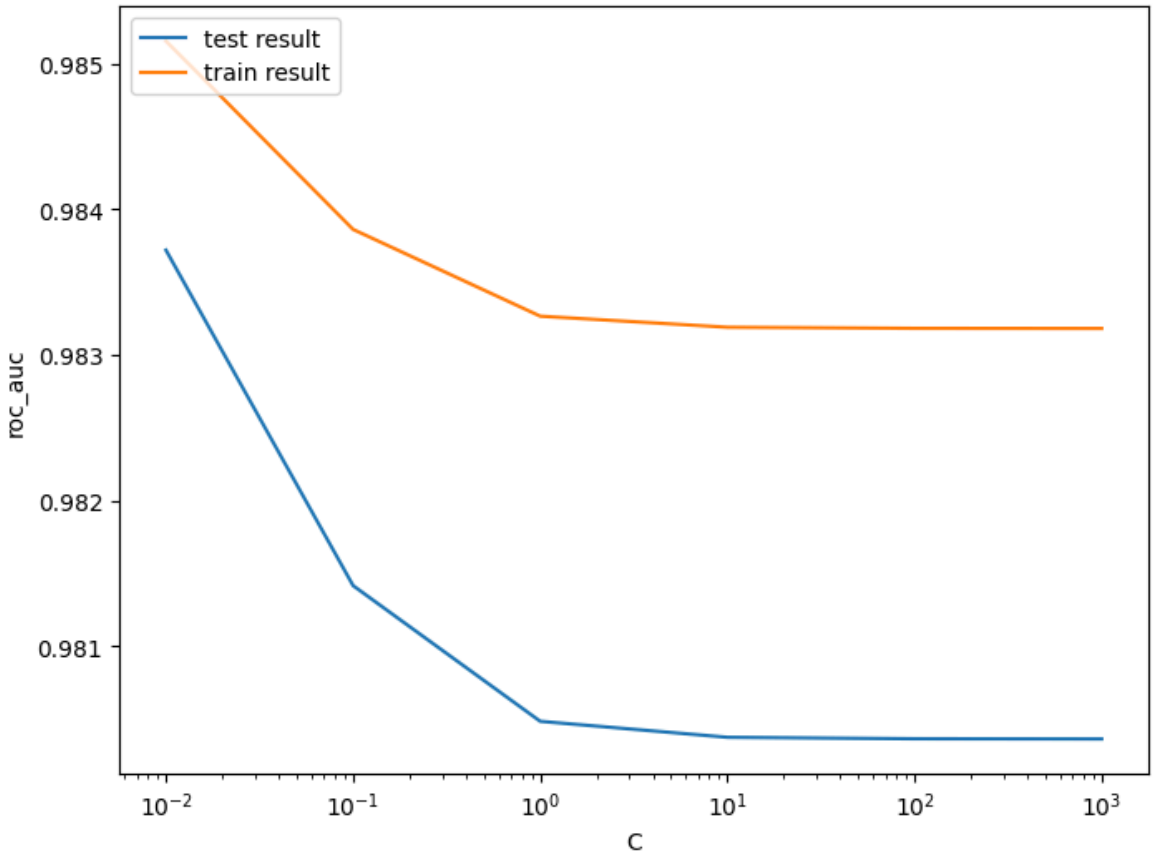


In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split
0	1.271986	0.379019	0.038552	0.019045	0.01	{'C': 0.01}	
1	1.184194	0.135116	0.031305	0.012860	0.1	{'C': 0.1}	
2	1.139086	0.123698	0.029236	0.012638	1	{'C': 1}	
3	1.118935	0.054724	0.026788	0.013746	10	{'C': 10}	
4	1.062211	0.118397	0.024069	0.005773	100	{'C': 100}	
5	1.103135	0.100601	0.024923	0.010002	1000	{'C': 1000}	

In [ ]:
```
# 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')
```



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

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

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

Logistic regression with optimal C

In [ ]:
```
# Instantiate the model with best C
logistic_imb = LogisticRegression(C=0.01)
```

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

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = logistic_imb_model.predict(X_train)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)
```

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

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train, y_train_pred))
```

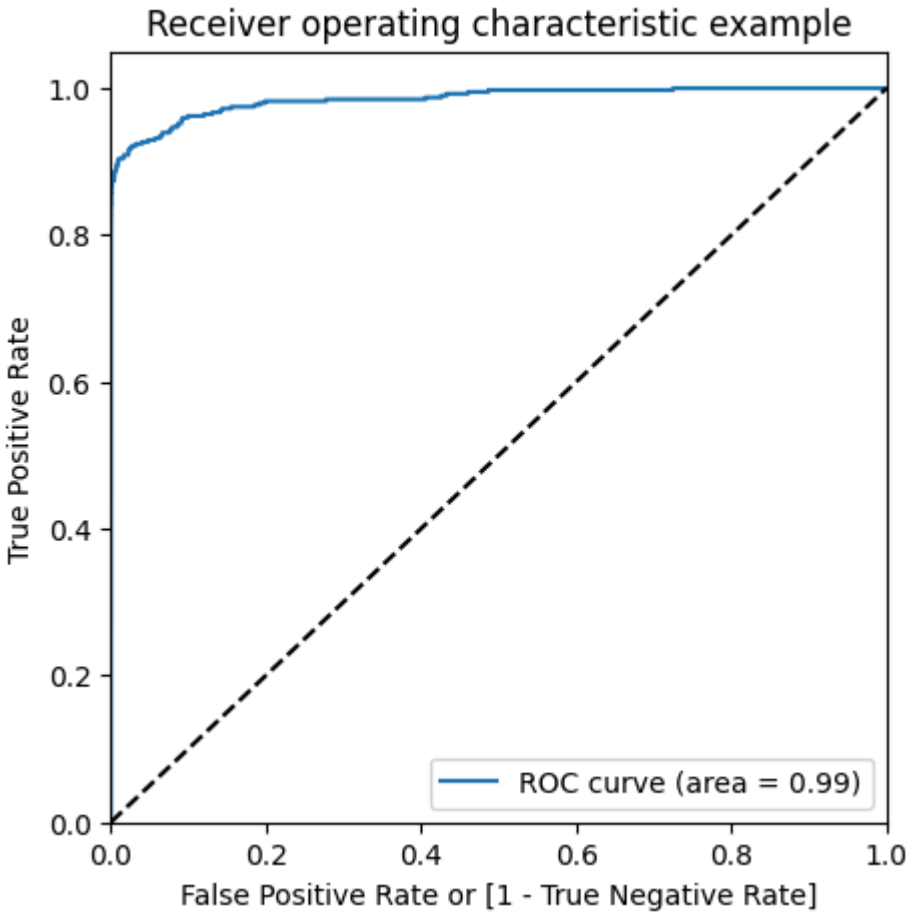
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       0.92      0.66      0.77       396

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

ROC on the train set

In [ ]:
```
# ROC Curve function

def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve( actual, probs,
                                              drop_intermediate = False )
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(5, 5))
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return None
```

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

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba)
```



We acheived very good ROC 0.99 on the train set.

Prediction on the test set

In [ ]:
```
# Prediction on the test set
y_test_pred = logistic_imb_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

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

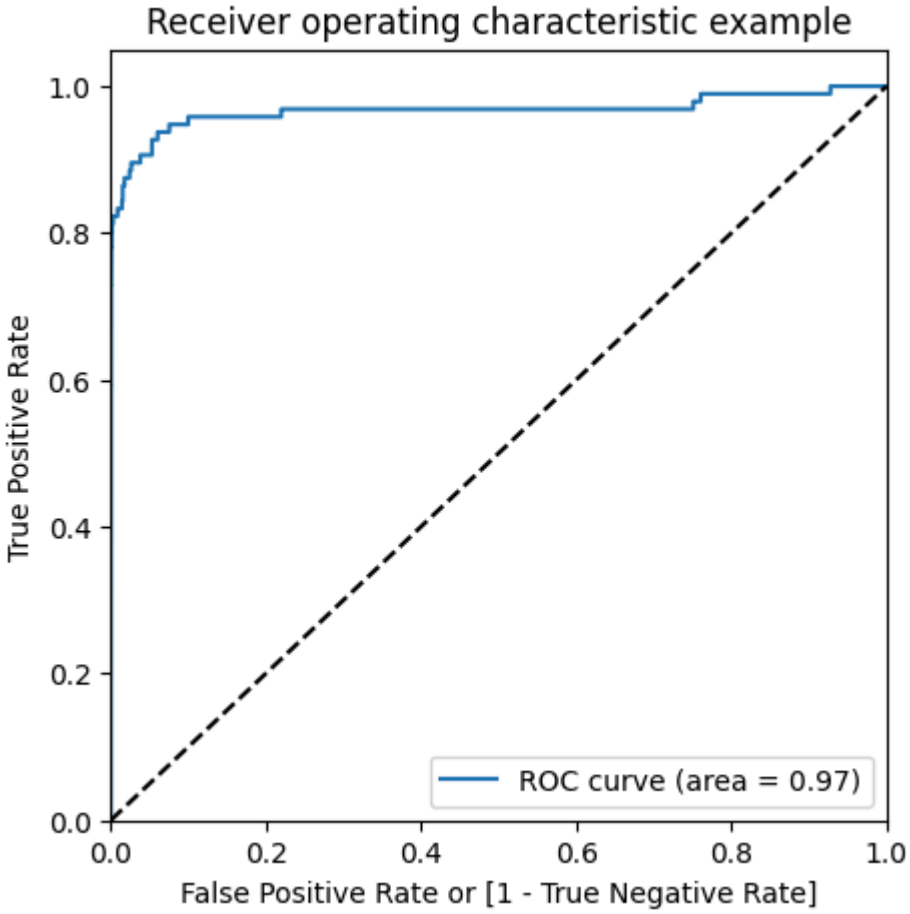
In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.77      0.56      0.65        96

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

ROC on the test set

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

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



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

Model summary

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

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

XGBoost

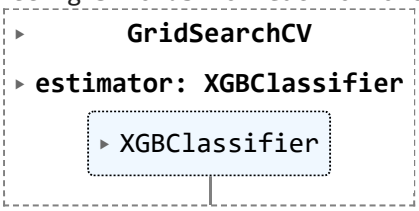
In [ ]:
```
# Importing XGBoost
from xgboost import XGBClassifier
```

Tuning the hyperparameters

In [ ]:
```
# 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
```

Out[ ]:



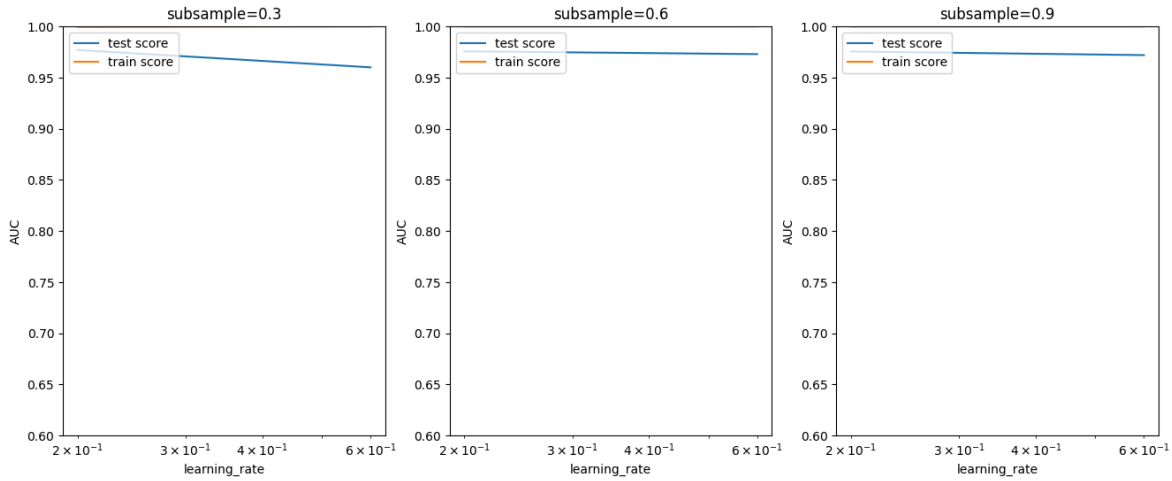
In [ ]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	p
0	2.535768	0.737668	0.067588	0.005965	0.2	
1	1.793594	0.134591	0.061041	0.004689	0.2	
2	1.675877	0.025984	0.056897	0.003275	0.2	
3	1.776437	0.021462	0.059674	0.001658	0.6	
4	1.748587	0.081171	0.057439	0.000894	0.6	
5	1.858781	0.060790	0.063512	0.007455	0.6	

In [ ]:

