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 [3]:

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

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

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

In [4]:

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

Exploratory data analysis

Reading and understanding the data

In [5]:

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

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.6
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.1
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.5

In [6]:

```
df.shape
```

Out[6]:

```
(284807, 31)
```

In [7]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
Time      284807 non-null float64
V1        284807 non-null float64
V2        284807 non-null float64
V3        284807 non-null float64
V4        284807 non-null float64
V5        284807 non-null float64
V6        284807 non-null float64
V7        284807 non-null float64
V8        284807 non-null float64
V9        284807 non-null float64
V10       284807 non-null float64
V11       284807 non-null float64
V12       284807 non-null float64
V13       284807 non-null float64
V14       284807 non-null float64
V15       284807 non-null float64
V16       284807 non-null float64
V17       284807 non-null float64
V18       284807 non-null float64
V19       284807 non-null float64
V20       284807 non-null float64
V21       284807 non-null float64
V22       284807 non-null float64
V23       284807 non-null float64
V24       284807 non-null float64
V25       284807 non-null float64
V26       284807 non-null float64
V27       284807 non-null float64
V28       284807 non-null float64
Amount    284807 non-null float64
Class     284807 non-null int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

In [8]:
```
df.describe()
```

Out[8]:

	Time	V1	V2	V3	V4	V5	V6	V7	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15	2.010663e-15	-1.694249e-15	-1
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2

Handling missing values

Handling missing values in columns

In [9]:
```
# Cheking percent of missing values in columns
df_missing_columns = (round(((df.isnull().sum()/len(df.index))*100),2).to_frame('null')).sort_values('null', as
df_missing_columns
```

Out[9]:

	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 [10]:

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

Out[10]:

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

In [11]:

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

Out[11]:

```
99.83
```

In [12]:

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

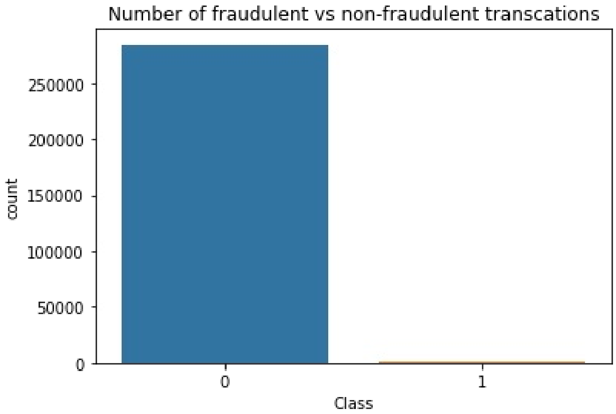
Out[12]:

```
0.17
```

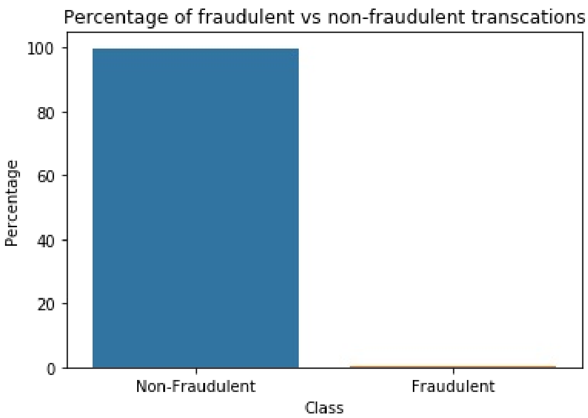
We can see that there is only 0.17% frauds. We will take care of the class imbalance later.

In [13]:

```
# 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 [14]:
```
# Bar plot for the percentage of fraudulent vs non-fraudulent transcations
fraud_percentage = {'Class':['Non-Fraudulent', 'Fraudulent'], 'Percentage':[normal_share, fraud_share]}
df_fraud_percentage = pd.DataFrame(fraud_percentage)
sns.barplot(x='Class',y='Percentage', data=df_fraud_percentage)
plt.title('Percentage of fraudulent vs non-fraudulent transcations')
plt.show()
```



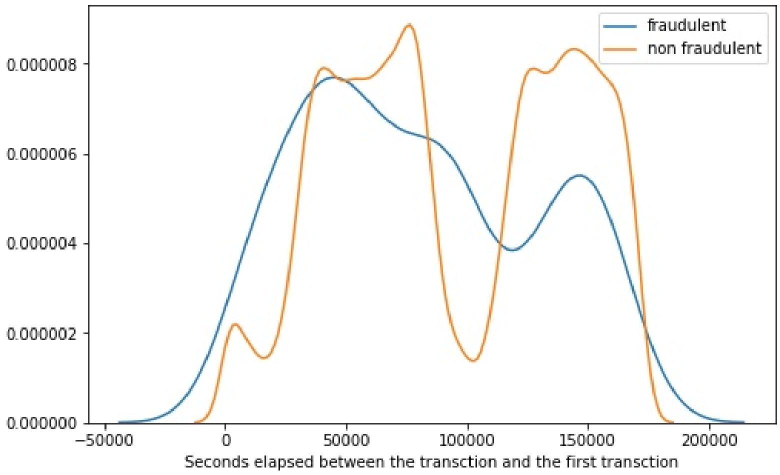
Outliers treatment

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

Observe the distribution of classes with time

In [15]:
```
# Creating fraudulent dataframe
data_fraud = df[df['Class'] == 1]
# Creating non fraudulent dataframe
data_non_fraud = df[df['Class'] == 0]
```

In [16]:
```
# 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()
```



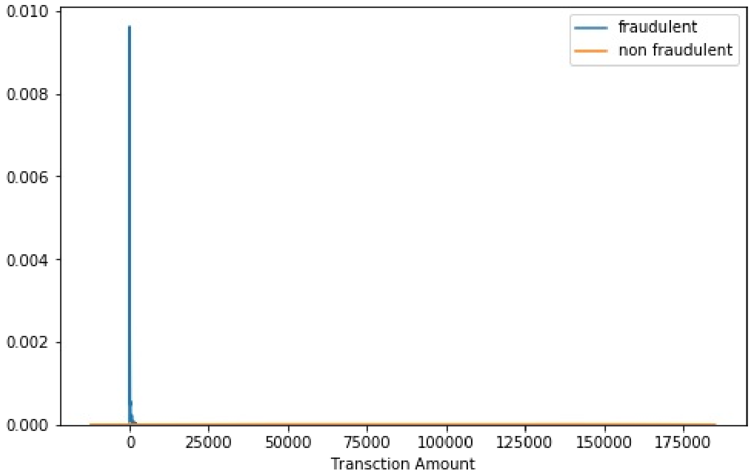
Analysis

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 [17]:
```
# Dropping the Time column
df.drop('Time', axis=1, inplace=True)
```

Observe the distribution of classes with amount

In [18]:
```
# 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 [19]:
```
# Import library
from sklearn.model_selection import train_test_split
```

In [20]:
```
# Putting feature variables into X
X = df.drop(['Class'], axis=1)
```

In [21]:
```
# Putting target variable to y
y = df['Class']
```

In [22]:
```
# Splitting data into train and test set 80:20
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size=0.2, random_state=100)
```

Feature Scaling

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

In [23]:
```
# Standardization method
from sklearn.preprocessing import StandardScaler
```

In [24]:
```
# Instantiate the Scaler
scaler = StandardScaler()
```

In [25]:
```
# Fit the data into scaler and transform
X_train['Amount'] = scaler.fit_transform(X_train[['Amount']])
```

In [26]:
```
X_train.head()
```

Out[26]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	
201788	2.023734	-0.429219	-0.691061	-0.201461	-0.162486	0.283718	-0.674694	0.192230	1.124319	-0.037763	0.308648	0.87
179369	-0.145286	0.736735	0.543226	0.892662	0.350846	0.089253	0.626708	-0.049137	-0.732566	0.297692	0.519027	0.04
73138	-3.015846	-1.920606	1.229574	0.721577	1.089918	-0.195727	-0.462586	0.919341	-0.612193	-0.966197	1.106534	1.02
208679	1.851980	-1.007445	-1.499762	-0.220770	-0.568376	-1.232633	0.248573	-0.539483	-0.813368	0.785431	-0.784316	0.67
206534	2.237844	-0.551513	-1.426515	-0.924369	-0.401734	-1.438232	-0.119942	-0.449263	-0.717258	0.851668	-0.497634	-0.44

Scaling the test set

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

In [27]:
```
# Transform the test set
X_test['Amount'] = scaler.transform(X_test[['Amount']])
X_test.head()
```

Out[27]:

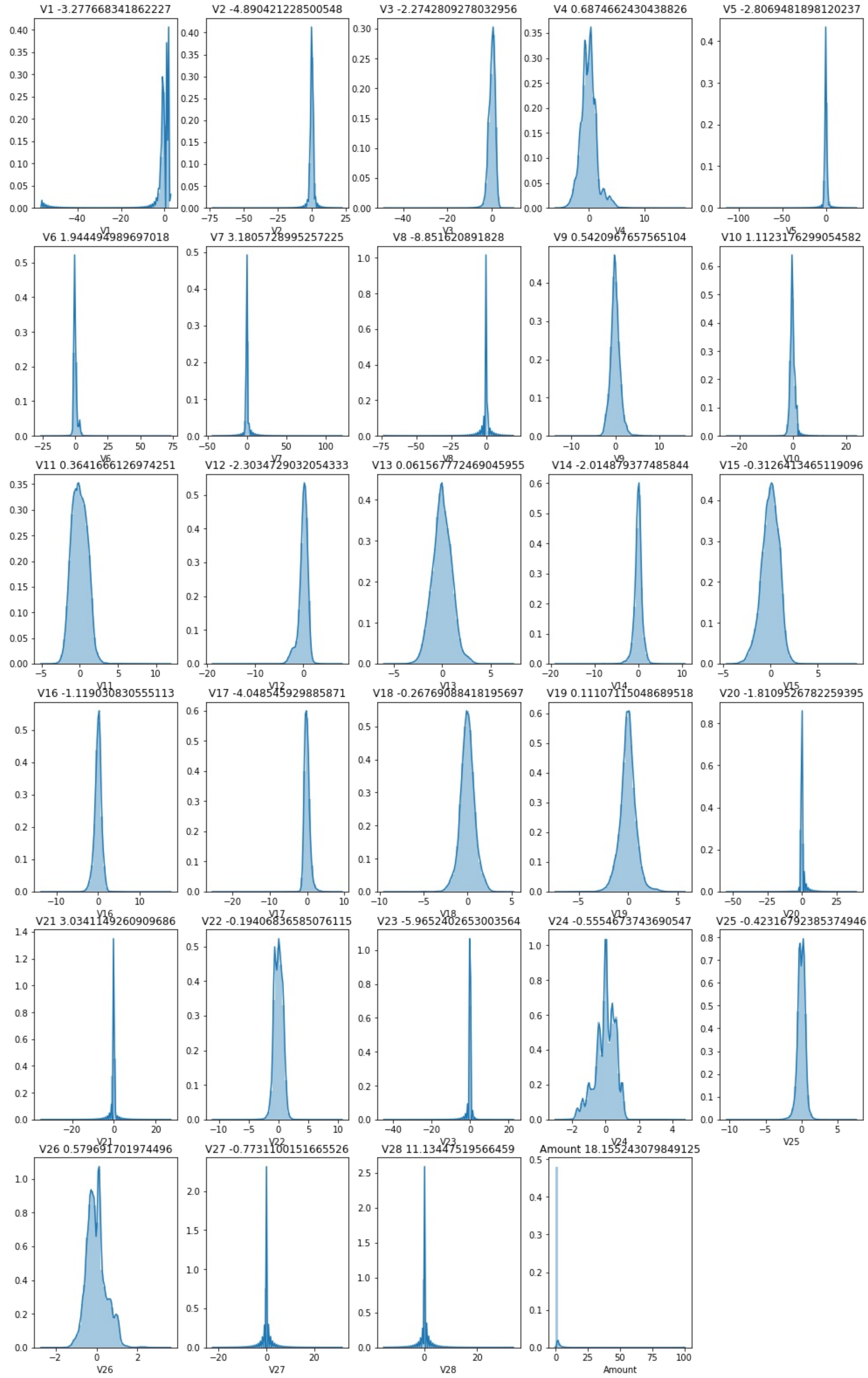
	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	
49089	1.229452	-0.235478	-0.627166	0.419877	1.797014	4.069574	-0.896223	1.036103	0.745991	-0.147304	-0.850459	0.39
154704	2.016893	-0.088751	-2.989257	-0.142575	2.675427	3.332289	-0.652336	0.752811	1.962566	-1.025024	1.126976	-2.41
67247	0.535093	-1.469185	0.868279	0.385462	-1.439135	0.368118	-0.499370	0.303698	1.042073	-0.437209	1.145725	0.90
251657	2.128486	-0.117215	-1.513910	0.166456	0.359070	-0.540072	0.116023	-0.216140	0.680314	0.079977	-1.705327	-0.12
201903	0.558593	1.587908	-2.368767	5.124413	2.171788	-0.500419	1.059829	-0.254233	-1.959060	0.948915	-0.288169	-1.00

Checking the Skewness

In [28]:
```
# Listing the columns
cols = X_train.columns
cols
```

Out[28]:
```
Index(['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11',
       'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21',
       'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount'],
      dtype='object')
```

In [29]:
```
# Plotting the distribution of the variables (skewness) of all the columns
k=0
plt.figure(figsize=(17,28))
for col in cols :
    k=k+1
    plt.subplot(6, 5,k)
    sns.distplot(X_train[col])
    plt.title(col+' '+str(X_train[col].skew()))
```