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

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


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


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

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



Model with optimal hyperparameters

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

In [ ]:
```
model_cv.best_params_
```

Out[ ]:
```
{'learning_rate': 0.2, 'subsample': 0.3}
```

In [ ]:
```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we
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)
```

Out[ ]:
```
XGBClassifier

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = xgb_imb_model.predict(X_train)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
```

```
print(confusion)
```

```
[[227449      0]
 [     0    396]]
```

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

In [ ]:
```
# 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
```

In [ ]:
```
# classification_report
print(classification_report(y_train, y_train_pred))
```

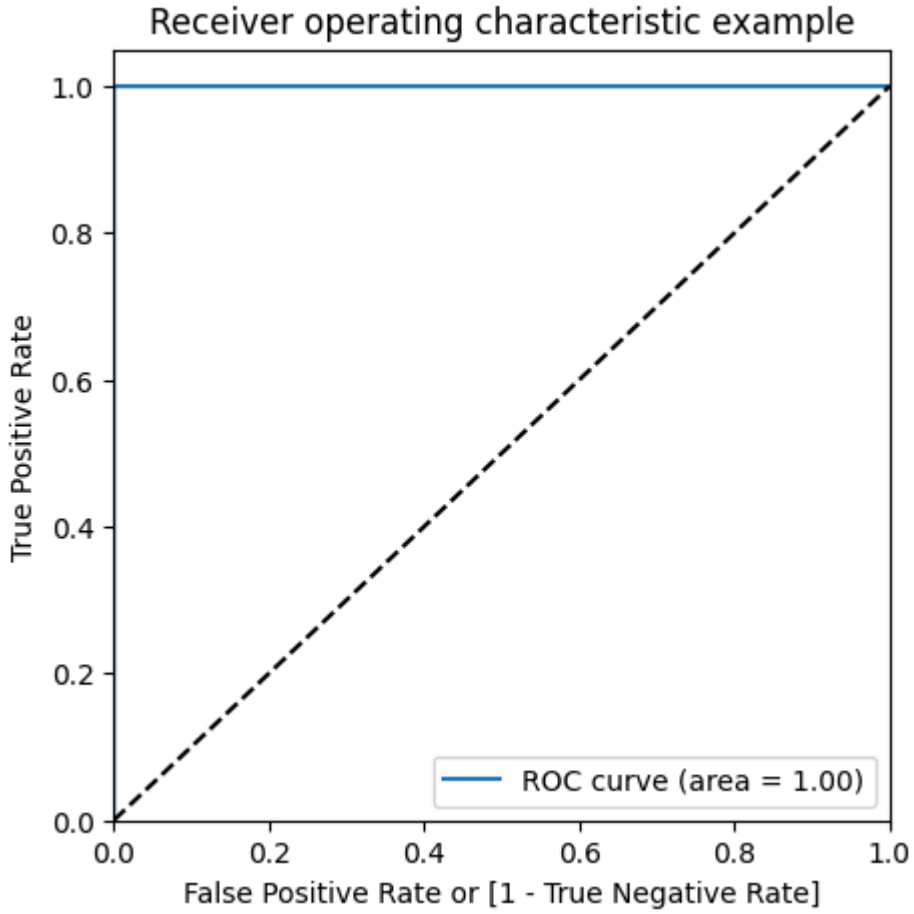
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00       396

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train, y_train_pred_proba_imb_xgb)
auc
```

Out[ ]:
```
1.0
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba_imb_xgb)
```



Prediction on the test set

In [ ]:
```
# Predictions on the test set
y_test_pred = xgb_imb_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

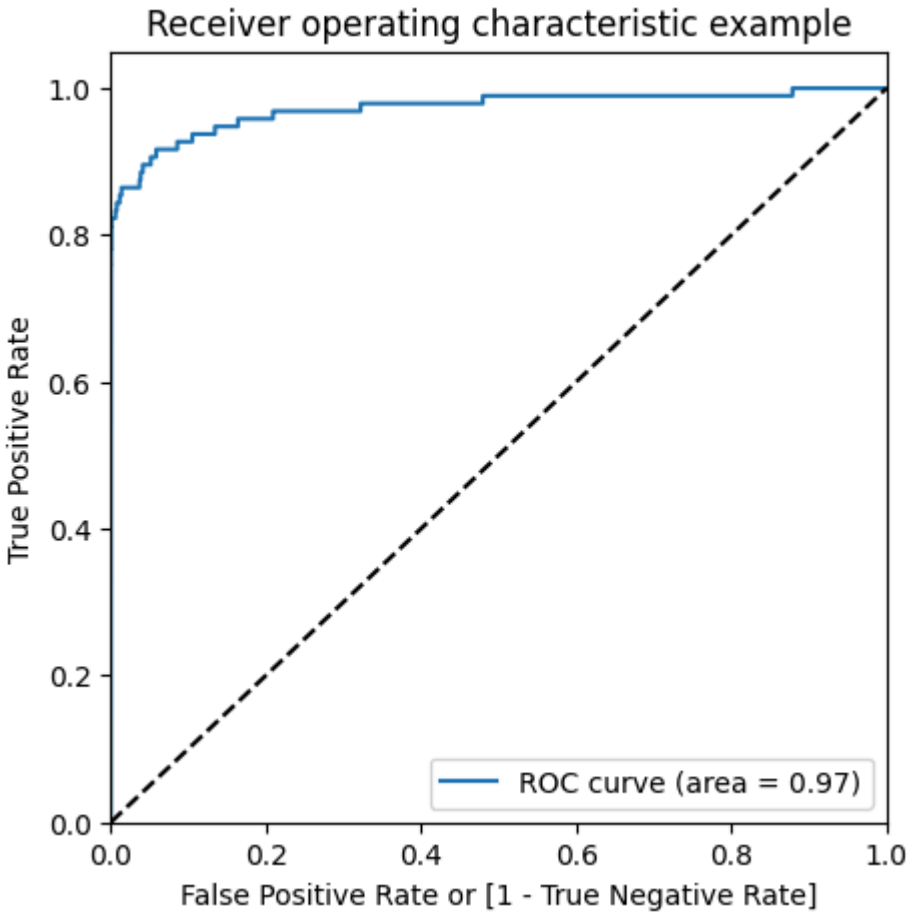
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.90      0.74      0.81        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9723599118981465
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

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

Choosing best model on the imbalanced data

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

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

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

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

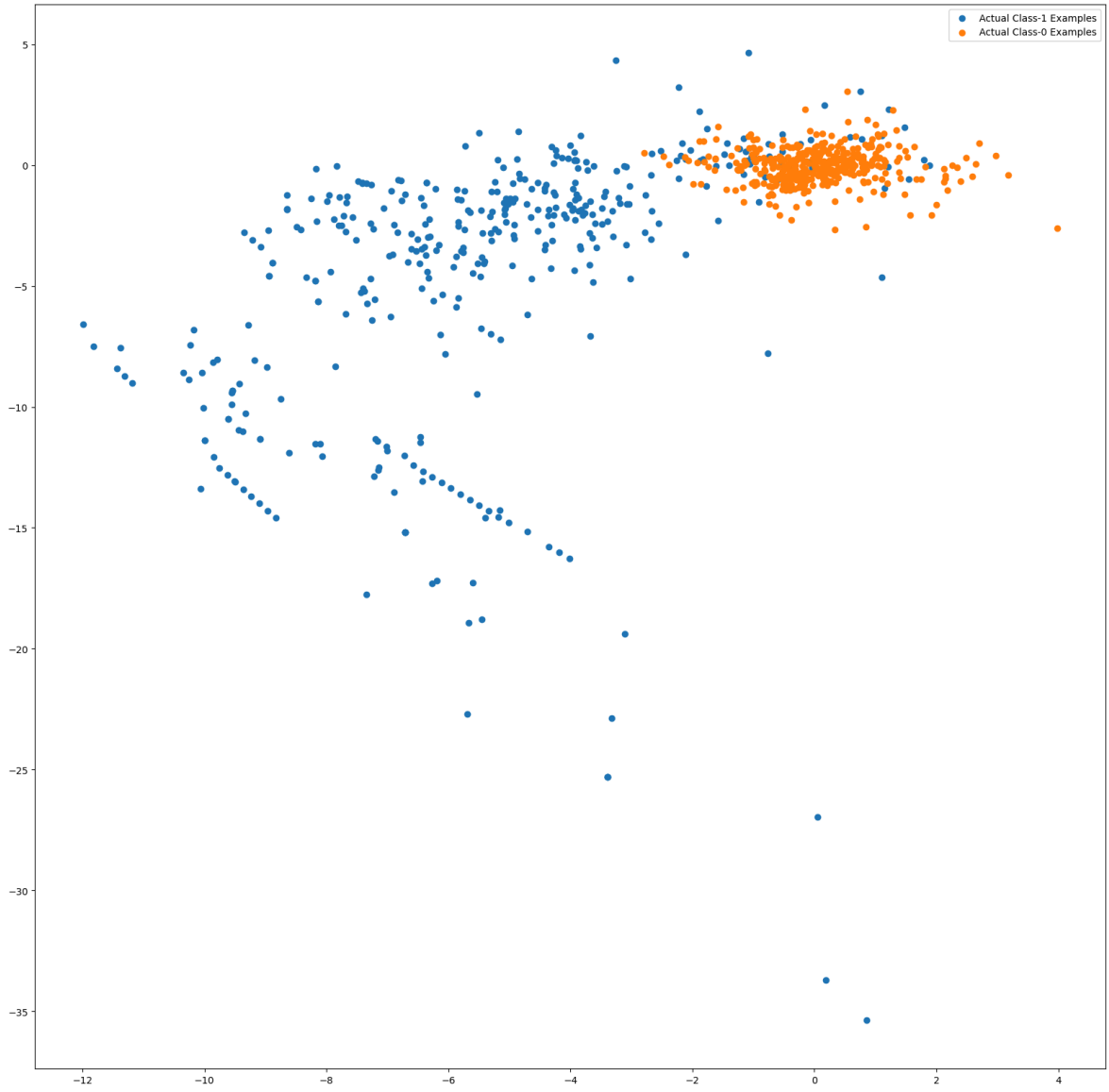
- This will not give much explanation on the already transformed dataset
- But it will help us in understanding if the dataset is not PCA transformed

In [ ]:
```
# Features of XGBoost model

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

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

np.random.shuffle(X_train_0)
```

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

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

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

Out[ ]: <matplotlib.legend.Legend at 0x161d3791990>



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

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

```
Train auc = 1.0
Threshold= 0.82052475
```

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

Handling data imbalance

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

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

Undersampling

In [ ]:
```
# Importing undersampler library
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter
```

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

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

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

Model building on balanced data with Undersampling

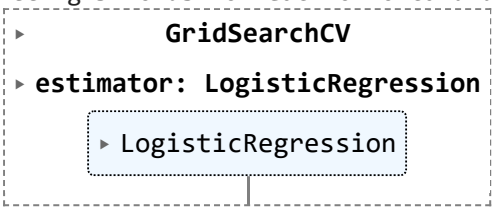
Logistic Regression

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

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

# Specifing score as roc-auc
model_cv = GridSearchCV(estimator = LogisticRegression(),
```

```
                        param_grid = params,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

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

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

Out[ ]:

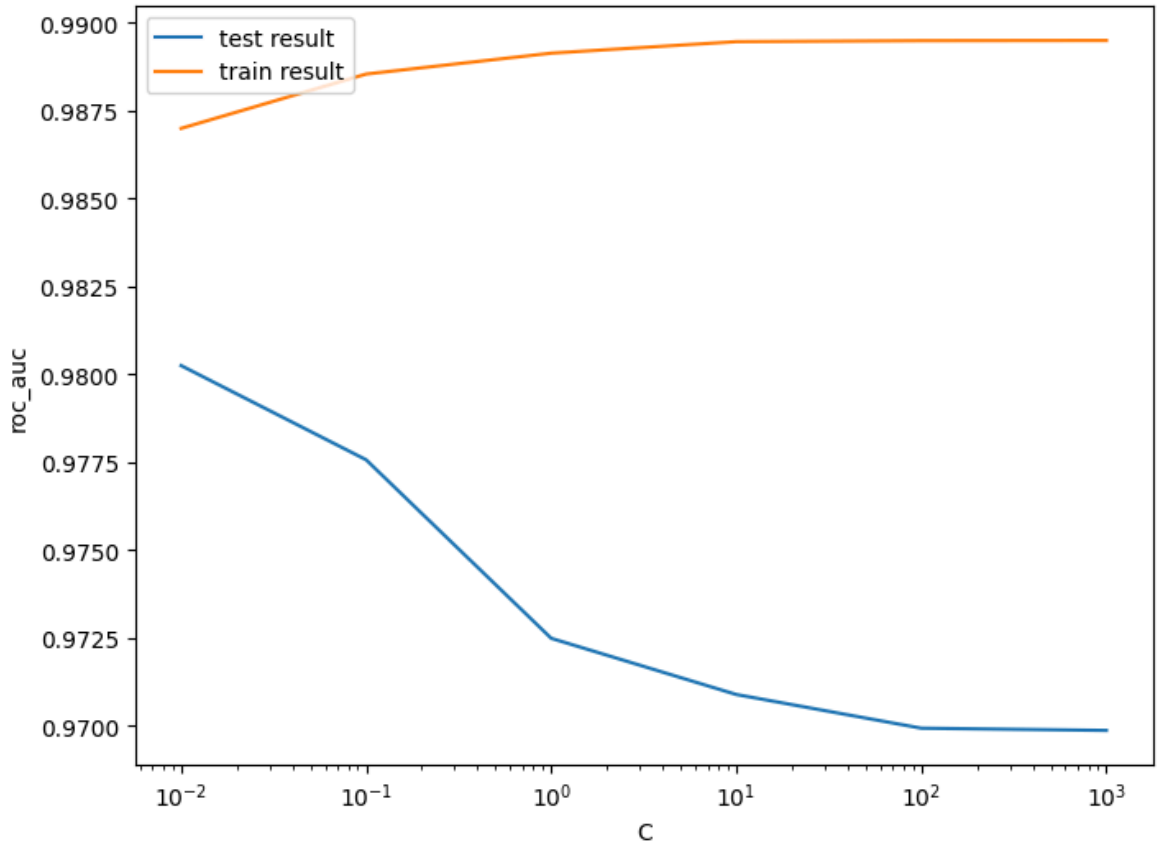


In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

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

In [ ]:
```
# 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')
```



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

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

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

Logistic regression with optimal C

In [ ]:
```
# Instantiate the model with best C
logistic_bal_rus = LogisticRegression(C=0.1)
```

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

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = logistic_bal_rus_model.predict(X_train_rus)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_pred)
print(confusion)
```

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

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

In [ ]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_rus, y_train_pred))

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

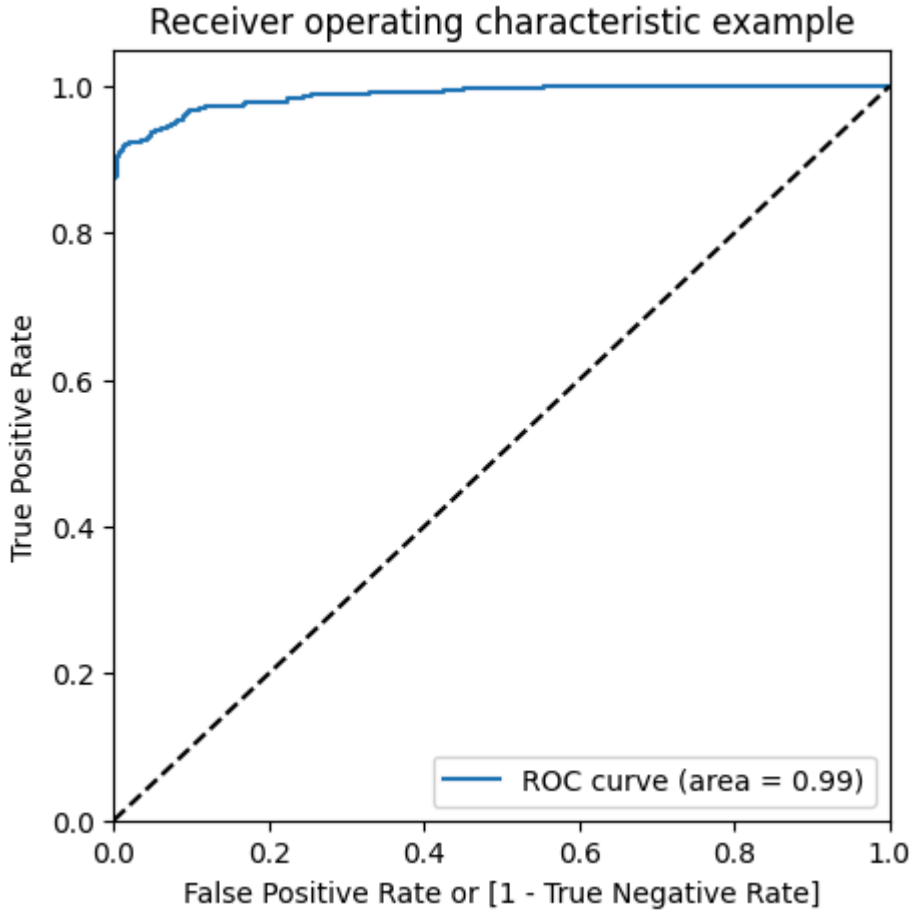
```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95       396
           1       0.98      0.92      0.95       396

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

Out[ ]:
```
0.9878264972961943
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Prediction on the test set
y_test_pred = logistic_bal_rus_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

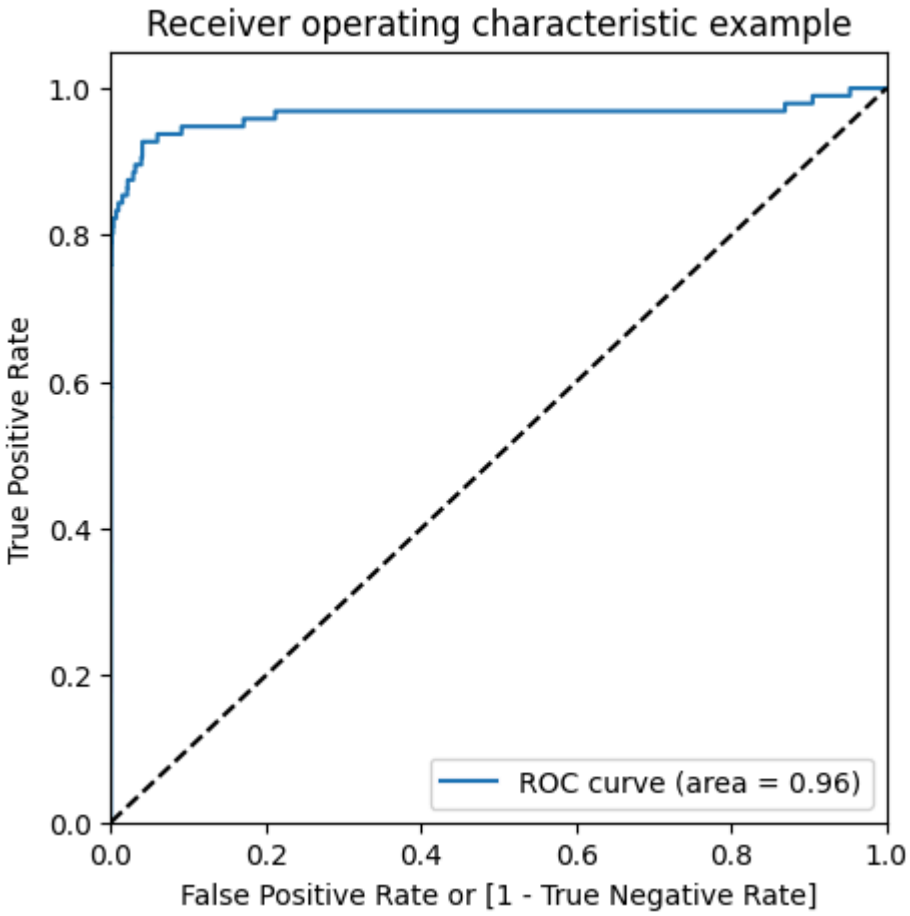
```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.07      0.85      0.13        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9630049516993165
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.95
  - Sensitivity = 0.92

- Specificity = 0.98
- ROC = 0.99

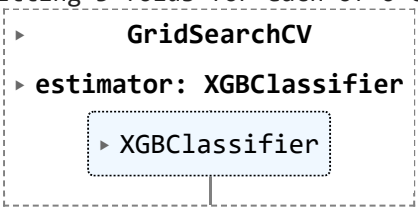
- Test set
  - Accuracy = 0.97
  - Sensitivity = 0.86
  - Specificity = 0.97
  - ROC = 0.96

XGBoost

In [ ]:
```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

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


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

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

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

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

Out[ ]:



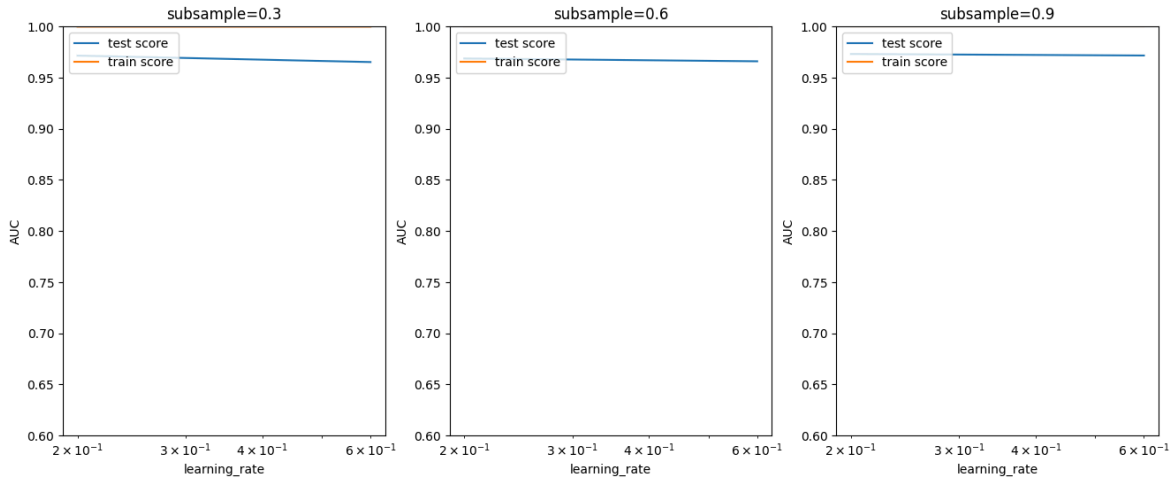
In [ ]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	p
0	0.150507	0.035023	0.012531	0.001830	0.2	
1	0.121757	0.009973	0.008923	0.000888	0.2	
2	0.118611	0.005923	0.010269	0.001908	0.2	
3	0.098031	0.007062	0.009510	0.001451	0.6	
4	0.101163	0.000077	0.010325	0.000427	0.6	
5	0.132967	0.049887	0.013083	0.003599	0.6	

In [ ]:

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

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


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


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

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



Model with optimal hyperparameters

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

In [ ]:
```
model_cv.best_params_
```

Out[ ]:
```
{'learning_rate': 0.2, 'subsample': 0.9}
```

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

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

Out[ ]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = xgb_bal_rus_model.predict(X_train_rus)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_rus)
```

```
print(confusion)
```

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

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

In [ ]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_rus, y_train_pred))

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

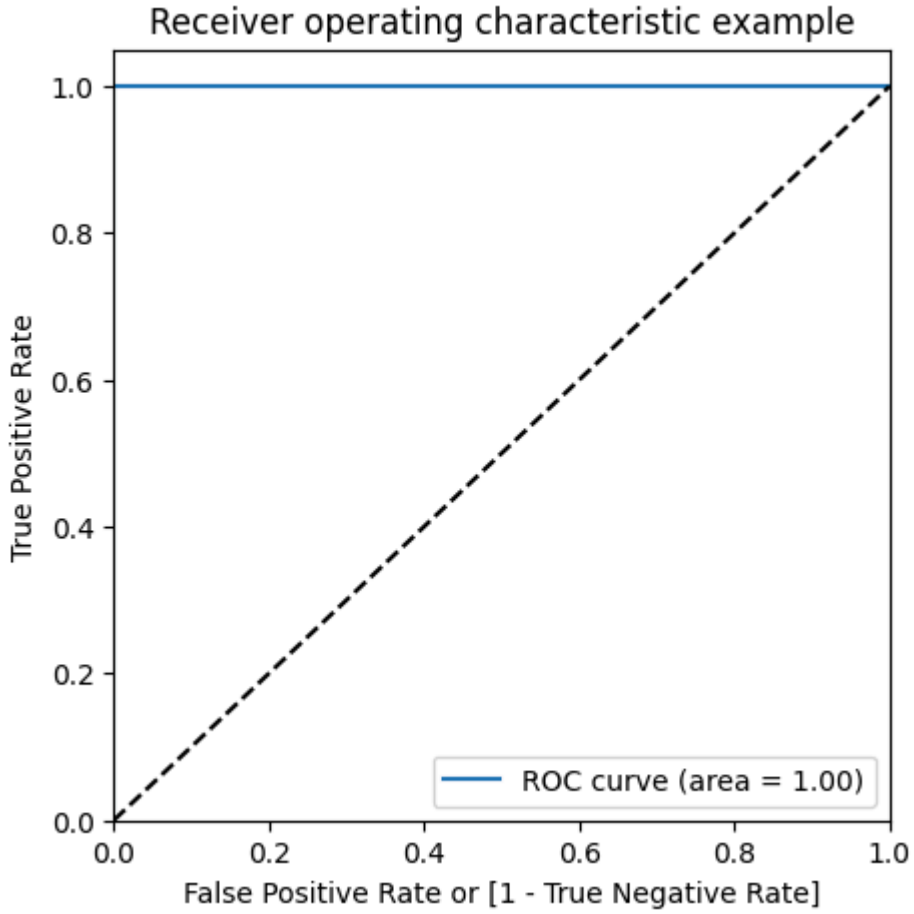
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       396
           1       1.00      1.00      1.00       396