V1 -3.277668341862227
V2 -4.890421228500548
V3 -2.2742809278032956
V4 0.6874662430438826
V5 -2.8069481898120237
V6 1.944494989697018
V7 3.1805728995257225
V8 -8.851620891828
V9 0.5420967657565104
V10 1.1123176299054582
V11 0.3641666126974251
V12 -2.3034729032054333
V13 0.061567772469045955
V14 -2.014879377485844
V15 -0.3126413465119096
V16 -1.119030830555113
V17 -4.048545929885871
V18 -0.26769088418195697
V19 0.11107115048689518
V20 -1.8109526782259395
V21 3.0341149260909686
V22 -0.19406836585076115
V23 -5.9652402653003564
V24 -0.5554673743690547
V25 -0.42316792385374946
V26 0.579691701974496
V27 -0.7731100151665526
V28 11.13447519566459
Amount 18.155243079849125

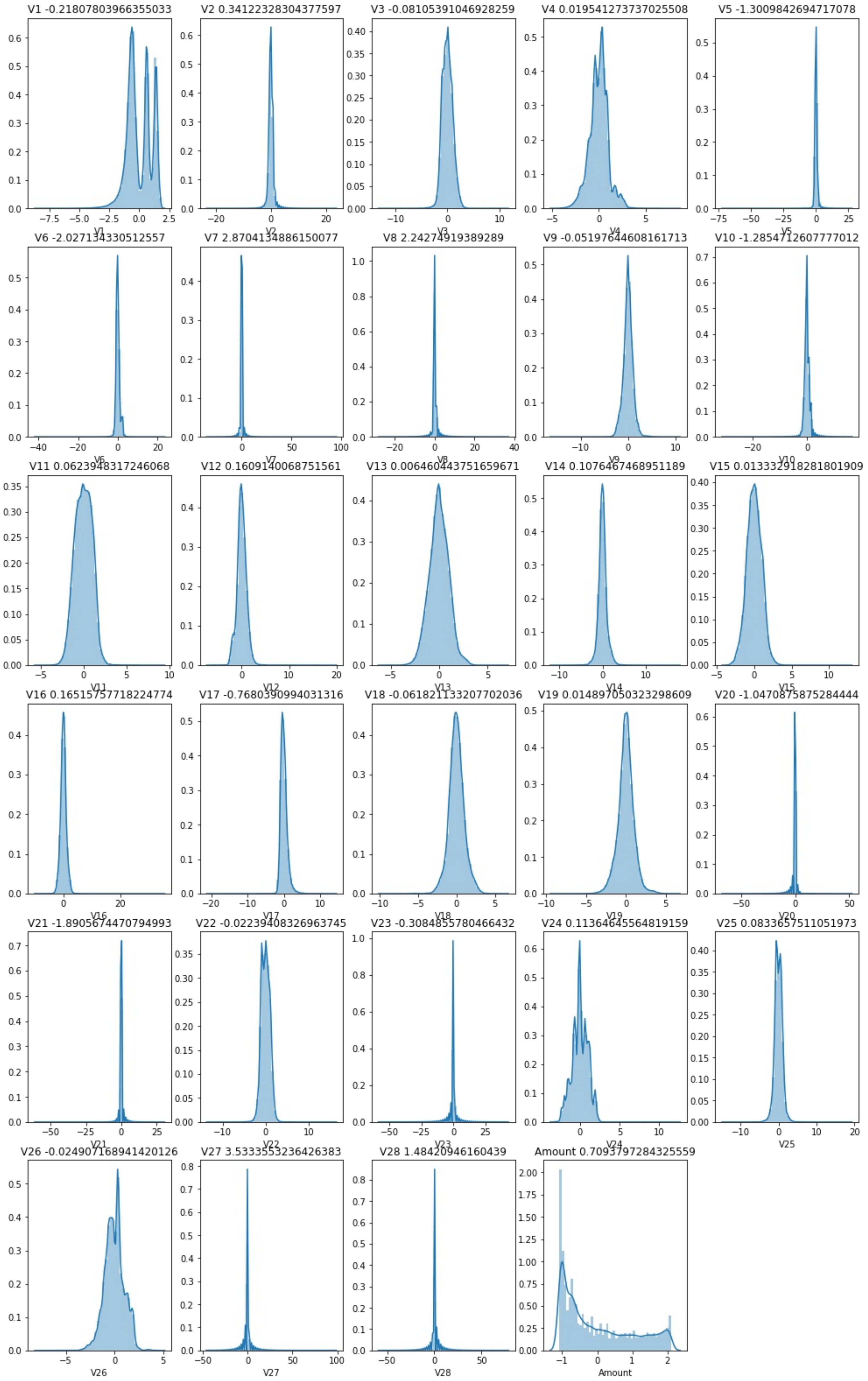
We see that there are many variables, which are heavily skewed. We will mitigate the skewness only for those variables for bringing them into normal distribution.

Mitigate skweness with PowerTransformer

In [31]:
```
# 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 [32]:
```
# Transform the test set
X_test[cols] = pt.transform(X_test)
```

In [33]:
```
# Plotting the distribution of the variables (skewness) of all the columns
k=0
plt.figure(figsize=(17,28))
for col in cols :
    k=k+1
    plt.subplot(6, 5,k)
    sns.distplot(X_train[col])
    plt.title(col+' '+str(X_train[col].skew()))
```


V1 -0.21807803966355033
V2 0.34122328304377597
V3 -0.08105391046928259
V4 0.019541273737025508
V5 -1.3009842694717078
V6 -2.027134330512557
V7 2.8704134886150077
V8 2.24274919389289
V9 -0.05197644608161713
V10 -1.2854712607777012
V11 0.0623948317246068
V12 0.1609140068751561
V13 0.006460443751659671
V14 0.1076467468951189
V15 0.013332918281801909
V16 0.16515757718224774
V17 -0.7680390994031316
V18 -0.061821133207702036
V19 0.014897050323298609
V20 -1.0470875875284444
V21 -1.8905674470794993
V22 -0.022394083269637 45
V23 -0.3084855780466432
V24 0.11364645564819159
V25 0.0833657511051973
V26 -0.024907168941420126
V27 3.5333553236426383
V28 1.48420946160439
Amount 0.7093797284325559
V25
V26
V27
V28
Amount

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

As we have seen that the data is heavily imbalanced, where only 0.17% transctions are fraudulent, we should not consider Accuracy as a good measure for evaluating the model. Because in the case of all the datapoints return a particular class(1/0) irrespective of any prediction, still the model will result more than 99% Accuracy.

Hence, we have to measure the ROC-AUC score for fair evaluation of the model. The ROC curve is used to understand the strength of the model by evaluating the performance of the model at all the classification thresholds. The default threshold of 0.5 is not always the ideal threshold to find the best classification label of the test point. Because the ROC curve is measured at all thresholds, the best threshold would be one at which the TPR is high and FPR is low, i.e., misclassifications are low. After determining the optimal threshold, we can calculate the F1 score of the classifier to measure the precision and recall at the selected threshold.

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

In the dataset we have 284807 datapoints and in the case of Oversampling we would have even more number of datapoints. SVM is not very efficient with large number of datapoints beacuse it takes lot of computational power and resources to make the transformation. When we perform the cross validation with K-Fold for hyperparameter tuning, it takes lot of computational resources and it is very time consuming. Hence, because of the unavailablity of the required resources and time SVM was not tried.

For the same reason Random forest was also not tried for model building in few of the hyperparameter tuning for oversampling technique.

Why KNN was not used for model building?

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

Logistic regression

In [34]:
```
# Importing scikit logistic regression module
from sklearn.linear_model import LogisticRegression
```

In [35]:
```
# 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 [1]:
```
# 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 [40]:
```
# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

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

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

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

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:   43.0s finished
```

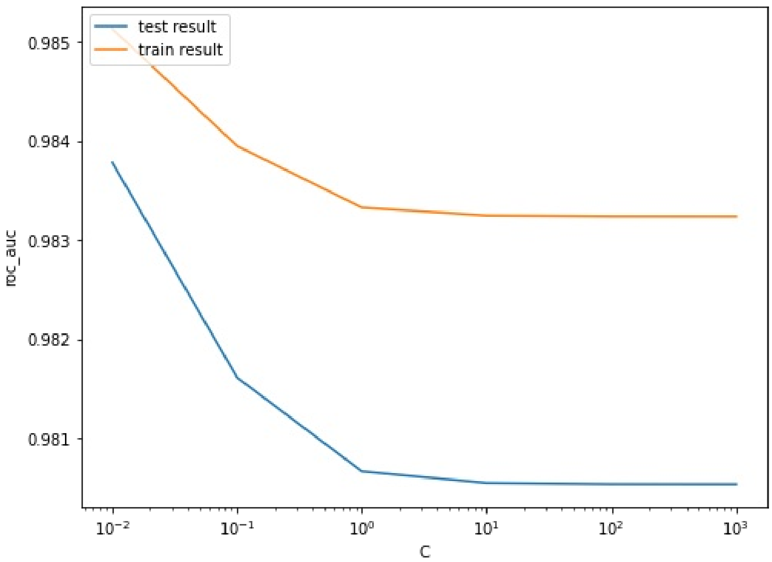
Out[40]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [41]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[41]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test_
0	0.868854	0.035584	0.023276	0.000451	0.01	{'C': 0.01}	0.986923	0.987267	0.9
1	1.270447	0.082098	0.024949	0.004554	0.1	{'C': 0.1}	0.986361	0.987820	0.9
2	1.442430	0.119548	0.023093	0.000241	1	{'C': 1}	0.986199	0.987685	0.9
3	1.442806	0.093519	0.022923	0.000483	10	{'C': 10}	0.986179	0.987669	0.9
4	1.434997	0.079551	0.023676	0.001787	100	{'C': 100}	0.986177	0.987666	0.9
5	1.453797	0.105531	0.022806	0.000277	1000	{'C': 1000}	0.986176	0.987665	0.9

In [42]:
```
# 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 [43]:
```
# 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 0.9837811907775487 at C = 0.01
```

Logistic regression with optimal C

In [38]:
```
# Instantiate the model with best C
logistic_imb = LogisticRegression(C=0.01)
```

In [39]:
```
# Fit the model on the train set
logistic_imb_model = logistic_imb.fit(X_train, y_train)
```

Prediction on the train set

In [40]:
```
# Predictions on the train set
y_train_pred = logistic_imb_model.predict(X_train)
```

In [41]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)
```

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

In [42]:
```
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 [43]:
```
# 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 [46]:
```
# classification_report
print(classification_report(y_train, y_train_pred))
```

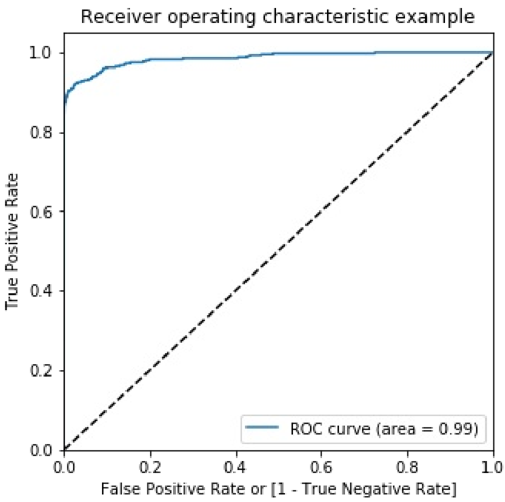
```
              precision    recall  f1-score   support

           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 [47]:
```
# 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 [48]:
```
# Predicted probability
y_train_pred_proba = logistic_imb_model.predict_proba(X_train)[:,1]
```

In [49]:
```
# 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 [50]:
```
# Prediction on the test set
y_test_pred = logistic_imb_model.predict(X_test)
```

In [51]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

In [52]:
```
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 [53]:
```
# 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
```

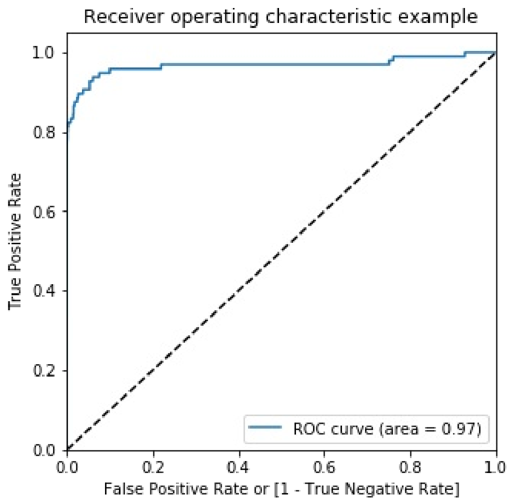
In [54]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              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 [55]:
```
# Predicted probability
y_test_pred_proba = logistic_imb_model.predict_proba(X_test)[:,1]
```

In [56]:
```
# 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 [37]:
```
# Importing XGBoost
from xgboost import XGBClassifier
```

Tuning the hyperparameters

In [65]:
```
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed: 12.6min finished
```

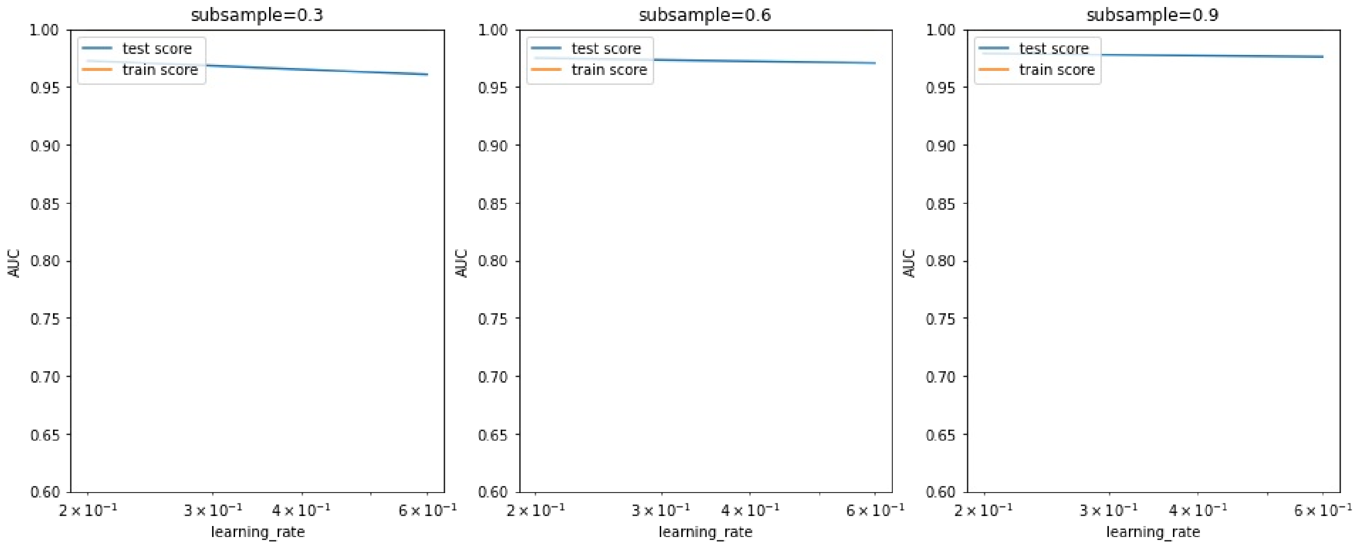
Out[65]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                     colsample_bylevel=1, colsample_bynode=1,
                                     colsample_bytree=1, gamma=0,
                                     learning_rate=0.1, max_delta_step=0,
                                     max_depth=2, min_child_weight=1,
                                     missing=None, n_estimators=200, n_jobs=1,
                                     nthread=None, objective='binary:logistic',
                                     random_state=0, reg_alpha=0, reg_lambda=1,
                                     scale_pos_weight=1, seed=None, silent=None,
                                     subsample=1, verbosity=1),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [66]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[66]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_subsample	params	split0_test
0	33.505086	0.727619	0.384090	0.004365	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.
1	44.019166	0.072776	0.384185	0.006928	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.
2	45.915397	0.132965	0.382851	0.006526	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.
3	32.986417	0.376595	0.399465	0.001861	0.6	0.3	{'learning_rate': 0.6, 'subsample': 0.3}	0.
4	42.858867	0.385860	0.394540	0.003410	0.6	0.6	{'learning_rate': 0.6, 'subsample': 0.6}	0.
5	45.059620	0.152377	0.397230	0.002187	0.6	0.9	{'learning_rate': 0.6, 'subsample': 0.9}	0.

In [67]:
```
# # 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 [68]:
```
model_cv.best_params_
```

Out[68]:
```
{'learning_rate': 0.2, 'subsample': 0.9}
```

In [38]:
```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': 0.2,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.9,
         'objective':'binary:logistic'}

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

Out[38]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic',
              params={'learning_rate': 0.2, 'max_depth': 2, 'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample': 0.9},
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method=None, validate_parameters=False,
              verbosity=None)
```

Prediction on the train set

In [39]:
```
# Predictions on the train set
y_train_pred = xgb_imb_model.predict(X_train)
```

In [40]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)
```

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

In [41]:
```
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 [42]:
```
# 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 [43]:
```
# classification_report
print(classification_report(y_train, y_train_pred))
```

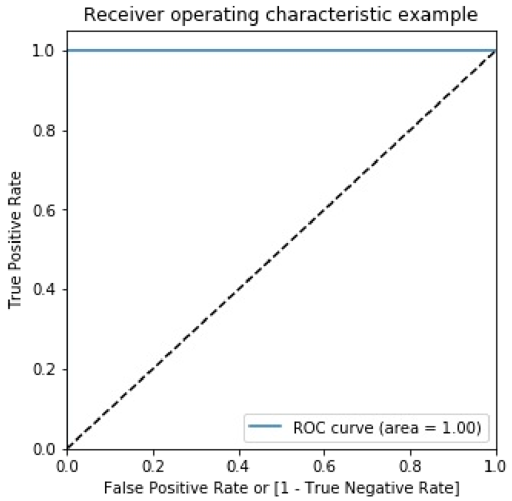
```
              precision    recall  f1-score   support

           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 [58]:
```
# Predicted probability
y_train_pred_proba_imb_xgb = xgb_imb_model.predict_proba(X_train)[:,1]
```

In [59]:
```
# roc_auc
auc = metrics.roc_auc_score(y_train, y_train_pred_proba_imb_xgb)
auc
```

Out[59]:
```
1.0
```

In [60]:
```
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba_imb_xgb)
```



Prediction on the test set

In [49]:
```
# Predictions on the test set
y_test_pred = xgb_imb_model.predict(X_test)
```

In [50]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56859     7]
 [   24    72]]
```

In [51]:
```
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 [52]:
```
# 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:- 0.9994557775359011
Sensitivity:- 0.75
Specificity:- 0.999876903597932
F1-Score:- 0.8228571428571428
```

In [53]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

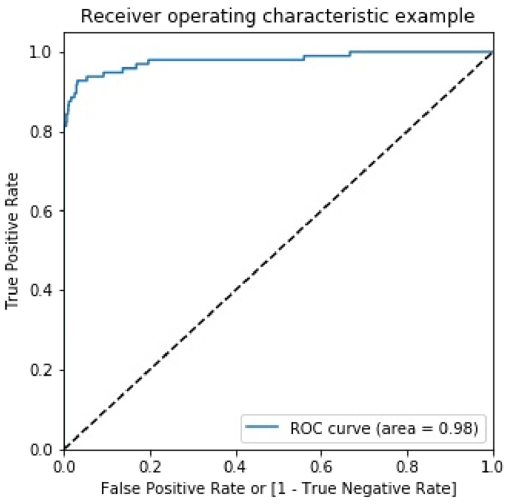
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.91      0.75      0.82        96

    accuracy                           1.00     56962
   macro avg       0.96      0.87      0.91     56962
weighted avg       1.00      1.00      1.00     56962
```

In [54]:
```
# Predicted probability
y_test_pred_proba = xgb_imb_model.predict_proba(X_test)[:,1]
```

In [55]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[55]:
```
0.9785370798602564
```

In [56]:
```
# 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.

Decision Tree

In [75]:
```
# Importing decision tree classifier
from sklearn.tree import DecisionTreeClassifier
```

In [83]:
```
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
}
```

```python
# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

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

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:  2.2min finished
```

Out[83]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(5, 15, 5),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=1)
```

In [84]:
```python
# cv results
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

Out[84]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_leaf	param_min_samples_
0	3.762192	0.022569	0.024710	0.000671	5	50	
1	3.764455	0.016738	0.024145	0.000872	5	50	
2	3.760637	0.012987	0.024381	0.000568	5	100	
3	3.750272	0.029414	0.024302	0.000159	5	100	
4	7.425092	0.014732	0.030241	0.003743	10	50	
5	7.398933	0.015277	0.025900	0.000441	10	50	
6	7.358769	0.028188	0.026375	0.000218	10	100	
7	7.382580	0.027872	0.026896	0.000646	10	100	

In [85]:
```python
# Printing the optimal sensitivity score and hyperparameters
print("Best roc_auc:-", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Best roc_auc:- 0.9382050164508641
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=5, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=100,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [76]:
```python
# Model with optimal hyperparameters
dt_imb_model = DecisionTreeClassifier(criterion = "gini",
                                  random_state = 100,
                                  max_depth=5,
                                  min_samples_leaf=100,
                                  min_samples_split=100)

dt_imb_model.fit(X_train, y_train)
```

Out[76]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=5, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=100,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=100, splitter='best')
```

Prediction on the train set

In [77]:
```python
# Predictions on the train set
y_train_pred = dt_imb_model.predict(X_train)
```

In [78]:
```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train)
print(confusion)
```

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

In [79]:
```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [80]:
```python
# 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:- 0.9991704887094297
Sensitivity:- 1.0
Specificity:- 1.0
F1-Score:- 0.7490039840637449
```

In [81]:
```python
# classification_report
print(classification_report(y_train, y_train_pred))
```

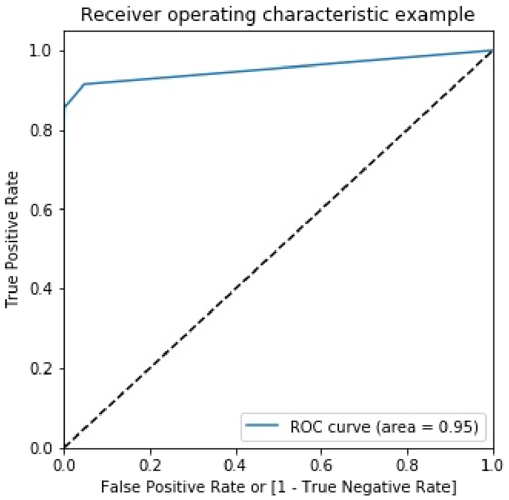
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       0.79      0.71      0.75       396

    accuracy                           1.00    227845
   macro avg       0.89      0.86      0.87    227845
weighted avg       1.00      1.00      1.00    227845
```

In [82]:
```python
# Predicted probability
y_train_pred_proba = dt_imb_model.predict_proba(X_train)[:,1]
```

In [83]:
```python
# roc_auc
auc = metrics.roc_auc_score(y_train, y_train_pred_proba)
auc
```

Out[83]:
```
0.9534547393930157
```

In [84]:
```python
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba)
```



Prediction on the test set

In [85]:
```
# Predictions on the test set
y_test_pred = dt_imb_model.predict(X_test)
```

In [86]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56836    30]
 [   40    56]]
```

In [87]:
```
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 [88]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

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

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

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

```
Accuracy:- 0.9987711105649381
Sensitivity:- 0.5833333333333334
Specificity:- 0.9994724439911371
F1-Score:- 0.7490039840637449
```

In [92]:
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

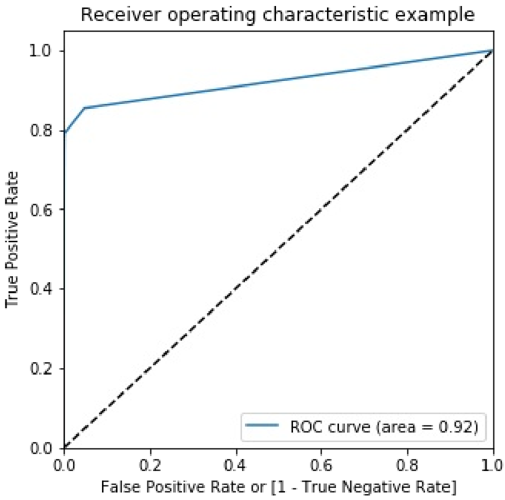
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.65      0.58      0.62        96

    accuracy                           1.00     56962
   macro avg       0.83      0.79      0.81     56962
weighted avg       1.00      1.00      1.00     56962
```

In [90]:
```
# Predicted probability
y_test_pred_proba = dt_imb_model.predict_proba(X_test)[:,1]
```

In [91]:
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[91]: 0.92174979703748

In [93]:
```
# 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
  - F1-Score = 0.75
  - ROC-AUC = 0.95
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.58
  - Specificity = 0.99
  - F-1 Score = 0.75
  - ROC-AUC = 0.92

Random forest

In [94]:
```
# Importing random forest classifier
from sklearn.ensemble import RandomForestClassifier
```

In [100…
```
param_grid = {
    'max_depth': range(5,10,5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'n_estimators': [100,200,300],
    'max_features': [10, 20]
}
# Create a based model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf,
                           param_grid = param_grid,
                           cv = 2,
                           n_jobs = -1,
                           verbose = 1,
                           return_train_score=True)

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

```
Fitting 2 folds for each of 24 candidates, totalling 48 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed: 101.0min finished
```

```
Out[100… GridSearchCV(cv=2, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,

                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=-1,
             param_grid={'max_depth': range(5, 10, 5), 'max_features': [10, 20],
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50),
                         'n_estimators': [100, 200, 300]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring=None, verbose=1)
```

In [101…
```
# printing the optimal accuracy score and hyperparameters
print('We can get accuracy of',grid_search.best_score_,'using',grid_search.best_params_)
```

```
We can get accuracy of 0.9992933790590904 using {'max_depth': 5, 'max_features': 10, 'min_samples_leaf': 50, 'mi
n_samples_split': 50, 'n_estimators': 100}
```

In [95]:
```
# model with the best hyperparameters

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

In [96]:
```
# Fit the model
rfc_imb_model.fit(X_train, y_train)
```

```
Out[96]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=5, max_features=10,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Prediction on the train set

In [97]:
```
# Predictions on the train set
y_train_pred = rfc_imb_model.predict(X_train)
```

In [98]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train)
print(confusion)
```

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

In [99]:
```
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 [100…
```
# 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:- 0.9993460466545239
Sensitivity:- 1.0
Specificity:- 1.0
F1-Score:- 0.7983761840324763
```

In [101…
```python
# classification_report
print(classification_report(y_train, y_train_pred))
```