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

Out[ ]:
```
1.0
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Predictions on the test set
y_test_pred = xgb_bal_rus_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55312  1554]
 [   14    82]]
```

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

In [ ]:
```
# 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:- 0.9724728766546118
Sensitivity:- 0.8541666666666666
Specificity:- 0.9726725987408996
```

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

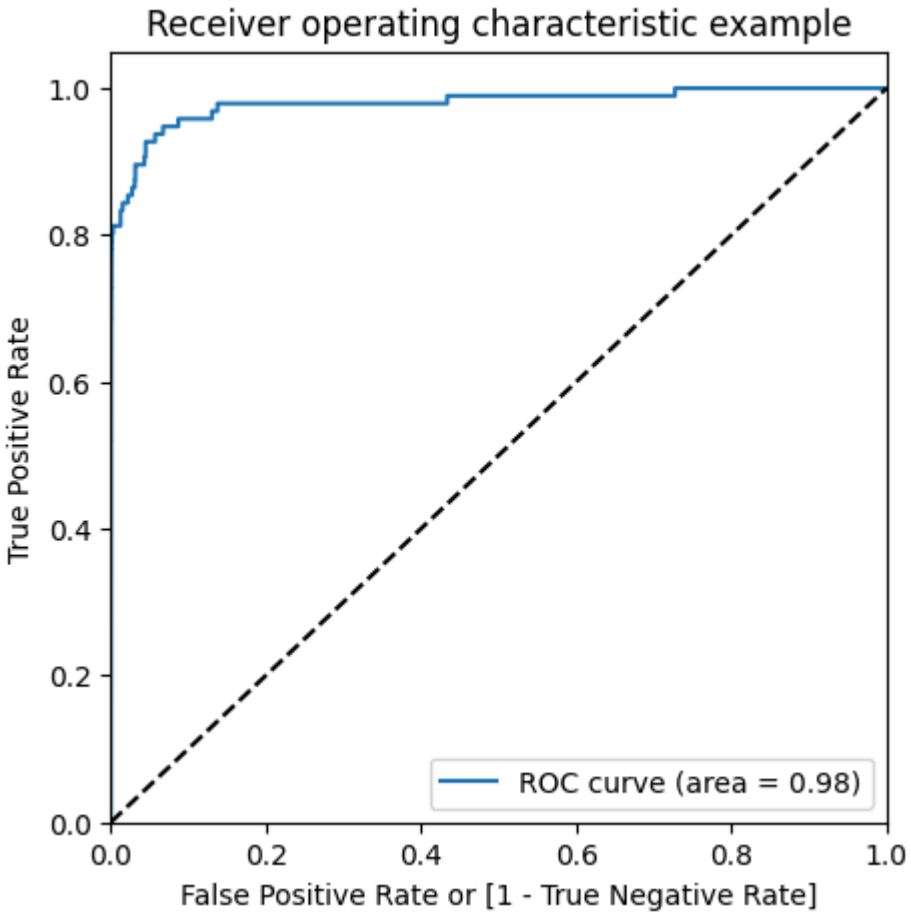
```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     56866
           1       0.05      0.85      0.09        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9792754750934947
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 1.0
  - Sensitivity = 1.0

- Specificity = 1.0
- ROC-AUC = 1.0

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

Oversampling

In [ ]:
```
# Importing oversampler library
from imblearn.over_sampling import RandomOverSampler
```

In [ ]:
```
# instantiating the random oversampler
ros = RandomOverSampler()
# resampling X, y
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)
```

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

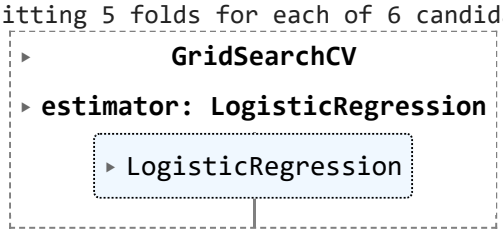
```
Before sampling class distribution:- Counter({0: 227449, 1: 396})
New class distribution:- Counter({0: 227449, 1: 227449})
```

Logistic Regression

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

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

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

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

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

Out[ ]:

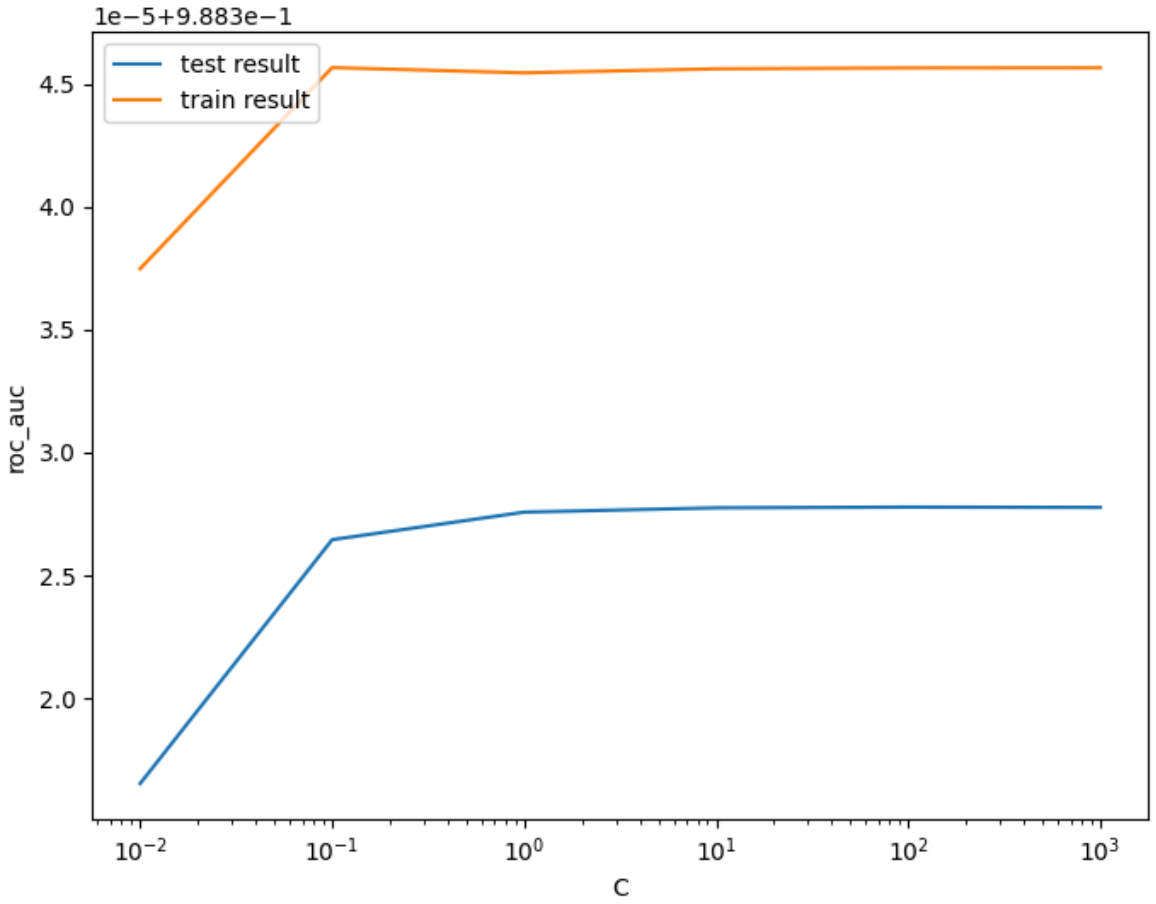


In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split
0	3.019965	0.728350	0.055811	0.010448	0.01	{'C': 0.01}	
1	2.162980	0.074955	0.036827	0.002772	0.1	{'C': 0.1}	
2	2.068724	0.116851	0.038328	0.002808	1	{'C': 1}	
3	2.268915	0.160516	0.045153	0.005852	10	{'C': 10}	
4	2.186963	0.119889	0.041423	0.002915	100	{'C': 100}	
5	2.184774	0.096105	0.039226	0.002758	1000	{'C': 1000}	

In [ ]:
```
# 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')
```



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

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

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

Logistic regression with optimal C

In [ ]:
```
# Instantiate the model with best C
logistic_bal_ros = LogisticRegression(C=0.1)
```

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

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = logistic_bal_ros_model.predict(X_train_ros)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_pred)
print(confusion)
```

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

In [ ]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
```

```
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [ ]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_ros, y_train_pred))

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

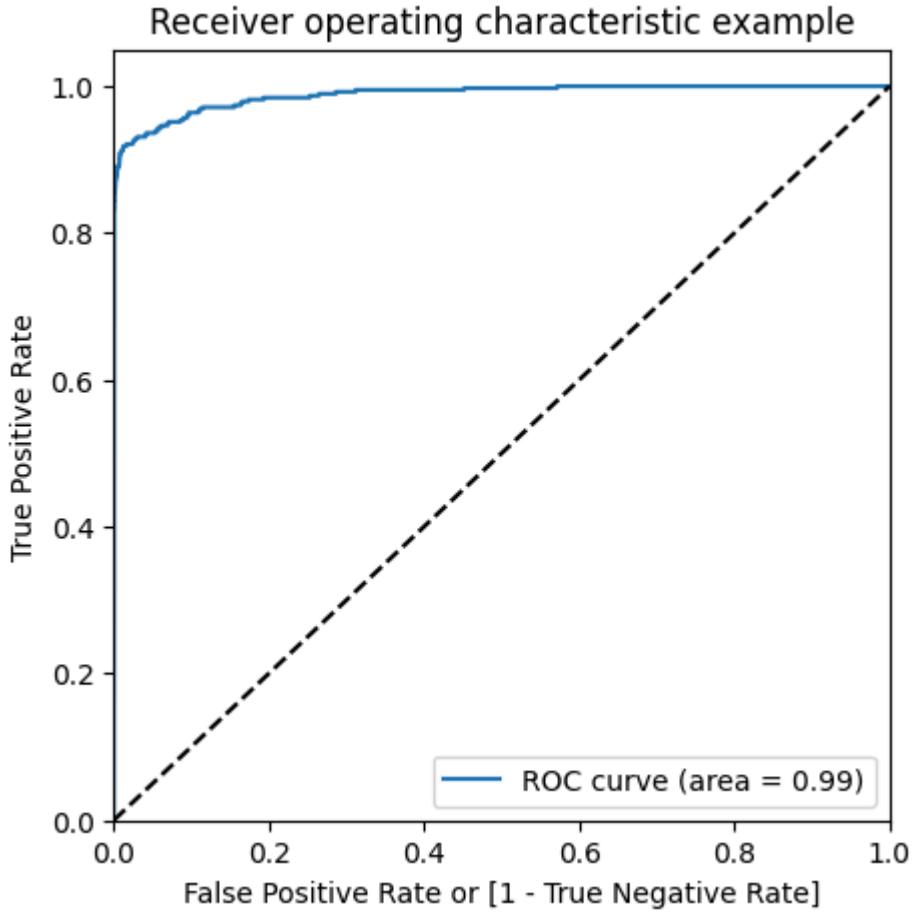
```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95    227449
           1       0.98      0.92      0.95    227449

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

Out[ ]:
```
0.9883382343883345
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Prediction on the test set
y_test_pred = logistic_bal_ros_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55531  1335]
 [   11    85]]
```

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

In [ ]:
```
# 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:- 0.9763702117200941
Sensitivity:- 0.8854166666666666
Specificity:- 0.9765237576055992
```

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

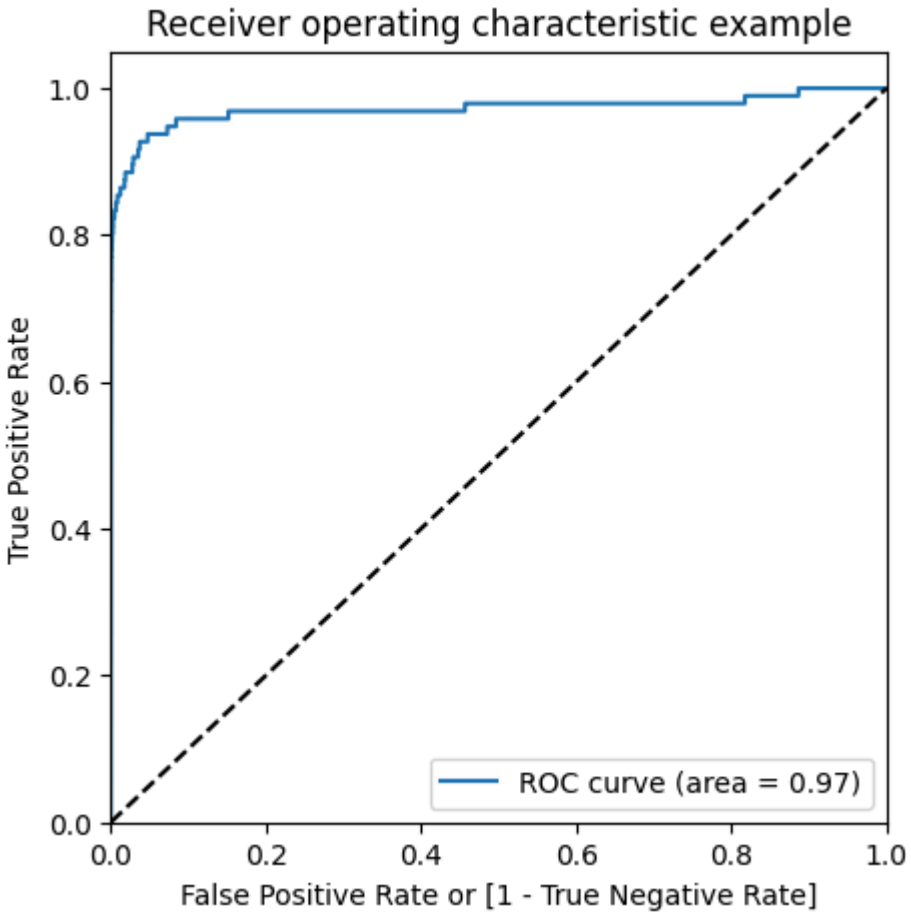
```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.06      0.89      0.11        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9714092120071747
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.95
  - Sensitivity = 0.92