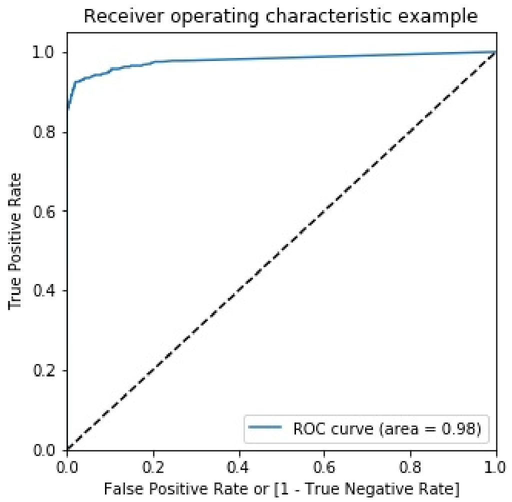
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227449
           1       0.86      0.74      0.80       396

    accuracy                           1.00    227845
   macro avg       0.93      0.87      0.90    227845
weighted avg       1.00      1.00      1.00    227845
```

In [102…
```python
# Predicted probability
y_train_pred_proba = rfc_imb_model.predict_proba(X_train)[:,1]
```

In [103…
```python
# roc_auc
auc = metrics.roc_auc_score(y_train, y_train_pred_proba)
auc
```

Out[103… 0.9791822295960585

In [104…
```python
# Plot the ROC curve
draw_roc(y_train, y_train_pred_proba)
```



Prediction on the test set

In [105…
```python
# Predictions on the test set
y_test_pred = rfc_imb_model.predict(X_test)
```

In [106…
```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56841    25]
 [   36    60]]
```

In [107…
```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [108…
```python
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

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

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

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

```
Accuracy:- 0.9989291106351603
Sensitivity:- 0.625
Specificity:- 0.9995603699926142
F1-Score:- 0.7983761840324763
```

In [109…

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

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.71      0.62      0.66        96

    accuracy                           1.00     56962
   macro avg       0.85      0.81      0.83     56962
weighted avg       1.00      1.00      1.00     56962
```

In [110…

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

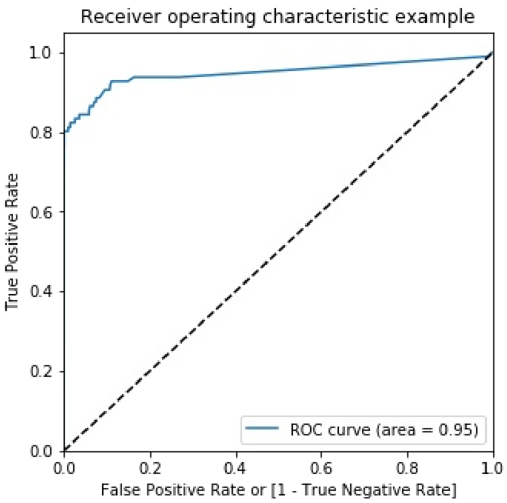
In [111…

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

Out[111… 0.9474696179029063

In [112…

```
# 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
  - F1-Score = 0.80
  - ROC-AUC = 0.98
- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.62
  - Specificity = 0.99
  - F-1 Score = 0.75
  - ROC-AUC = 0.96

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

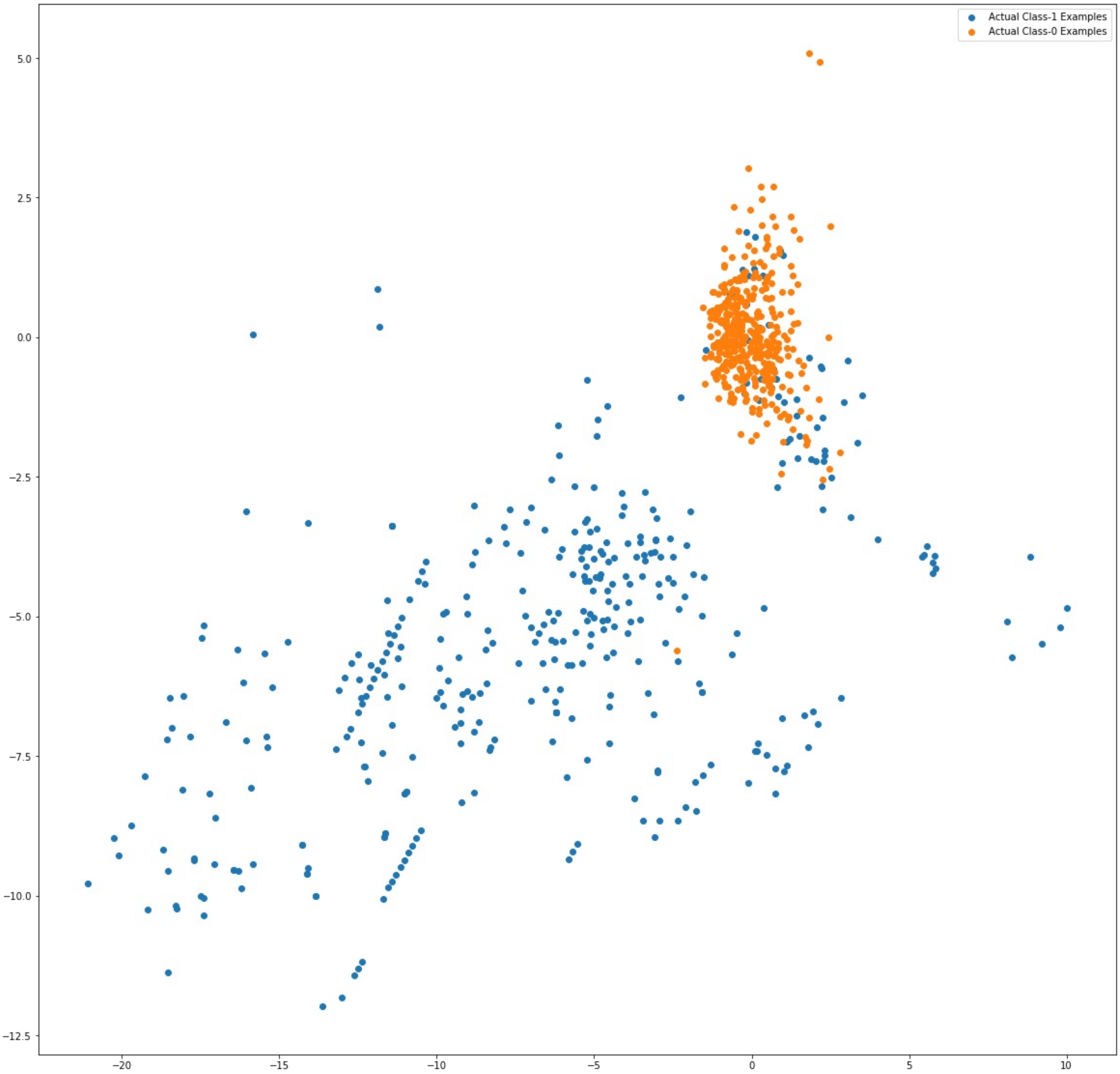
- This will not give much explanation on the already transformed dataset

- But it will help us in understanding if the dataset is not PCA transformed

In [57]:
```python
# Features of XGBoost model

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

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

np.random.shuffle(X_train_0)

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

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

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

Out[57]: <matplotlib.legend.Legend at 0x11887c88>



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

In [66]:
```
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= 0.8474788
```

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 [116…
```
# Importing undersampler library
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter
```

In [117…
```
# instantiating the random undersampler
rus = RandomUnderSampler()
# resampling X, y
X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)
```

In [118…
```
# 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 [50]:
```
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:    0.7s finished
```

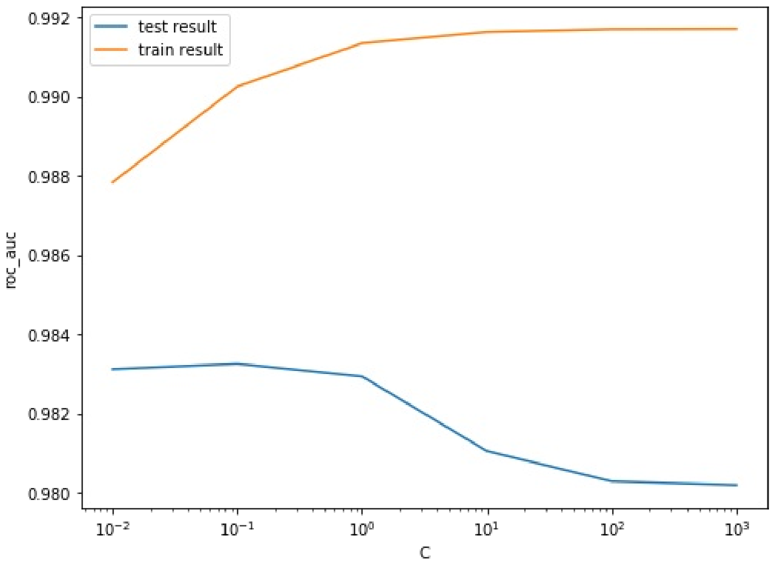
Out[50]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [51]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[51]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test
0	0.016201	0.009065	0.0040	1.095540e-03	0.01	{'C': 0.01}	0.983943	0.995410	0.9
1	0.016801	0.002136	0.0042	7.483665e-04	0.1	{'C': 0.1}	0.981240	0.995568	0.9
2	0.026201	0.004118	0.0040	1.095453e-03	1	{'C': 1}	0.981081	0.994302	0.9
3	0.020201	0.002786	0.0030	9.536743e-08	10	{'C': 10}	0.975199	0.994777	0.9
4	0.020801	0.002561	0.0030	6.324097e-04	100	{'C': 100}	0.972496	0.994619	0.9
5	0.021601	0.001497	0.0026	4.898624e-04	1000	{'C': 1000}	0.972178	0.994619	0.9

In [52]:
```
# 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 [53]:
```
# 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 0.9832637280039689 at C = 0.1
```

Logistic regression with optimal C

In [119…
```
# Instantiate the model with best C
logistic_bal_rus = LogisticRegression(C=0.1)
```

In [120…
```
# 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 [121…
```
# Predictions on the train set
y_train_pred = logistic_bal_rus_model.predict(X_train_rus)
```

In [122…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_pred)
print(confusion)
```

```
[[391   5]
 [ 32 364]]
```

In [123…
```
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 [124…
```
# 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:- 0.9532828282828283
Sensitivity:- 0.9191919191919192
Specificity:- 0.9873737373737373
F1-Score:- 0.9516339869281046
```

In [125…
```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

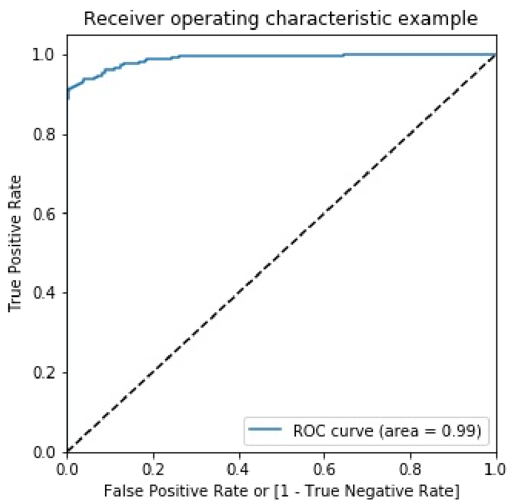
```
              precision    recall  f1-score   support

           0       0.92      0.99      0.95       396
           1       0.99      0.92      0.95       396

    accuracy                           0.95       792
   macro avg       0.96      0.95      0.95       792
weighted avg       0.96      0.95      0.95       792
```

In [126…
```
# Predicted probability
y_train_pred_proba = logistic_bal_rus_model.predict_proba(X_train_rus)[:,1]
```

In [127…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

Out[127… 0.9892230384654627

In [128…
```
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

In [129…
```
# Prediction on the test set
y_test_pred = logistic_bal_rus_model.predict(X_test)
```

In [130…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55658  1208]
 [   13    83]]
```

In [131…
```
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 [132…
```
# 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:- 0.9785646571398476
Sensitivity:- 0.8645833333333334
Specificity:- 0.978757078043119
```

In [133…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

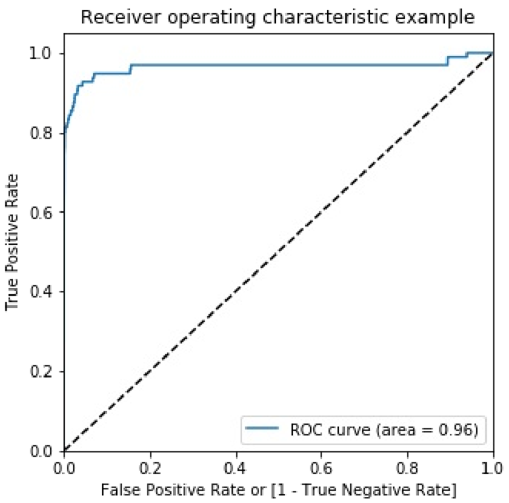
```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.06      0.86      0.12        96

    accuracy                           0.98     56962
   macro avg       0.53      0.92      0.55     56962
weighted avg       1.00      0.98      0.99     56962
```

In [134…
```
# Predicted probability
y_test_pred_proba = logistic_bal_rus_model.predict_proba(X_test)[:,1]
```

In [135…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[135… 0.9639748854031114

In [136…
```
# 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 [73]:
```
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:    3.9s finished
```

Out[73]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=2, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimato...
                                     objective='binary:logistic',
                                     random_state=None, reg_alpha=None,
                                     reg_lambda=None, scale_pos_weight=None,
                                     subsample=None, tree_method=None,
                                     validate_parameters=False,
                                     verbosity=None),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