- Specificity = 0.97
- ROC = 0.98

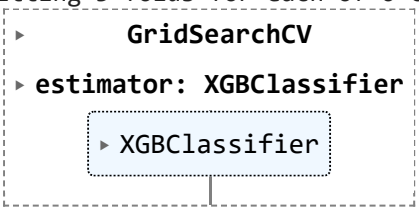
- Test set
  - Accuracy = 0.97
  - Sensitivity = 0.89
  - Specificity = 0.97
  - ROC = 0.97

XGBoost

In [ ]:
```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

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


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

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

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

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

Out[ ]:



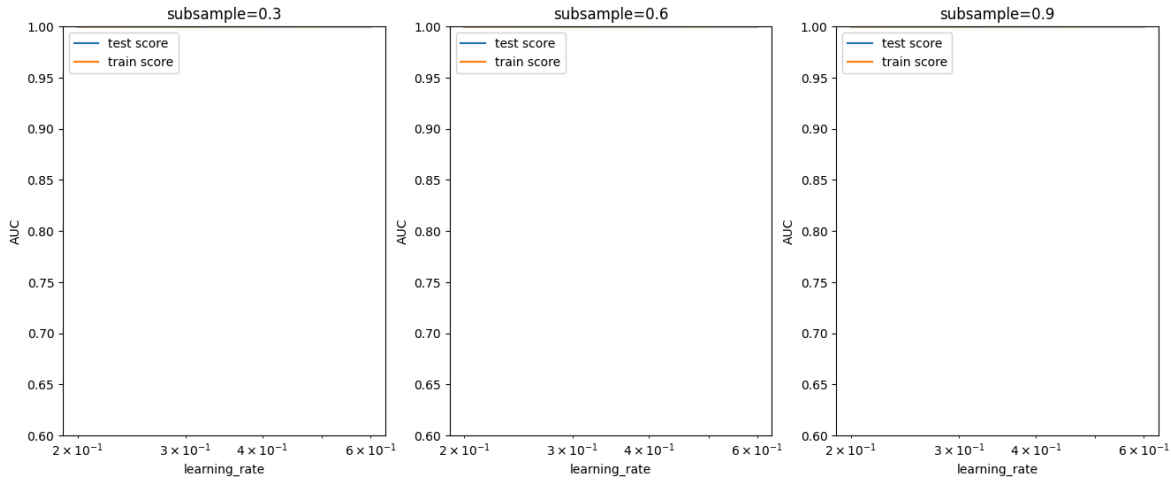
In [ ]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	p
0	3.700946	0.591655	0.144920	0.034687	0.2	
1	5.044792	0.300841	0.141428	0.022726	0.2	
2	4.751309	1.015213	0.111906	0.001667	0.2	
3	4.852920	0.447064	0.168766	0.043294	0.6	
4	5.497935	0.152418	0.150439	0.026223	0.6	
5	5.396589	0.574716	0.149871	0.035196	0.6	

In [ ]:

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

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


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


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

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



Model with optimal hyperparameters

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

In [ ]:
```
model_cv.best_params_
```

Out[ ]:
```
{'learning_rate': 0.6, 'subsample': 0.3}
```

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

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

Out[ ]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = xgb_bal_ros_model.predict(X_train_ros)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_ros)
print(confusion)
```

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

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

In [ ]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train_ros, y_train_pred))

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

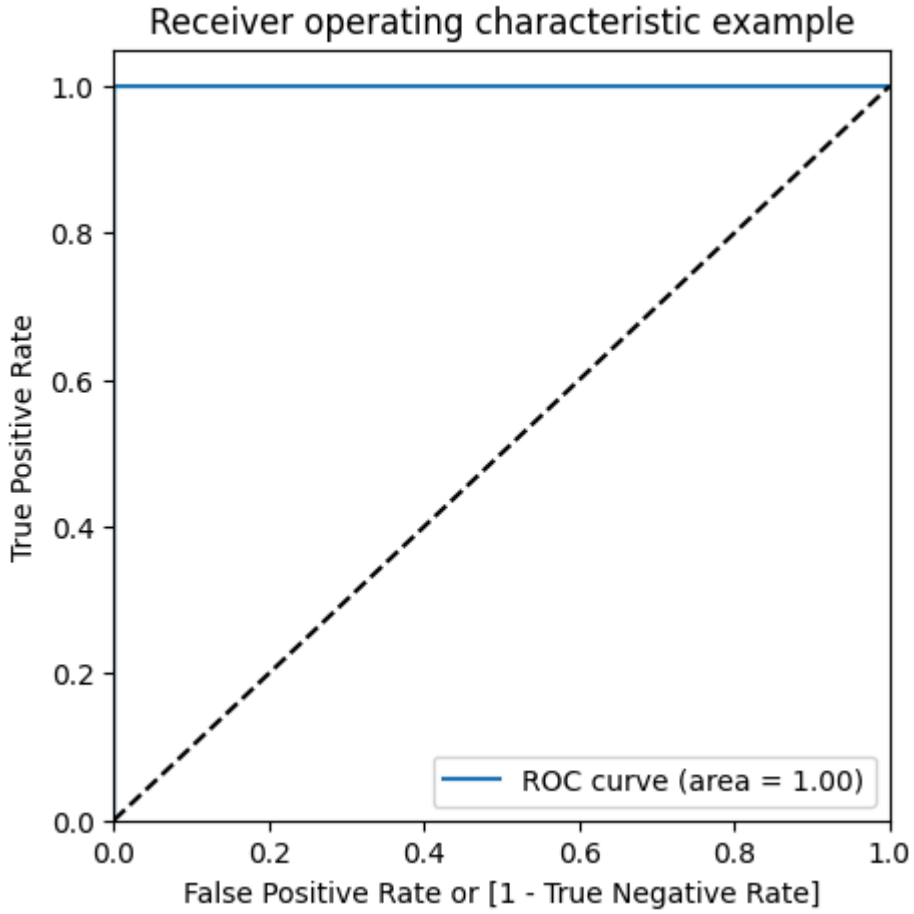
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227449

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

Out[ ]:
```
1.0
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Predictions on the test set
y_test_pred = xgb_bal_ros_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

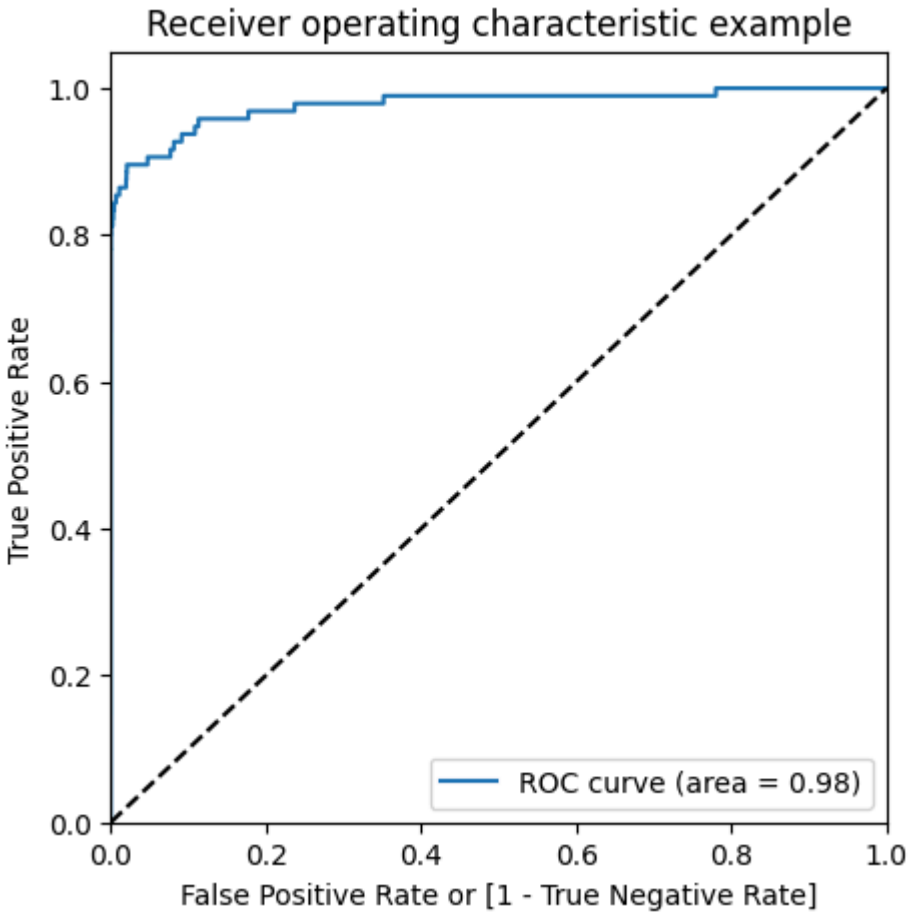
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.86      0.78      0.82        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9775024839095418
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 1.0
  - Sensitivity = 1.0

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

SMOTE (Synthetic Minority Oversampling Technique)

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

In [ ]:
```
# Importing SMOTE
from imblearn.over_sampling import SMOTE
```

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

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

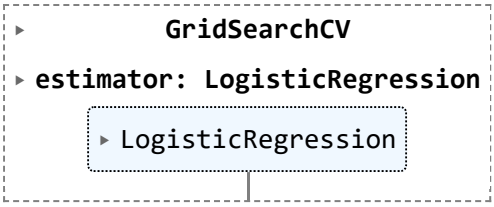
```
Before SMOTE oversampling X_train shape= (227845, 29)
After SMOTE oversampling X_train shape= (454898, 29)
```

Logistic Regression

In [ ]:
```
# 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
```

Out[ ]:

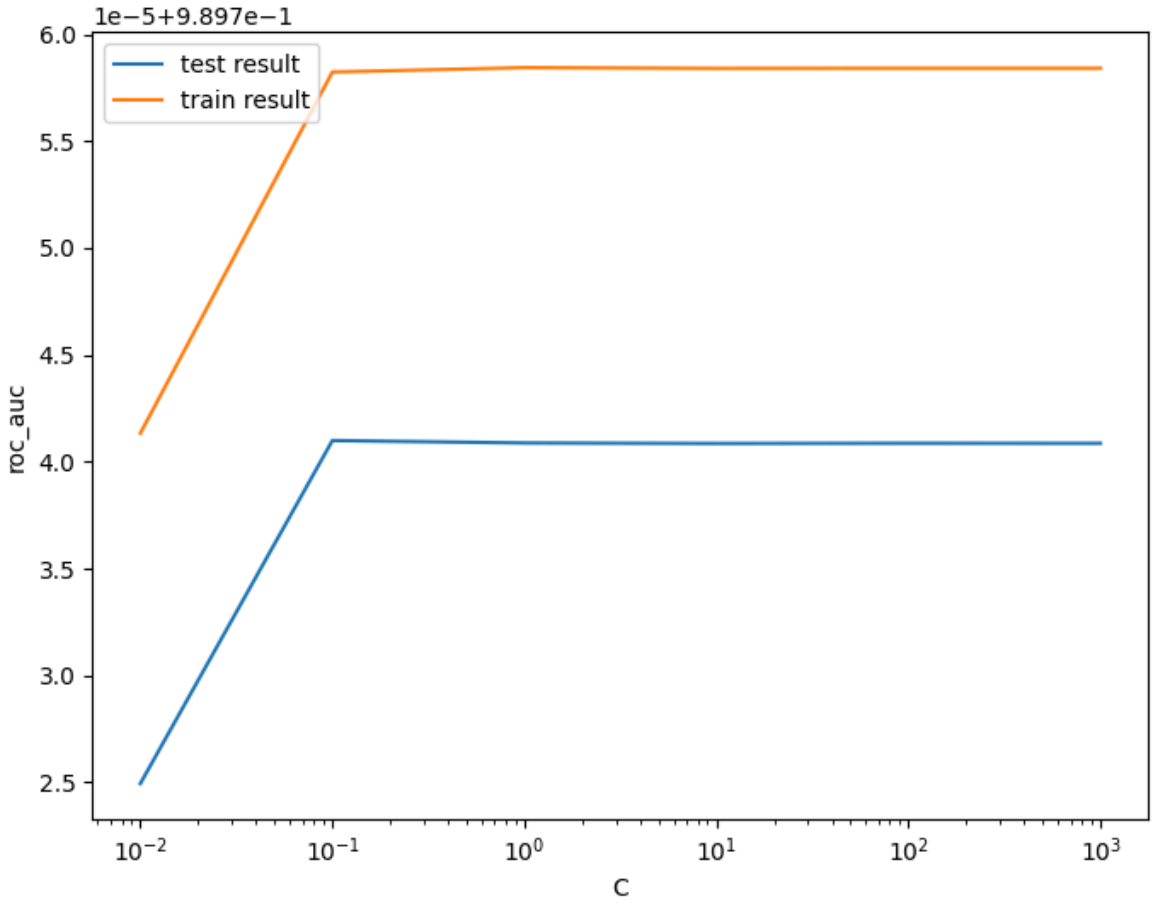


In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split
0	4.350599	0.614689	0.101291	0.034046	0.01	{'C': 0.01}	
1	3.001161	0.427049	0.083746	0.058739	0.1	{'C': 0.1}	
2	1.915668	0.158935	0.050117	0.023027	1	{'C': 1}	
3	2.146231	0.321787	0.070252	0.049219	10	{'C': 10}	
4	2.051609	0.502301	0.046772	0.013744	100	{'C': 100}	
5	2.154828	0.204071	0.062755	0.044003	1000	{'C': 1000}	

In [ ]:
```
# 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')
```



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

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

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

Logistic regression with optimal C

In [ ]:
```
# Instantiate the model with best C
logistic_bal_smote = LogisticRegression(C=0.1)
```

In [ ]:
```
# 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

In [ ]:
```
# Predictions on the train set
y_train_pred = logistic_bal_smote_model.predict(X_train_smote)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
print(confusion)
```

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

In [ ]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
```

```
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

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

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

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

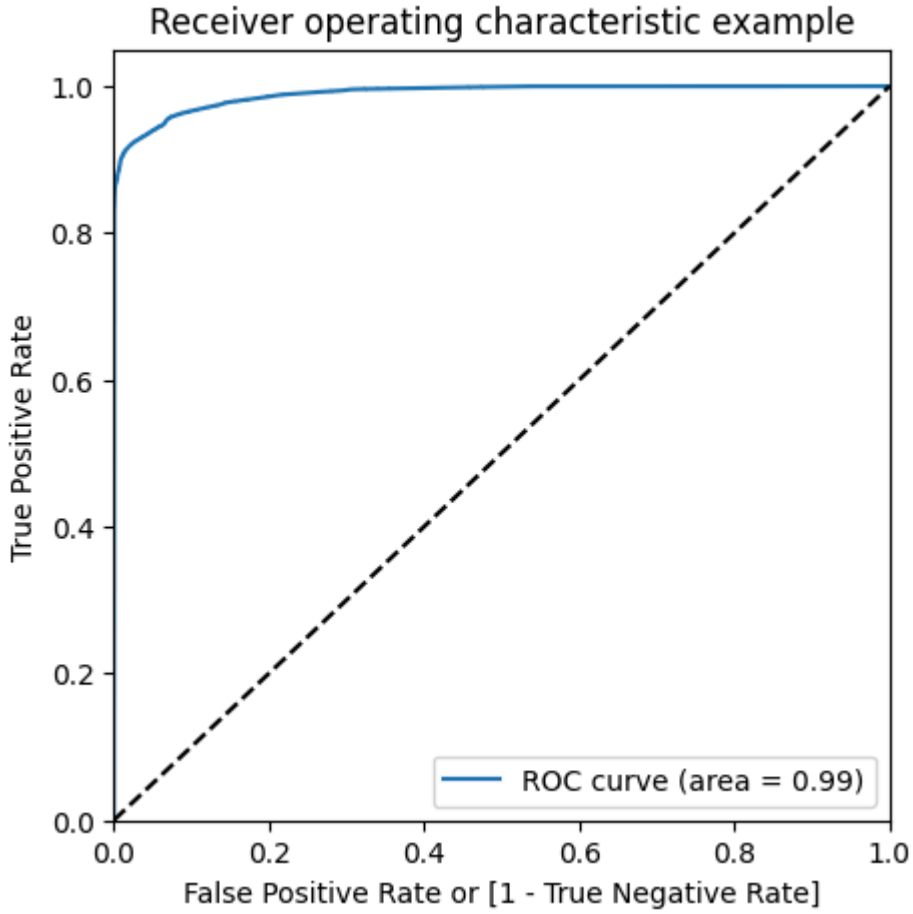
```
Accuracy:- 0.9489314087993352
Sensitivity:- 0.9222111330452102
Specificity:- 0.9756516845534603
```

In [ ]:
```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95    227449
           1       0.97      0.92      0.95    227449

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

In [ ]:
```
# Predicted probability
y_train_pred_proba_log_bal_smote = logistic_bal_smote_model.predict_proba(X_trai
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba_log_bal_smote)
```



Prediction on the test set

In [ ]:
```
# Prediction on the test set
y_test_pred = logistic_bal_smote_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

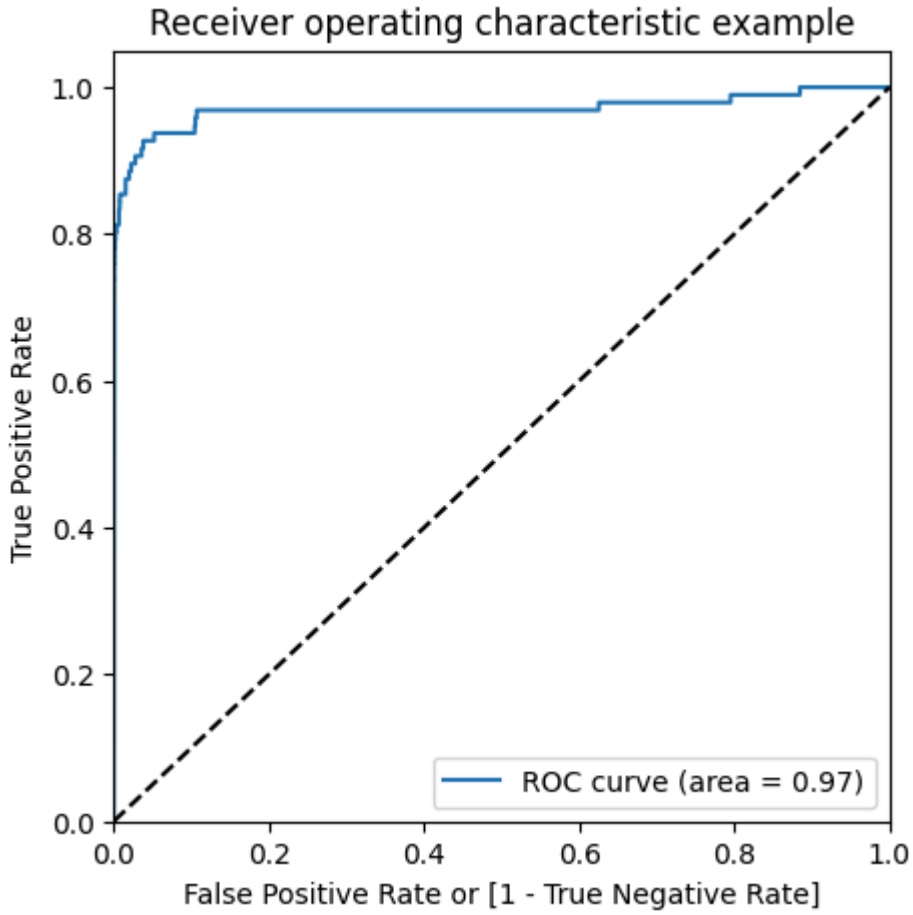
In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     56866
           1       0.06      0.90      0.11        96

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

ROC on the test set

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

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

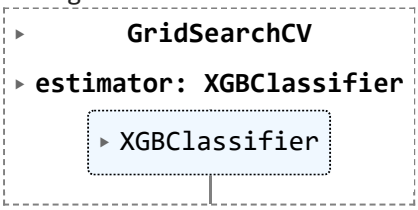
- Accuracy = 0.97
- Sensitivity = 0.90
- Specificity = 0.99
- ROC = 0.97

XGBoost

In [ ]:
```
# 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
```

Out[ ]:



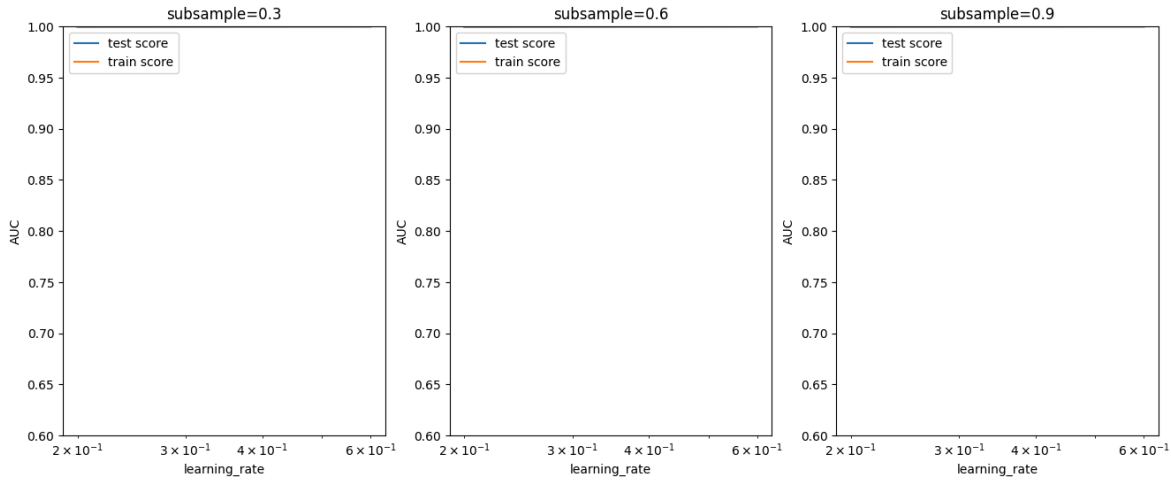
In [ ]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	p
0	4.830210	0.723539	0.153594	0.018271	0.2	
1	5.653752	0.740742	0.153280	0.030025	0.2	
2	5.075618	0.739584	0.200497	0.054708	0.2	
3	6.559366	1.417073	0.250432	0.058185	0.6	
4	5.646797	0.922618	0.180053	0.011094	0.6	
5	5.549089	1.385095	0.179097	0.053972	0.6	

In [ ]:

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

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


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


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

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



Model with optimal hyperparameters

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

In [ ]:
```
model_cv.best_params_
```

Out[ ]:
```
{'learning_rate': 0.6, 'subsample': 0.6}
```

In [ ]:
```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we
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)
```

Out[ ]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = xgb_bal_smote_model.predict(X_train_smote)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
```