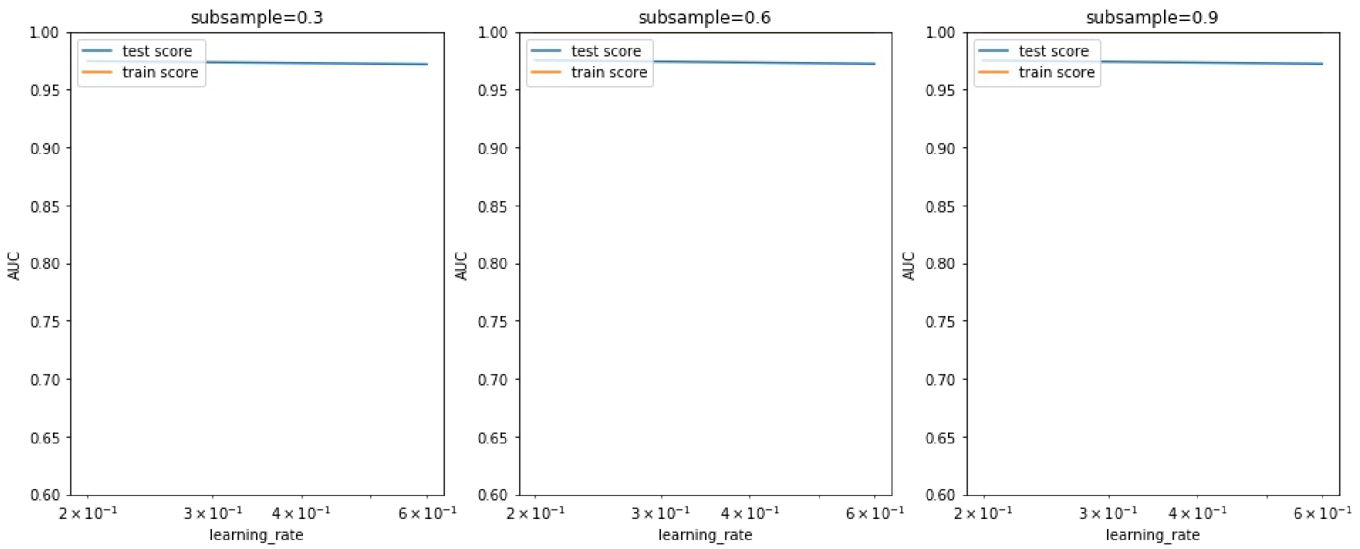
In [74]:
```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[74]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_subsample	params	split0_test
0	0.210345	0.069442	0.016668	0.014384	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.
1	0.168343	0.003300	0.006334	0.000471	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.
2	0.247348	0.025370	0.006334	0.000471	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.
3	0.247347	0.116300	0.011667	0.005437	0.6	0.3	{'learning_rate': 0.6, 'subsample': 0.3}	0.
4	0.188344	0.026248	0.007001	0.000817	0.6	0.6	{'learning_rate': 0.6, 'subsample': 0.6}	0.
5	0.171343	0.023115	0.006334	0.000471	0.6	0.9	{'learning_rate': 0.6, 'subsample': 0.9}	0.

In [75]:

```
# # 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 [76]:

```
model_cv.best_params_
```

Out[76]: {'learning_rate': 0.2, 'subsample': 0.6}

In [137…
```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': 0.2,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.6,
         'objective':'binary:logistic'}

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

Out[137…
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic',
              params={'learning_rate': 0.2, 'max_depth': 2, 'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample': 0.6},
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method=None, validate_parameters=False,
              verbosity=None)
```

Prediction on the train set

In [138…
```
# Predictions on the train set
y_train_pred = xgb_bal_rus_model.predict(X_train_rus)
```

In [139…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_rus)
print(confusion)
```

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

In [140…
```
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 [141…
```
# 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 [142…
```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

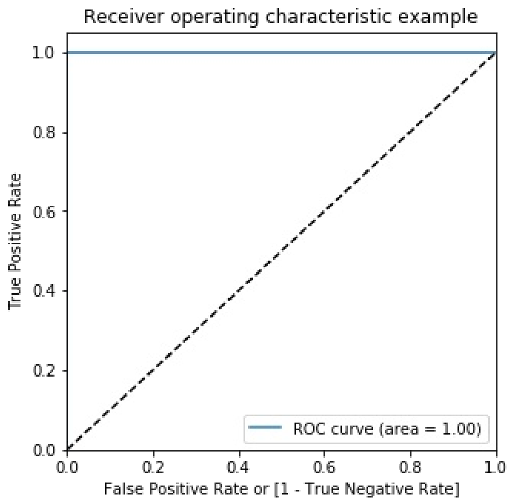
```
              precision    recall  f1-score   support

           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 [143…
```
# Predicted probability
y_train_pred_proba = xgb_bal_rus_model.predict_proba(X_train_rus)[:,1]
```

In [144…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

Out[144… 1.0

In [146…
```
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

In [147…
```
# Predictions on the test set
y_test_pred = xgb_bal_rus_model.predict(X_test)
```

In [148…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[54810  2056]
 [   11    85]]
```

In [149…
```
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 [150…
```
# 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:- 0.9637126505389558
Sensitivity:- 0.8854166666666666
Specificity:- 0.9638448281925931
```

In [151…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

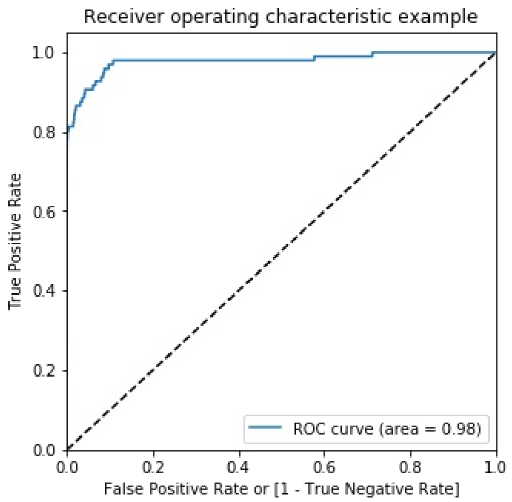
```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     56866
           1       0.04      0.89      0.08        96

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

In [152…
```
# Predicted probability
y_test_pred_proba = xgb_bal_rus_model.predict_proba(X_test)[:,1]
```

In [153…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[153… 0.9777381439114174

In [154…
```
# 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

Decision Tree

In [105…

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


# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

# Fit the grid search to the data
grid_search.fit(X_train_rus,y_train_rus)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:    0.2s finished
```

Out[105…

```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(5, 15, 5),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=1)
```

In [106]:
```
# cv results
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

Out[106]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_leaf	param_min_samples_
0	0.012001	1.632972e-03	0.004000	8.166321e-04	5	50	
1	0.009334	4.714266e-04	0.003000	1.123916e-07	5	50	
2	0.007334	4.712580e-04	0.004000	8.165347e-04	5	100	
3	0.007000	1.123916e-07	0.003334	4.714827e-04	5	100	
4	0.008667	4.715951e-04	0.003333	4.714266e-04	10	50	
5	0.013334	4.989110e-03	0.004000	8.165347e-04	10	50	
6	0.007334	1.247235e-03	0.004000	8.165347e-04	10	100	
7	0.007334	4.714827e-04	0.004000	1.414392e-03	10	100	

In [107]:
```
# Printing the optimal sensitivity score and hyperparameters
print("Best roc_auc:-", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Best roc_auc:- 0.9622073002754821
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=5, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [155]:
```
# Model with optimal hyperparameters
dt_bal_rus_model = DecisionTreeClassifier(criterion = "gini",
                                  random_state = 100,
                                  max_depth=5,
                                  min_samples_leaf=50,
                                  min_samples_split=50)

dt_bal_rus_model.fit(X_train_rus, y_train_rus)
```

Out[155]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=5, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=100, splitter='best')
```

Prediction on the train set

In [156]:
```
# Predictions on the train set
y_train_pred = dt_bal_rus_model.predict(X_train_rus)
```

In [157]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_rus, y_train_pred)
print(confusion)
```

```
[[391   5]
 [ 53 343]]
```

In [158]:
```
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 [159]:
```
# 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:- 0.9267676767676768
Sensitivity:- 0.8661616161616161
Specificity:- 0.9873737373737373
```

In [160…
```
# classification_report
print(classification_report(y_train_rus, y_train_pred))
```

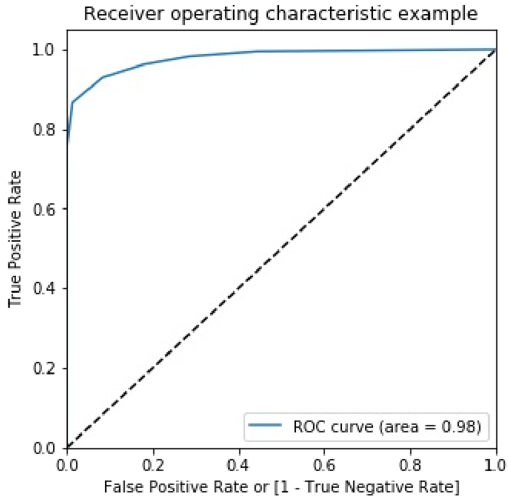
```
              precision    recall  f1-score   support

           0       0.88      0.99      0.93       396
           1       0.99      0.87      0.92       396

    accuracy                           0.93       792
   macro avg       0.93      0.93      0.93       792
weighted avg       0.93      0.93      0.93       792
```

In [161…
```
# Predicted probability
y_train_pred_proba = dt_bal_rus_model.predict_proba(X_train_rus)[:,1]
```

In [162…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_rus, y_train_pred_proba)
auc
```

Out[162… 0.9789944903581267

In [163…
```
# Plot the ROC curve
draw_roc(y_train_rus, y_train_pred_proba)
```



Prediction on the test set

In [164…
```
# Predictions on the test set
y_test_pred = dt_bal_rus_model.predict(X_test)
```

In [165…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55851  1015]
 [   19    77]]
```

In [166…
```
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 [167…
```
# 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:- 0.9818475474877989
Sensitivity:- 0.8020833333333334
Specificity:- 0.9821510217001371
```

In [168…

```
# 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      0.80      0.13        96

    accuracy                           0.98     56962
   macro avg       0.54      0.89      0.56     56962
weighted avg       1.00      0.98      0.99     56962
```

In [169…

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

In [170…

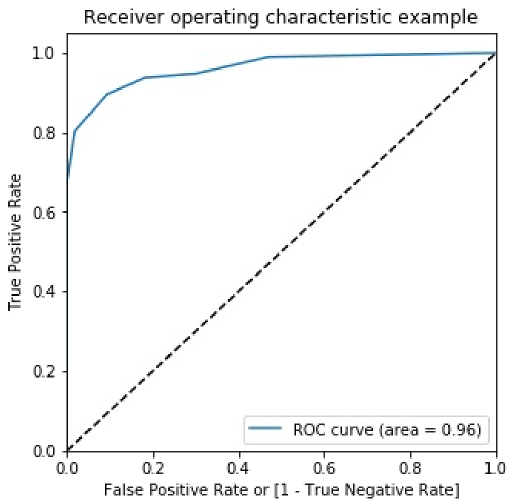
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[170…

```
0.9613739243719154
```

In [171…

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



Model summary

- Train set
  - Accuracy = 0.93
  - Sensitivity = 0.88
  - Specificity = 0.97
  - ROC-AUC = 0.98
- Test set
  - Accuracy = 0.96
  - Sensitivity = 0.85
  - Specificity = 0.96
  - ROC-AUC = 0.96

Random forest

In [123…

```
param_grid = {
    'max_depth': range(5,10,5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'n_estimators': [100,200,300],
    'max_features': [10, 20]
}
# Create a based model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf,
                           param_grid = param_grid,
                           scoring= 'roc_auc',
                           cv = 2,
                           n_jobs = -1,
```

```
                           verbose = 1,
                           return_train_score=True)

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

```
Fitting 2 folds for each of 24 candidates, totalling 48 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed:   11.4s finished
```

Out[123…

```
GridSearchCV(cv=2, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=-1,
             param_grid={'max_depth': range(5, 10, 5), 'max_features': [10, 20],
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50),
                         'n_estimators': [100, 200, 300]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [124…

```
# printing the optimal accuracy score and hyperparameters
print('We can get roc-auc of',grid_search.best_score_,'using',grid_search.best_params_)
```

```
We can get roc-auc of 0.976788082848689 using {'max_depth': 5, 'max_features': 10, 'min_samples_leaf': 50, 'min_
samples_split': 50, 'n_estimators': 200}
```

In [172…

```
# model with the best hyperparameters

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

In [173…

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

Out[173…

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=5, max_features=10,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Prediction on the train set

In [174…

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

In [175…

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

```
[[391   5]
 [ 44 352]]
```

In [176…

```
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 [177…

```
# 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:- 0.9381313131313131
Sensitivity:- 0.8888888888888888
Specificity:- 0.9873737373737373
F1-Score:- 0.9349269588313412
```

In [178…

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