```
print(confusion)
```

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

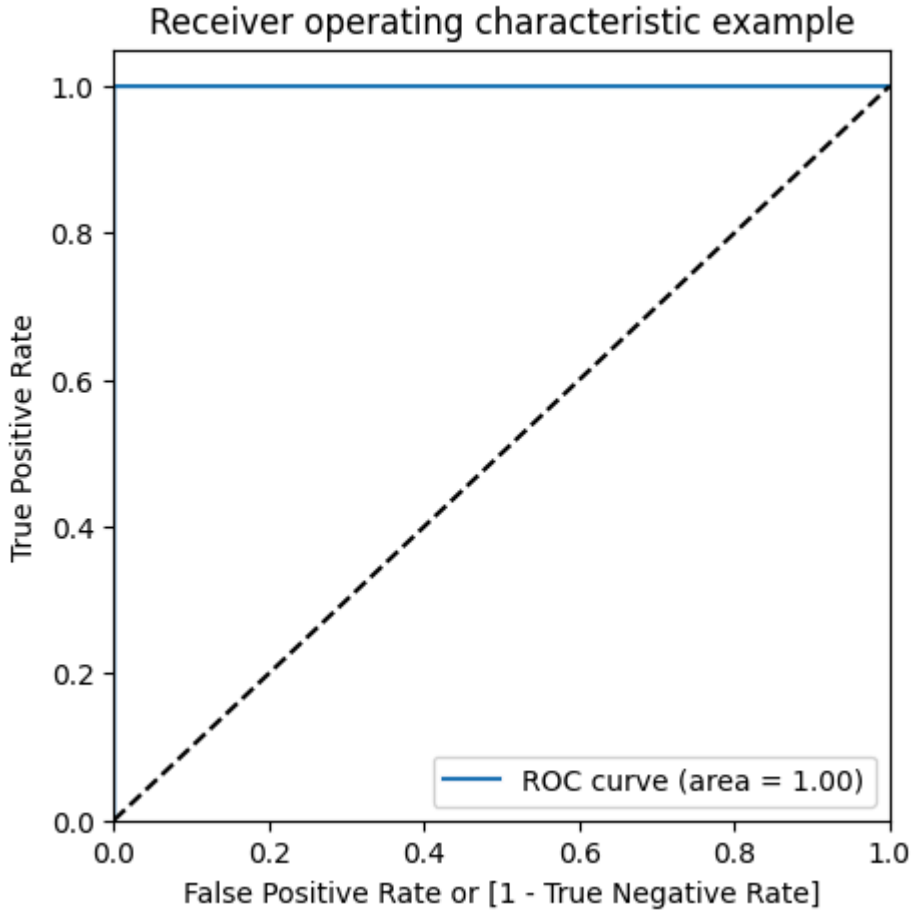
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227449

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_smote, y_train_pred_proba)
auc
```

Out[ ]:
```
0.9999999890785479
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Predictions on the test set
y_test_pred = xgb_bal_smote_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56833    33]
 [   20    76]]
```

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

In [ ]:
```
# 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:- 0.9990695551420246
Sensitivity:- 0.7916666666666666
Specificity:- 0.9994196883902507
```

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

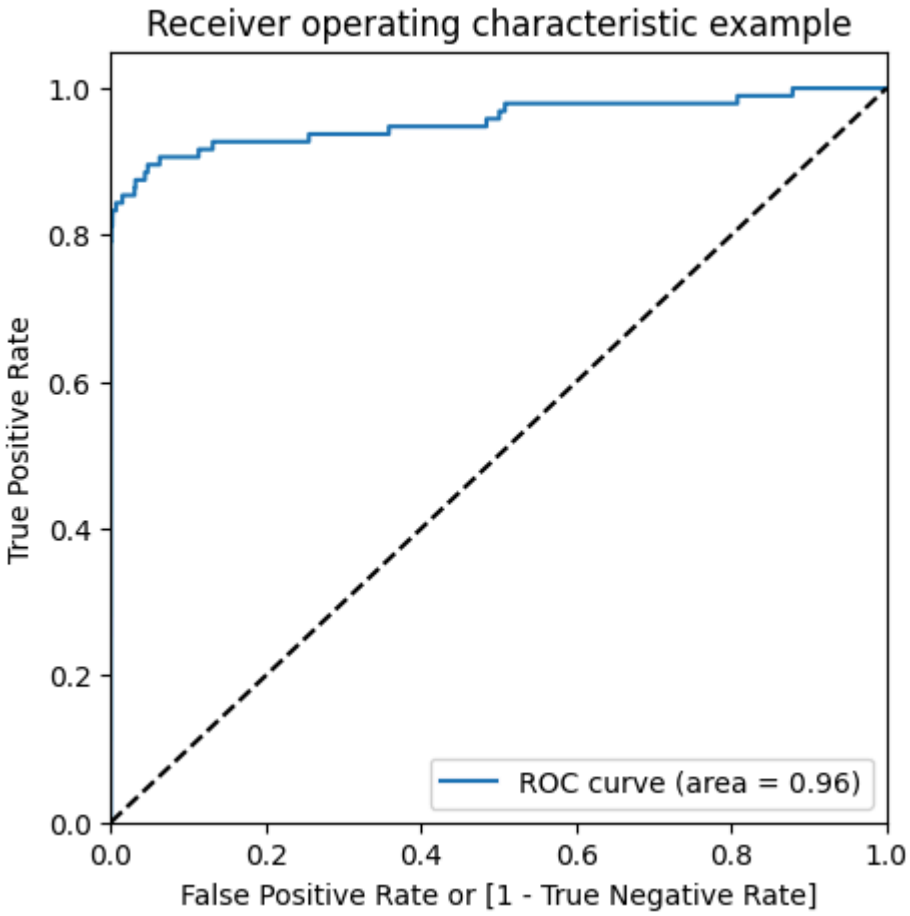
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.70      0.79      0.74        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9553290117703608
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.99
  - Sensitivity = 1.0

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

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

AdaSyn (Adaptive Synthetic Sampling)

In [ ]:
```
# Importing adasyn
from imblearn.over_sampling import ADASYN
```

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

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

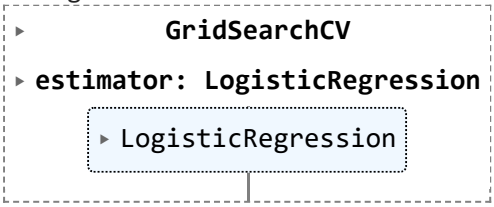
```
Before sampling class distribution:- Counter({0: 227449, 1: 396})
New class distribution:- Counter({0: 227449, 1: 227448})
```

Logistic Regression

In [ ]:
```
# 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
```

Out[ ]:

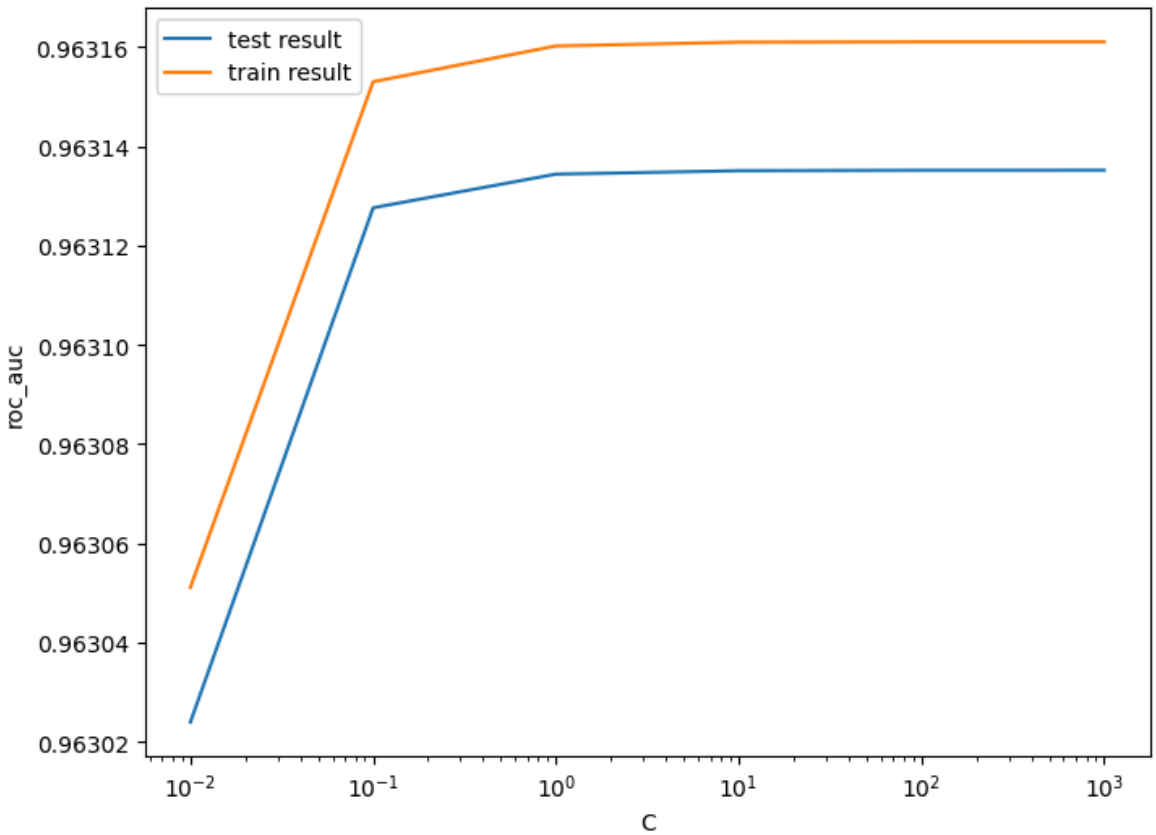


In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split
0	2.829567	0.559018	0.129948	0.010375	0.01	{'C': 0.01}	
1	2.194193	0.401066	0.106578	0.028898	0.1	{'C': 0.1}	
2	2.973076	0.186132	0.092796	0.010743	1	{'C': 1}	
3	2.391114	0.050318	0.093079	0.003281	10	{'C': 10}	
4	2.126840	0.358569	0.092225	0.015129	100	{'C': 100}	
5	2.536523	0.508635	0.136290	0.053766	1000	{'C': 1000}	

In [ ]:
```
# 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')
```



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

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

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

Logistic regression with optimal C

In [ ]:
```
# Instantiate the model with best C
logistic_bal_adasyn = LogisticRegression(C=1000)
```

In [ ]:
```
# Fit the model on the train set
logistic_bal_adasyn_model = logistic_bal_adasyn.fit(X_train_adasyn, y_train_adas
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = logistic_bal_adasyn_model.predict(X_train_adasyn)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_adasyn, y_train_pred)
print(confusion)
```

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

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

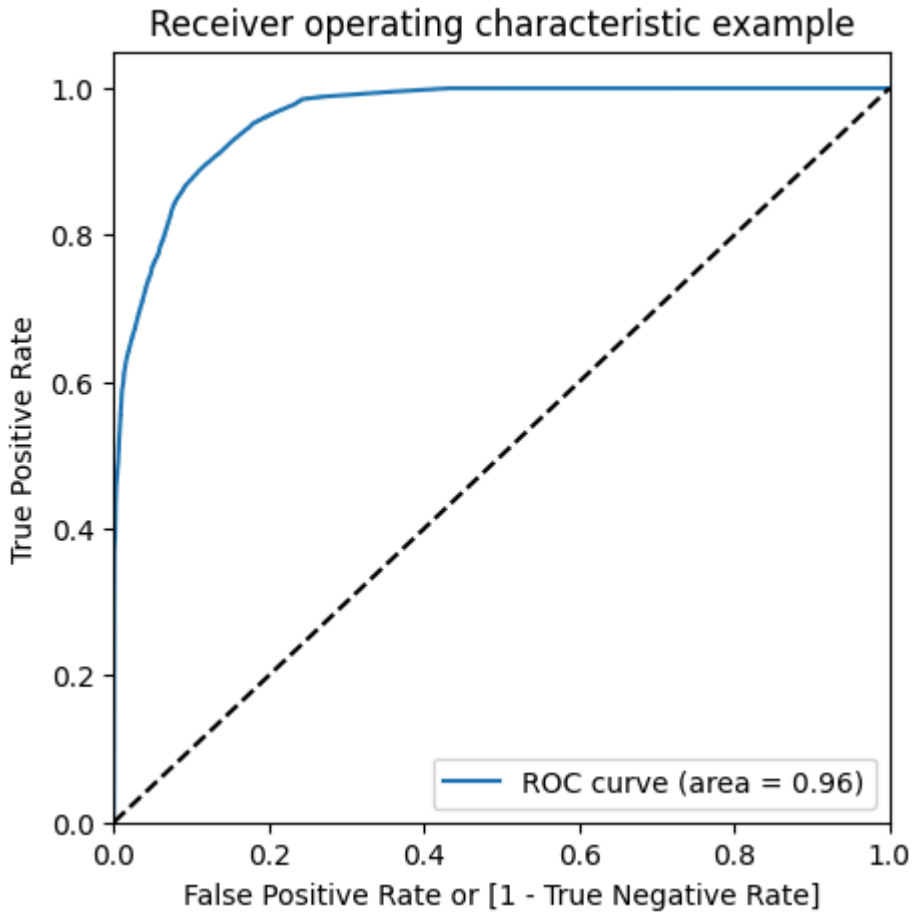
	precision	recall	f1-score	support
0	0.87	0.91	0.89	227449
1	0.91	0.86	0.88	227448
accuracy			0.89	454897
macro avg	0.89	0.89	0.89	454897
weighted avg	0.89	0.89	0.89	454897