```
              precision    recall  f1-score   support

           0       0.90      0.99      0.94       396
           1       0.99      0.89      0.93       396

    accuracy                           0.94       792
   macro avg       0.94      0.94      0.94       792
weighted avg       0.94      0.94      0.94       792
```

In [179…

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

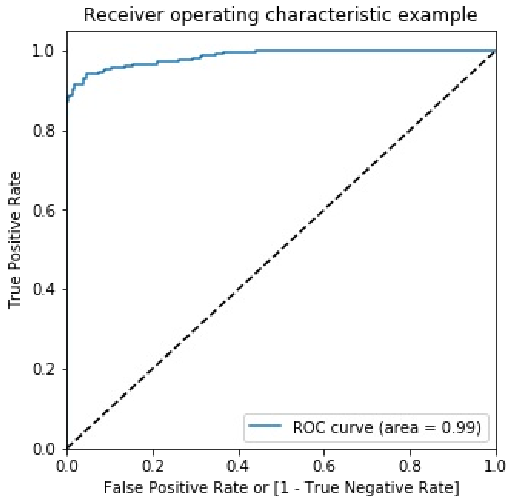
In [180…

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

Out[180… 0.9851099377614528

In [181…

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



Prediction on the test set

In [182…

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

In [183…

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

```
[[55832  1034]
 [   18    78]]
```

In [184…

```
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 [185…

```
# 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:- 0.9815315473473544
Sensitivity:- 0.8125
Specificity:- 0.981816902894524
```

In [186…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

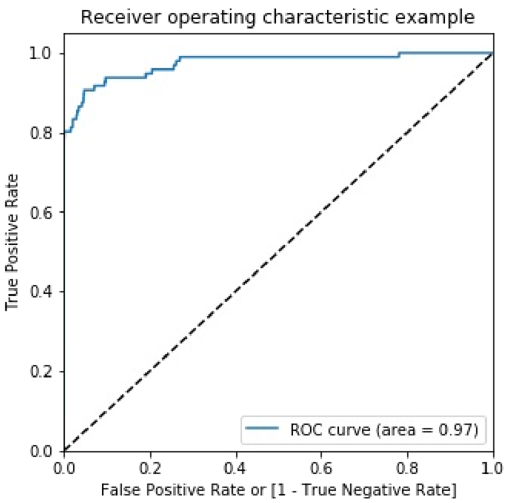
```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.07      0.81      0.13        96

    accuracy                           0.98     56962
   macro avg       0.53      0.90      0.56     56962
weighted avg       1.00      0.98      0.99     56962
```

In [187…
```
# Predicted probability
y_test_pred_proba = rfc_bal_rus_model.predict_proba(X_test)[:,1]
```

In [188…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[188… 0.9730361178032567

In [189…
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.94
  - Sensitivity = 0.89
  - Specificity = 0.98
  - ROC-AUC = 0.98
- Test set
  - Accuracy = 0.98
  - Sensitivity = 0.83
  - Specificity = 0.98
  - ROC-AUC = 0.97

Oversampling

In [190…
```
# Importing oversampler library
from imblearn.over_sampling import RandomOverSampler
```

In [191…
```
# instantiating the random oversampler
ros = RandomOverSampler()
# resampling X, y
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)
```

In [192…
```
# 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 [145…

```
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:  1.4min finished
```

Out[145…

```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [146…

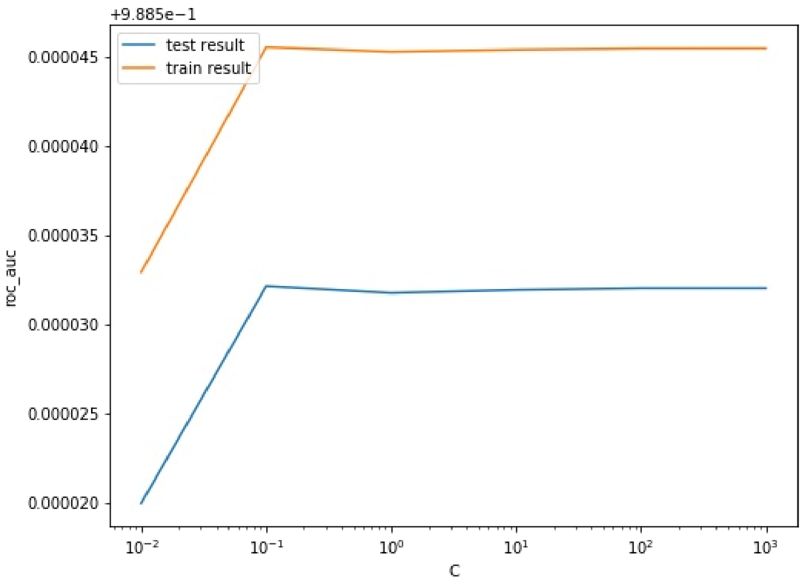
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[146…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test
0	2.392937	0.133817	0.052003	0.003847	0.01	{'C': 0.01}	0.988802	0.988039	0.9
1	2.366276	0.096595	0.048522	0.003303	0.1	{'C': 0.1}	0.988821	0.988048	0.9
2	2.725587	0.393503	0.056963	0.008990	1	{'C': 1}	0.988819	0.988049	0.9
3	2.949569	0.306817	0.061003	0.008391	10	{'C': 10}	0.988820	0.988049	0.9
4	2.584676	0.096526	0.056722	0.007632	100	{'C': 100}	0.988820	0.988050	0.9
5	2.384325	0.060643	0.050203	0.003371	1000	{'C': 1000}	0.988820	0.988050	0.9

In [147…

```
# 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 [148…
```
# 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 0.9885321193436821 at C = 0.1
```

Logistic regression with optimal C

In [193…
```
# Instantiate the model with best C
logistic_bal_ros = LogisticRegression(C=0.1)
```

In [194…
```
# 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 [195…
```
# Predictions on the train set
y_train_pred = logistic_bal_ros_model.predict(X_train_ros)
```

In [196…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_pred)
print(confusion)
```

```
[[222261   5188]
 [ 17649 209800]]
```

In [197…
```
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 [198…
```
# 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:- 0.9497975370302794
Sensitivity:- 0.9224045830054209
Specificity:- 0.9771904910551377
F1-Score:- 0.9483836116780467
```

In [199…
```
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

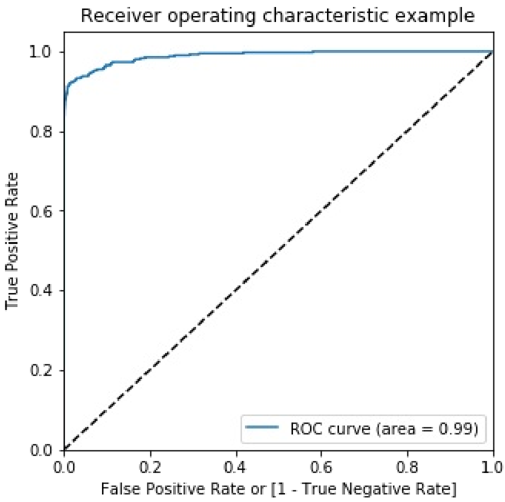
```
              precision    recall  f1-score   support

           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 [200…
```python
# Predicted probability
y_train_pred_proba = logistic_bal_ros_model.predict_proba(X_train_ros)[:,1]
```

In [201…
```python
# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

Out[201… 0.9886578544816166

In [202…
```python
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

In [203…
```python
# Prediction on the test set
y_test_pred = logistic_bal_ros_model.predict(X_test)
```

In [204…
```python
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[55540  1326]
 [   11    85]]
```

In [205…
```python
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [206…
```python
# 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:- 0.9765282117903163
Sensitivity:- 0.8854166666666666
Specificity:- 0.976682024408258
```

In [207…
```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56866
           1       0.06      0.89      0.11        96

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

In [208…

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

In [209…

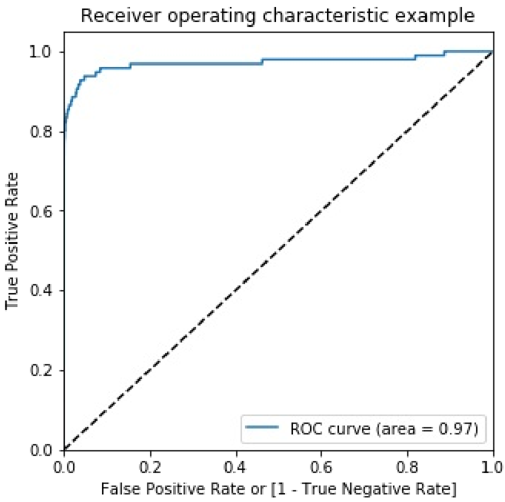
```python
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[209…

```
0.9712808034091842
```

In [210…

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



Model summary

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

XGBoost

In [222…

```python
# 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)
```

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

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed: 33.2min finished
```

Out[222…

```
GridSearchCV(cv=3, error_score=nan,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=2, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimato...
                                     objective='binary:logistic',
                                     random_state=None, reg_alpha=None,
                                     reg_lambda=None, scale_pos_weight=None,
                                     subsample=None, tree_method=None,
                                     validate_parameters=False,
                                     verbosity=None),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [164…

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

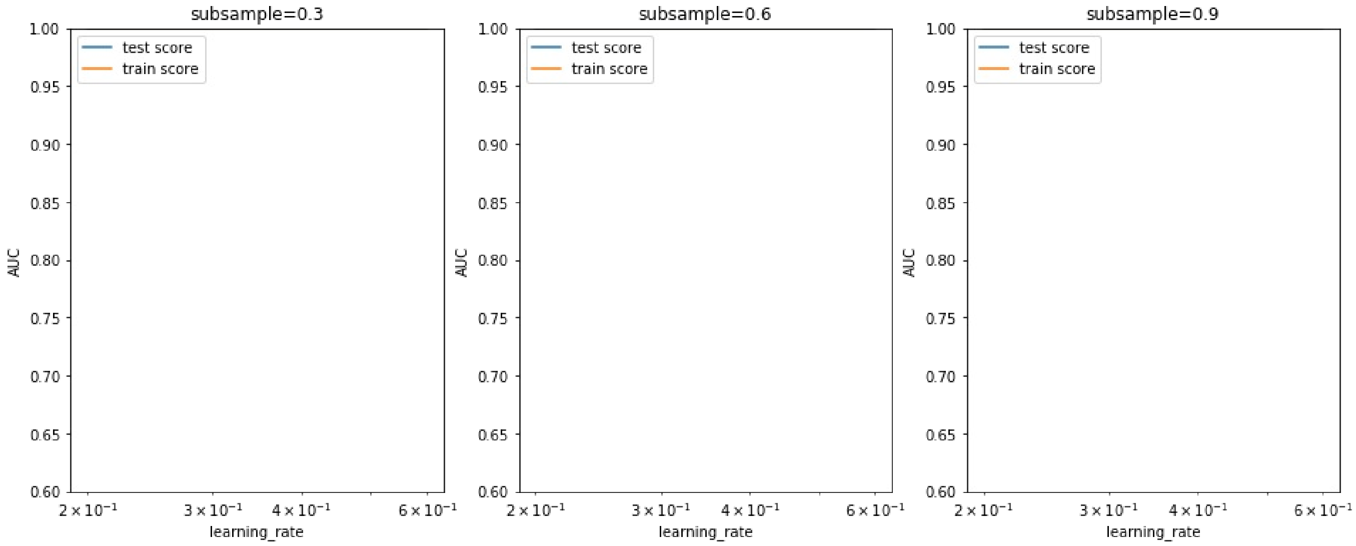
Out[164…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_subsample	params	split0_test
0	89.934024	4.977601	0.749709	0.031054	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.
1	117.291133	1.948062	0.781700	0.011541	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.
2	133.174869	3.055986	0.774044	0.020544	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.
3	108.884205	2.397979	0.861049	0.051926	0.6	0.3	{'learning_rate': 0.6, 'subsample': 0.3}	0.
4	126.067211	3.452522	0.857716	0.020887	0.6	0.6	{'learning_rate': 0.6, 'subsample': 0.6}	0.
5	134.505360	0.828143	0.842048	0.029339	0.6	0.9	{'learning_rate': 0.6, 'subsample': 0.9}	0.

In [165…

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

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


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


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

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

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



Model with optimal hyperparameters

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

In [166…

```
model_cv.best_params_
```

Out[166…

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

In [211…

```
# 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[211…

```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic',
              params={'learning_rate': 0.6, 'max_depth': 2, 'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample': 0.9},
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method=None, validate_parameters=False,
              verbosity=None)
```

Prediction on the train set

In [212…

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

In [213…

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

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

In [214…

```
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 [215…

```
# 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 [216…
```
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

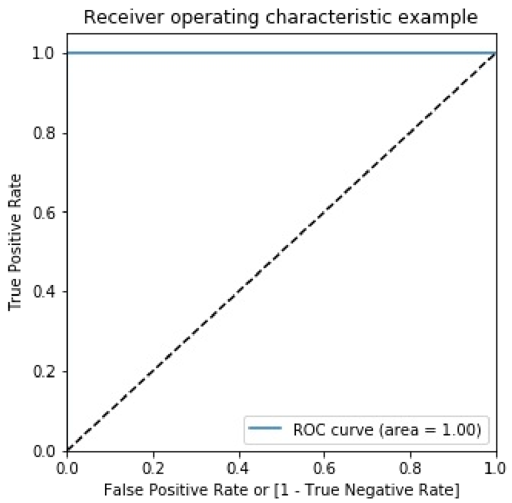
```
              precision    recall  f1-score   support

           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 [217…
```
# Predicted probability
y_train_pred_proba = xgb_bal_ros_model.predict_proba(X_train_ros)[:,1]
```

In [218…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

Out[218… 1.0

In [219…
```
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

In [223…
```
# Predictions on the test set
y_test_pred = xgb_bal_ros_model.predict(X_test)
```

In [224…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56857     9]
 [   19    77]]
```