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

Out[ ]: 0.9631610160068508

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Prediction on the test set
y_test_pred = logistic_bal_adasyn_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

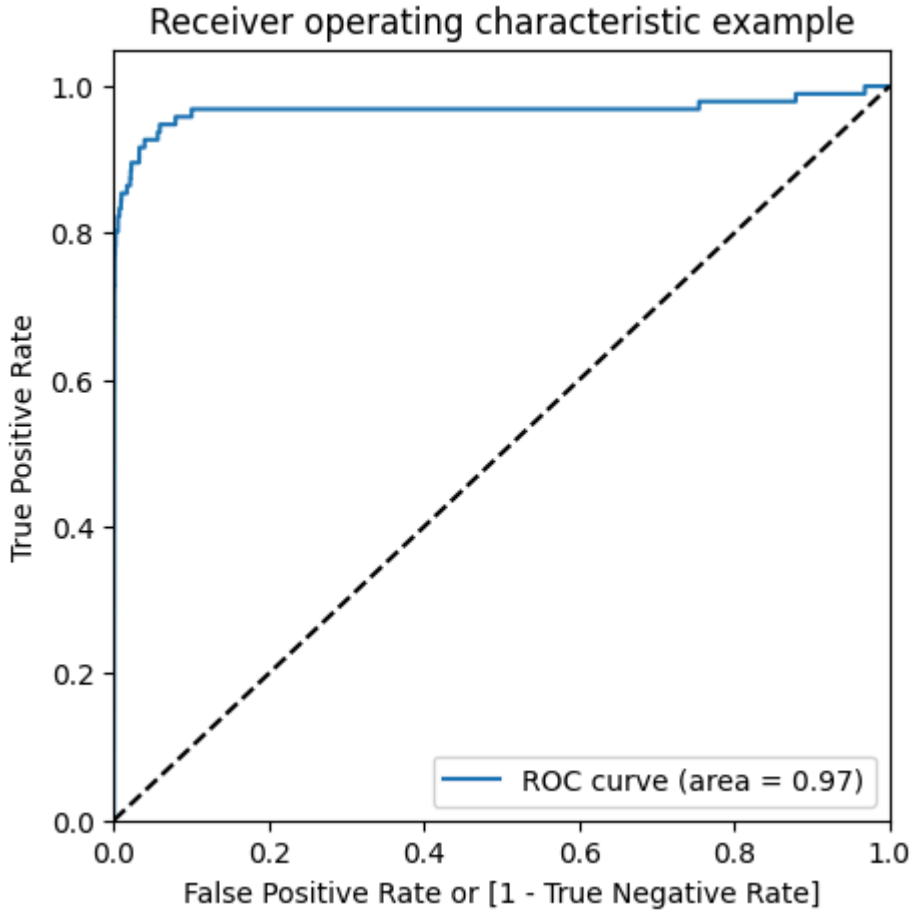
```
              precision    recall  f1-score   support

           0       1.00      0.91      0.95     56866
           1       0.02      0.96      0.03        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.9671573487086602
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

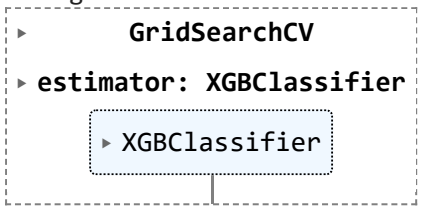
XGBoost

In [ ]:
```
# 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
```

Out[ ]:



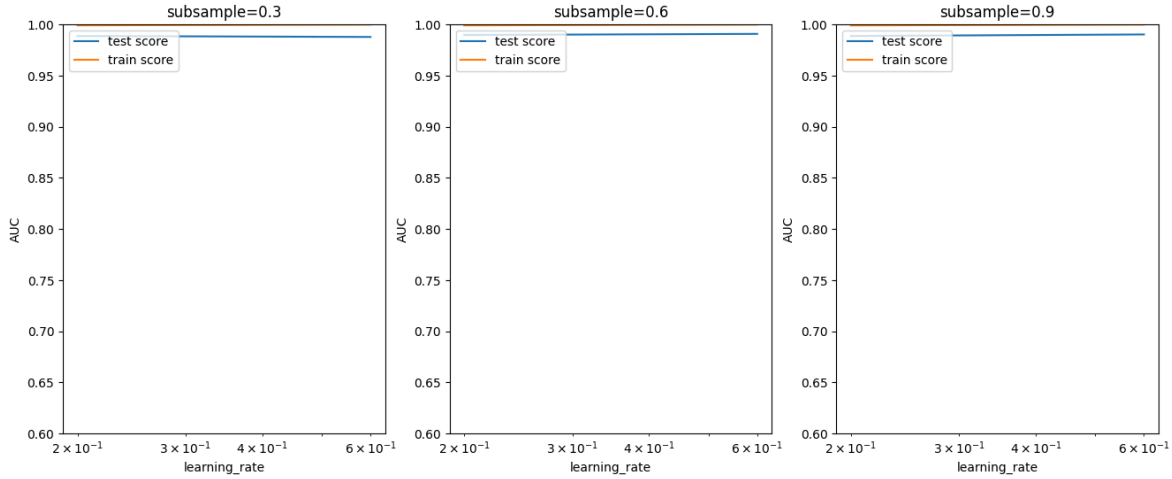
In [ ]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	p
0	3.846837	0.434049	0.108397	0.004416	0.2	
1	3.700305	0.469272	0.136020	0.030375	0.2	
2	4.342210	0.903681	0.153409	0.020724	0.2	
3	3.817883	0.039716	0.116133	0.012226	0.6	
4	9.147208	0.427080	0.254515	0.022016	0.6	
5	7.270713	1.113050	0.227000	0.053686	0.6	

In [ ]:
```
# # plotting
plt.figure(figsize=(16,6))
```

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


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


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

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



In [ ]:
```
model_cv.best_params_
```

Out[ ]:
```
{'learning_rate': 0.6, 'subsample': 0.6}
```

In [ ]:
```
# chosen hyperparameters

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

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

Out[ ]:

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
```

Prediction on the train set

In [ ]:
```
# Predictions on the train set
y_train_pred = xgb_bal_adasyn_model.predict(X_train_adasyn)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_adasyn, y_train_adasyn)
print(confusion)
```

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

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

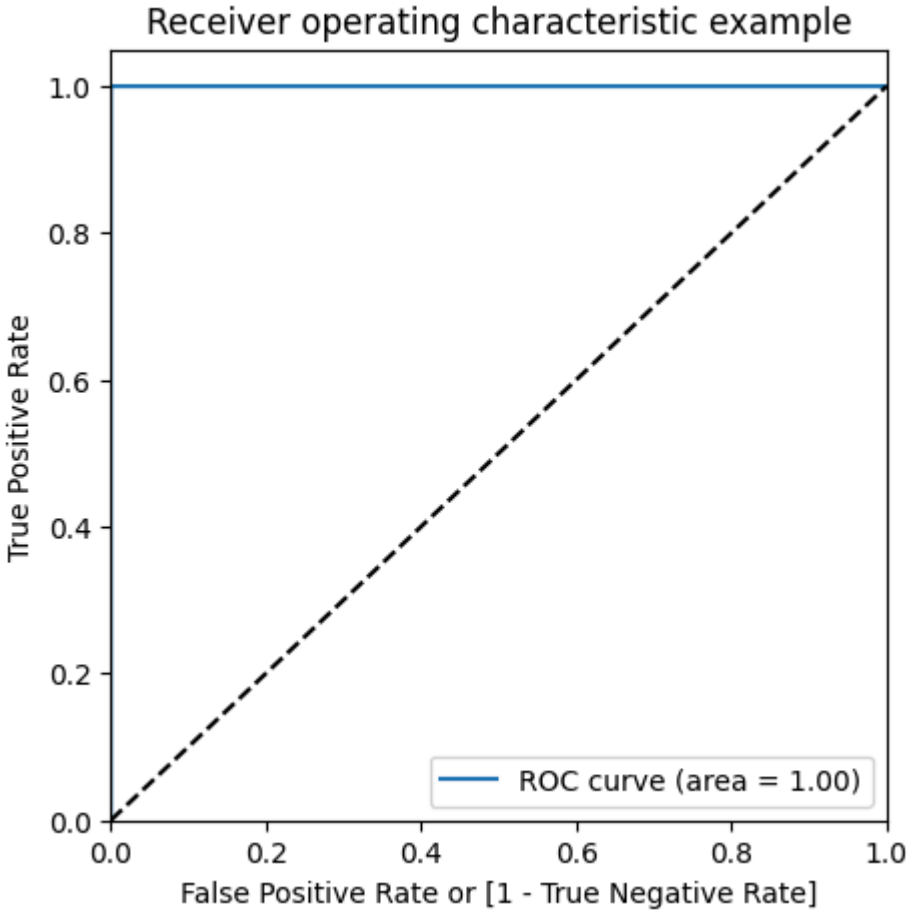
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       1.00      1.00      1.00    227448

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

Out[ ]: 1.0

In [ ]:
```
# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

In [ ]:
```
# Predictions on the test set
y_test_pred = xgb_bal_adasyn_model.predict(X_test)
```

In [ ]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56828    38]
 [   22    74]]
```

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

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

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

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

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

In [ ]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

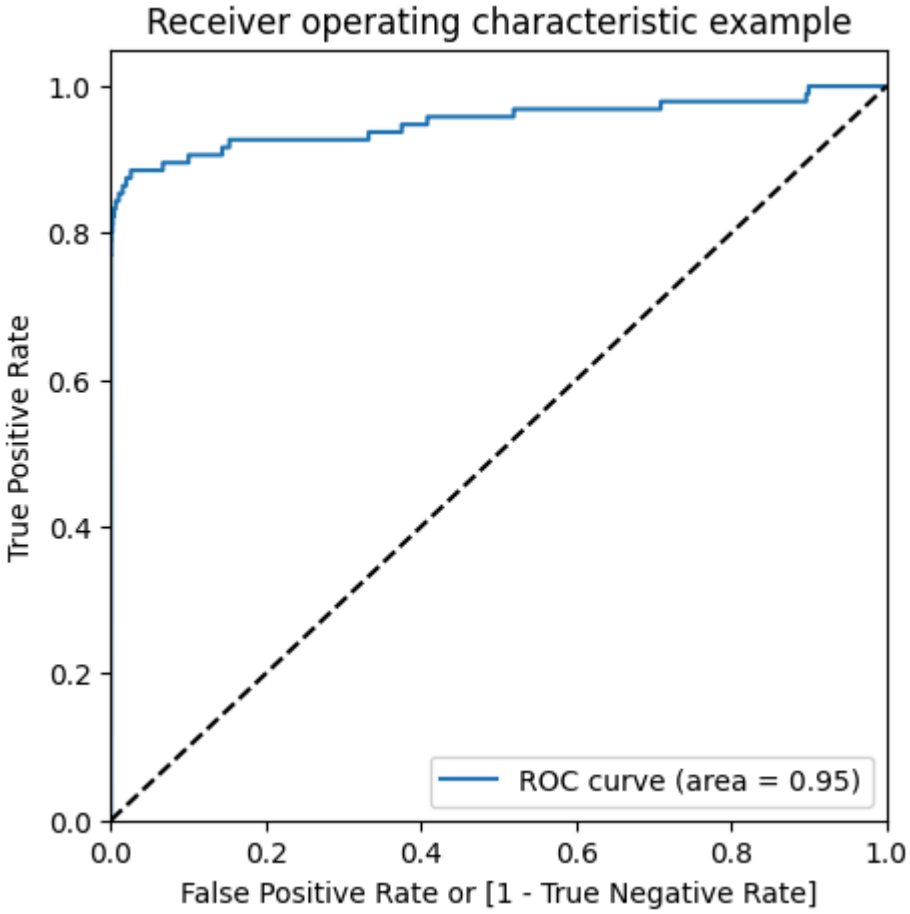
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.66      0.77      0.71        96

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

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

In [ ]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[ ]:
```
0.951038589256615
```

In [ ]:
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

Choosing best model on the balanced data

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

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

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

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

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

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

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

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

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

Cost benefit analysis

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

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

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

Summary to the business

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

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

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

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