In [225…
```
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 [226…
```
# 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:- 0.9995084442259752
Sensitivity:- 0.8020833333333334
Specificity:- 0.9998417331973412
```

In [227…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

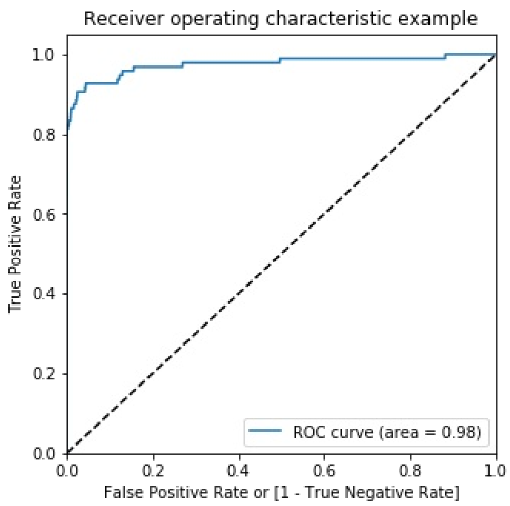
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.90      0.80      0.85        96

    accuracy                           1.00     56962
   macro avg       0.95      0.90      0.92     56962
weighted avg       1.00      1.00      1.00     56962
```

In [228…
```
# Predicted probability
y_test_pred_proba = xgb_bal_ros_model.predict_proba(X_test)[:,1]
```

In [229…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[229…
```
0.9751521119825555
```

In [230…
```
# 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

Decision Tree

In [180…
```
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
}


# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

# Fit the grid search to the data
grid_search.fit(X_train_ros,y_train_ros)
```

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

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:  3.2min finished
```

Out[180…

```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(5, 15, 5),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=1)
```

In [181…

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

Out[181…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_leaf	param_min_samples_
0	6.194021	0.117651	0.089005	1.123916e-07	5	50	
1	6.149352	0.019800	0.095672	9.428756e-03	5	50	
2	6.112016	0.013696	0.089672	4.715390e-04	5	100	
3	6.115016	0.025955	0.089672	4.715951e-04	5	100	
4	9.501210	0.124013	0.093672	1.247235e-03	10	50	
5	9.553962	0.181357	0.093402	2.803609e-04	10	50	
6	9.538348	0.164482	0.096734	4.431263e-03	10	100	
7	9.481282	0.091798	0.092400	1.697113e-03	10	100	

In [182…

```
# Printing the optimal sensitivity score and hyperparameters
print("Best roc_auc:-", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Best roc_auc:- 0.9996033327168891
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [231…

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

dt_bal_ros_model.fit(X_train_ros, y_train_ros)
```

Out[231… DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=100, splitter='best')

Prediction on the train set

In [232…
```
# Predictions on the train set
y_train_pred = dt_bal_ros_model.predict(X_train_ros)
```

In [233…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_ros, y_train_pred)
print(confusion)
```

```
[[225914   1535]
 [     0 227449]]
```

In [234…
```
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 [235…
```
# 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:- 0.9966256171713219
Sensitivity:- 1.0
Specificity:- 0.9932512343426438
```

In [236…
```
# classification_report
print(classification_report(y_train_ros, y_train_pred))
```

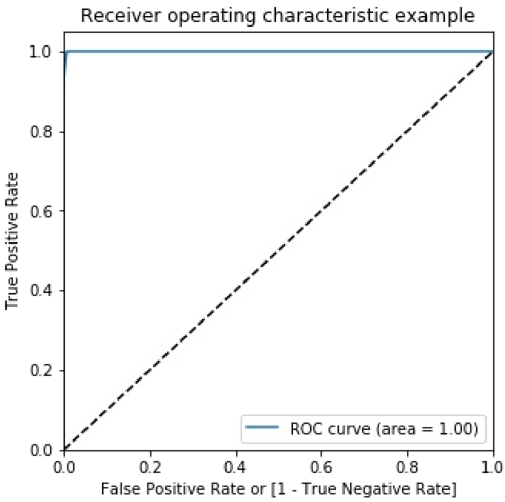
```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00    227449
           1       0.99      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 [237…
```
# Predicted probability
y_train_pred_proba = dt_bal_ros_model.predict_proba(X_train_ros)[:,1]
```

In [238…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_ros, y_train_pred_proba)
auc
```

Out[238… 0.9997642505020377

In [239…
```
# Plot the ROC curve
draw_roc(y_train_ros, y_train_pred_proba)
```



Prediction on the test set

In [240…
```
# Predictions on the test set
y_test_pred = dt_bal_ros_model.predict(X_test)
```

In [241…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56431   435]
 [   20    76]]
```

In [242…
```
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 [243…
```
# 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:- 0.9920122186720972
Sensitivity:- 0.7916666666666666
Specificity:- 0.9923504378714874
```

In [244…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

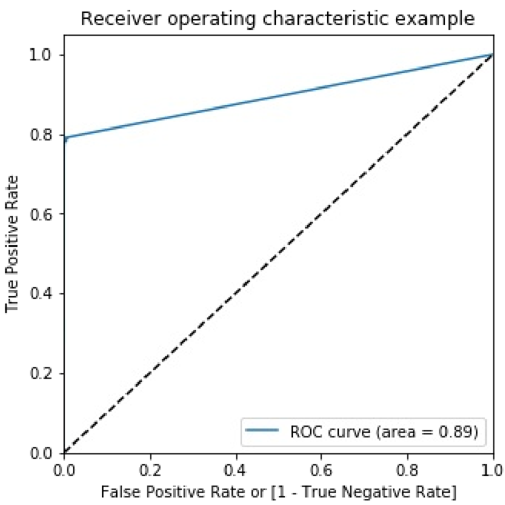
```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     56866
           1       0.15      0.79      0.25        96

    accuracy                           0.99     56962
   macro avg       0.57      0.89      0.62     56962
weighted avg       1.00      0.99      0.99     56962
```

In [245…
```
# Predicted probability
y_test_pred_proba = dt_bal_ros_model.predict_proba(X_test)[:,1]
```

In [246…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[246… 0.8948251151830618

In [247…
```
# 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 = 0.99

- Test set
  - Accuracy = 0.99
  - Sensitivity = 0.79
  - Specificity = 0.99
  - ROC-AUC = 0.90

SMOTE (Synthetic Minority Oversampling Technique)

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

In [68]:
```
# Importing SMOTE
from imblearn.over_sampling import SMOTE
```

In [69]:
```
# Instantiate SMOTE
sm = SMOTE(random_state=27)
# Fitting SMOTE to the train set
X_train_smote, y_train_smote = sm.fit_sample(X_train, y_train)
```

In [70]:
```
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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:  1.5min finished
```

Out[ ]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

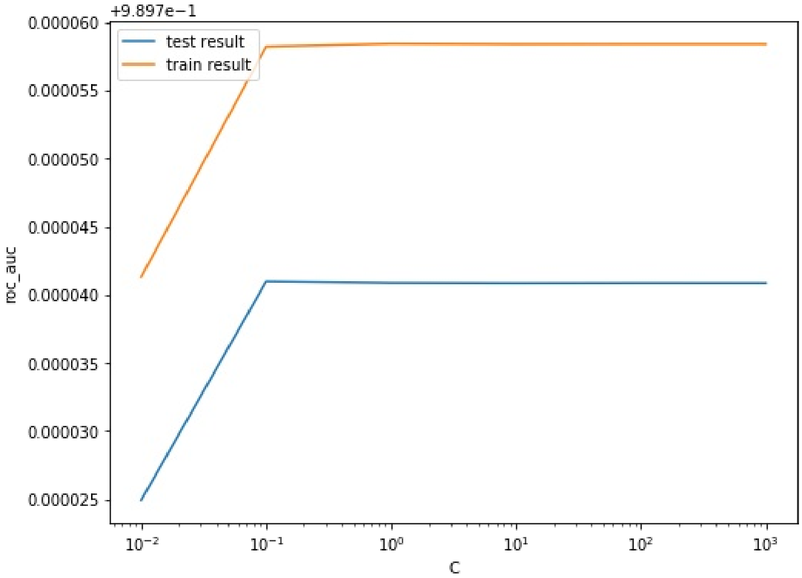
In [ ]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test
0	2.517373	0.063613	0.068640	0.012480	0.01	{'C': 0.01}	0.989805	0.989796	0.9
1	2.580687	0.139180	0.065641	0.006184	0.1	{'C': 0.1}	0.989834	0.989807	0.9
2	2.673410	0.107579	0.065920	0.006089	1	{'C': 1}	0.989836	0.989807	0.9
3	2.650617	0.100909	0.065520	0.006240	10	{'C': 10}	0.989836	0.989807	0.9
4	2.693168	0.148317	0.065520	0.006240	100	{'C': 100}	0.989836	0.989807	0.9
5	2.745892	0.157325	0.056160	0.007642	1000	{'C': 1000}	0.989836	0.989807	0.9

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 [71]:

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

In [72]:

```
# 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 [73]:

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

In [74]:

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

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

In [75]:

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

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

In [76]:
```
# 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))
```

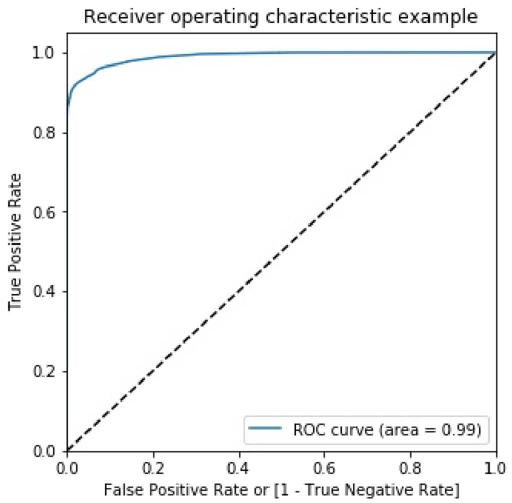
```
Accuracy:- 0.9489314087993352
Sensitivity:- 0.9222111330452102
Specificity:- 0.9756516845534603
```

In [77]:
```
# 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 [89]:
```
# Predicted probability
y_train_pred_proba_log_bal_smote = logistic_bal_smote_model.predict_proba(X_train_smote)[:,1]
```

In [90]:
```
# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba_log_bal_smote)
```



Prediction on the test set

In [80]:
```
# Prediction on the test set
y_test_pred = logistic_bal_smote_model.predict(X_test)
```

In [81]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

In [82]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [83]:
```
# 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
```

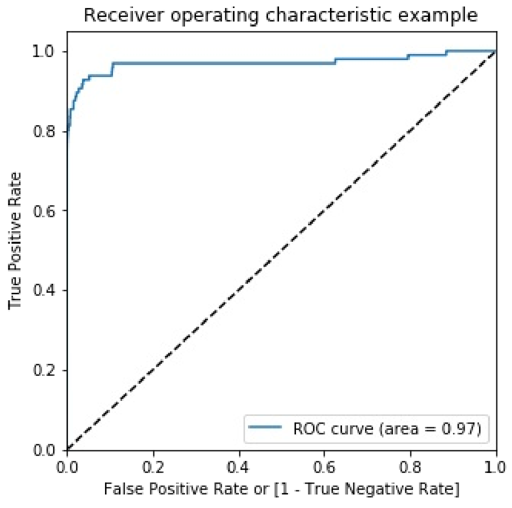
In [84]:
```python
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     56866
           1       0.06      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 [85]:
```python
# Predicted probability
y_test_pred_proba = logistic_bal_smote_model.predict_proba(X_test)[:,1]
```

In [86]:
```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

XGBoost

In [ ]:
```python
# 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
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed: 45.8min finished
```

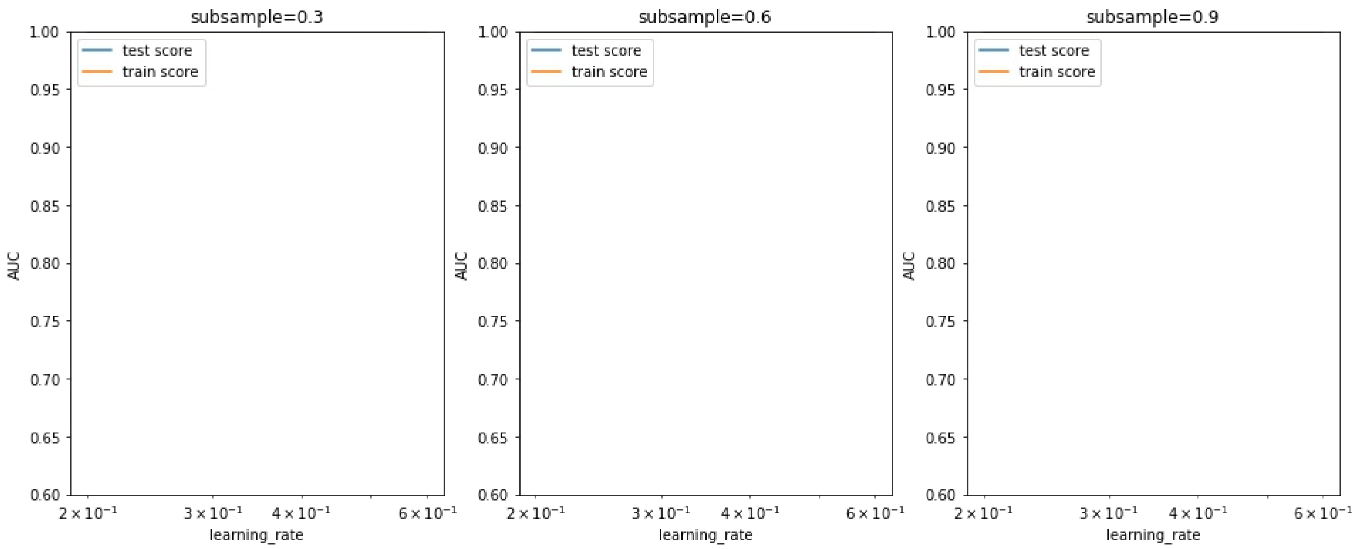
Out[ ]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=2, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimato...
                                     objective='binary:logistic',
                                     random_state=None, reg_alpha=None,
                                     reg_lambda=None, scale_pos_weight=None,
                                     subsample=None, tree_method=None,
                                     validate_parameters=False,
                                     verbosity=None),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

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	param_subsample	params	split0_test
0	120.679338	2.195478	0.847048	0.012833	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.
1	154.374830	1.994512	0.847715	0.007040	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.
2	173.991618	0.672700	0.816047	0.020834	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.
3	122.247942	0.657327	0.870037	0.022553	0.6	0.3	{'learning_rate': 0.6, 'subsample': 0.3}	0.
4	153.590355	1.449181	0.857216	0.032023	0.6	0.6	{'learning_rate': 0.6, 'subsample': 0.6}	0.
5	175.839443	2.079717	0.829144	0.007553	0.6	0.9	{'learning_rate': 0.6, 'subsample': 0.9}	0.

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': 0.9}
```

In [267…
```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we need for calculating auc
params = {'learning_rate': 0.6,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.9,
         'objective':'binary:logistic'}

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

Out[267…
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic',
              params={'learning_rate': 0.6, 'max_depth': 2, 'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample': 0.9},
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method=None, validate_parameters=False,
              verbosity=None)
```

Prediction on the train set

In [268…
```
# Predictions on the train set
y_train_pred = xgb_bal_smote_model.predict(X_train_smote)
```

In [269…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
print(confusion)
```

```
[[227447      2]
 [     0 227449]]
```

In [270…
```
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 [271…
```
# 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:- 0.9999956034099952
Sensitivity:- 1.0
Specificity:- 0.9999912068199904
```

In [272…

```
# 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 [273…

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

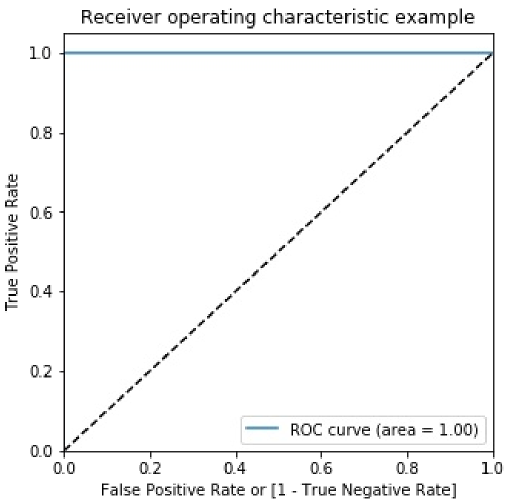
In [274…

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

Out[274… 1.0

In [275…

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



Prediction on the test set

In [276…

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

In [277…

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

```
[[56839    27]
 [   20    76]]
```

In [278…

```
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 [279…

```
# 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:- 0.9991748885221726
Sensitivity:- 0.7916666666666666
Specificity:- 0.9995251995920234
```

In [280…

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

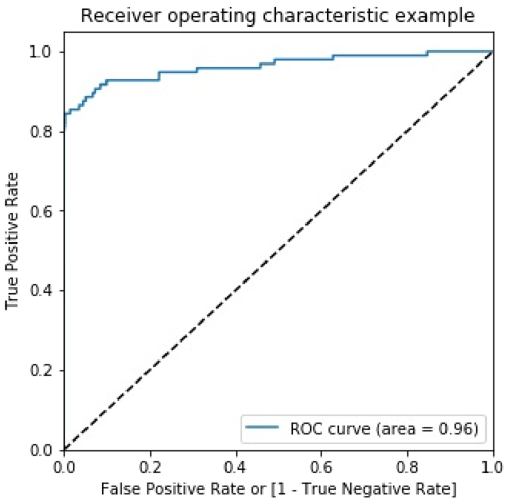
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.74      0.79      0.76        96

    accuracy                           1.00     56962
   macro avg       0.87      0.90      0.88     56962
weighted avg       1.00      1.00      1.00     56962
```

In [281…
```python
# Predicted probability
y_test_pred_proba = xgb_bal_smote_model.predict_proba(X_test)[:,1]
```

In [282…
```python
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[282… 0.9618437789423088

In [283…
```python
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

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

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

Decision Tree

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


# Instantiate the grid search model
dtree = DecisionTreeClassifier()

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

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

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:  5.9min finished
```

Out[ ]:

```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(5, 15, 5),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=1)
```

In [ ]:

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

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_leaf	param_min_samples_
0	9.971941	0.245916	0.091339	0.001700	5	50	
1	9.798227	0.043148	0.090672	0.002495	5	50	
2	9.804227	0.061308	0.090672	0.001247	5	100	
3	10.006914	0.237793	0.093204	0.000280	5	100	
4	22.208819	2.856627	0.096871	0.002617	10	50	
5	19.618377	0.749607	0.102538	0.008521	10	50	
6	18.075125	0.167592	0.096537	0.002231	10	100	
7	18.079367	0.254541	0.103004	0.002159	10	100	

In [ ]:

```
# Printing the optimal sensitivity score and hyperparameters
print("Best roc_auc:-", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Best roc_auc:- 0.9980773622123168
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=100,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [284…

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

dt_bal_smote_model.fit(X_train_smote, y_train_smote)
```

Out[284… DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=50, min_samples_split=100,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=100, splitter='best')

Prediction on the train set

In [285…
```
# Predictions on the train set
y_train_pred = dt_bal_smote_model.predict(X_train_smote)
```

In [286…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_smote, y_train_pred)
print(confusion)
```

```
[[223809   3640]
 [  2374 225075]]
```

In [287…
```
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 [288…
```
# 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:- 0.9867794538555896
Sensitivity:- 0.9895624953286232
Specificity:- 0.9839964123825561
```

In [290…
```
# classification_report
print(classification_report(y_train_smote, y_train_pred))
```

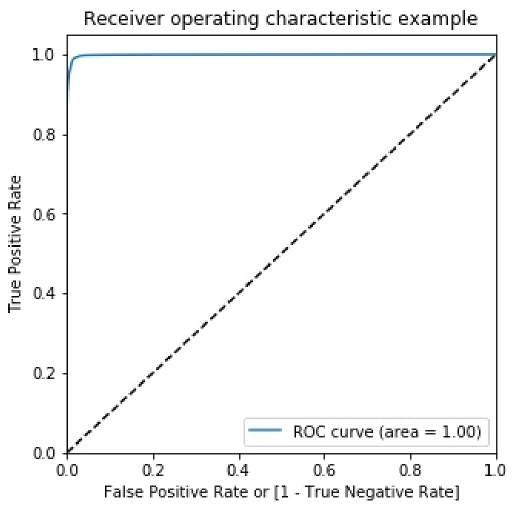
```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99    227449
           1       0.98      0.99      0.99    227449

    accuracy                           0.99    454898
   macro avg       0.99      0.99      0.99    454898
weighted avg       0.99      0.99      0.99    454898
```

In [291…
```
# Predicted probability
y_train_pred_proba = dt_bal_smote_model.predict_proba(X_train_smote)[:,1]
```

In [292…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_smote, y_train_pred_proba)
auc
```

Out[292… 0.9986355757920081

In [294…
```
# Plot the ROC curve
draw_roc(y_train_smote, y_train_pred_proba)
```



Prediction on the test set

In [295…

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

In [296…

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

```
[[55852  1014]
 [   19    77]]
```

In [297…

```
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 [298…

```
# 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:- 0.9818651030511569
Sensitivity:- 0.8020833333333334
Specificity:- 0.9821686069004326
```

In [299…

```
# 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      0.80      0.13        96

    accuracy                           0.98     56962
   macro avg       0.54      0.89      0.56     56962
weighted avg       1.00      0.98      0.99     56962
```

In [300…

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

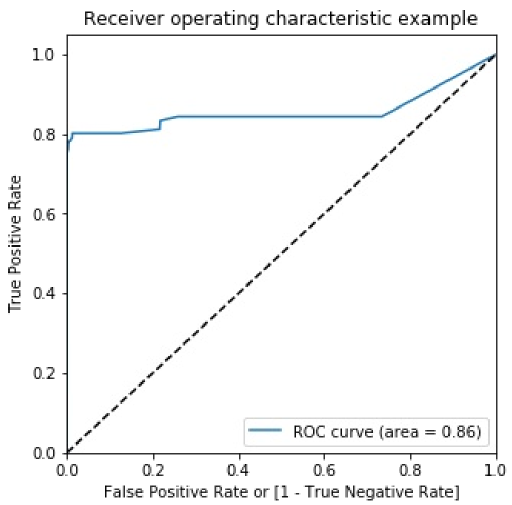
In [301…

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

Out[301… 0.8551876157692353

In [302…

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



Model summary

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

- Test set
  - Accuracy = 0.98
  - Sensitivity = 0.80
  - Specificity = 0.98
  - ROC-AUC = 0.86

AdaSyn (Adaptive Synthetic Sampling)

In [303…
```python
# Importing adasyn
from imblearn.over_sampling import ADASYN
```

In [304…
```python
# Instantiate adasyn
ada = ADASYN(random_state=0)
X_train_adasyn, y_train_adasyn = ada.fit_resample(X_train, y_train)
```

In [305…
```python
# 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 [238…
```python
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:   42.9s finished
```

Out[238…
```
GridSearchCV(cv=KFold(n_splits=3, random_state=4, shuffle=True),
             error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

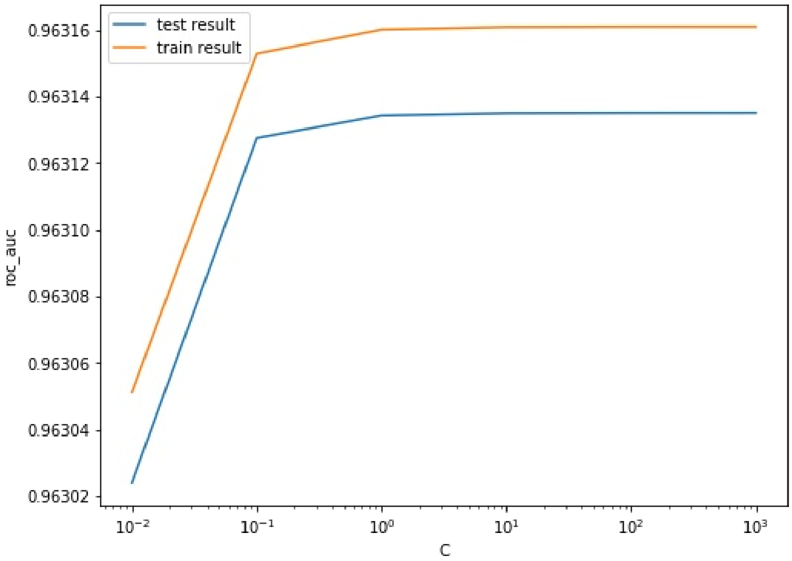
In [239…
```python
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[239…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test_
0	1.919777	0.172030	0.106673	0.021640	0.01	{'C': 0.01}	0.963472	0.962327	0.9
1	1.930110	0.079116	0.090672	0.002495	0.1	{'C': 0.1}	0.963578	0.962435	0.9
2	2.188786	0.195766	0.093672	0.004497	1	{'C': 1}	0.963585	0.962442	0.9
3	2.356865	0.238141	0.103206	0.009386	10	{'C': 10}	0.963585	0.962443	0.9
4	2.035450	0.107419	0.091672	0.002357	100	{'C': 100}	0.963585	0.962443	0.9
5	2.046450	0.091060	0.094005	0.007119	1000	{'C': 1000}	0.963585	0.962443	0.9

In [240…

```
# 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 [241…

```
# 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 0.9631351482818916 at C = 1000
```

Logistic regression with optimal C

In [306…

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

In [307…

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

Prediction on the train set

In [308…

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

In [309…

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

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

In [310…

```
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 [311…
```
# 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 [312…
```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

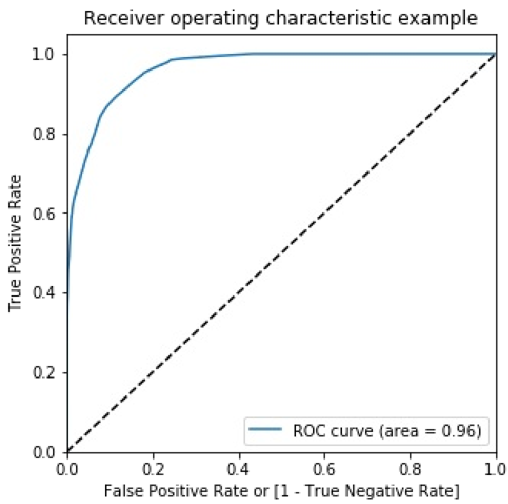
```
              precision    recall  f1-score   support

           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 [313…
```
# Predicted probability
y_train_pred_proba = logistic_bal_adasyn_model.predict_proba(X_train_adasyn)[:,1]
```

In [314…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

Out[314… 0.9631610161614914

In [315…
```
# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

In [316…
```
# Prediction on the test set
y_test_pred = logistic_bal_adasyn_model.predict(X_test)
```

In [317…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

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

In [318…
```
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 [319…
```
# 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 [320…

```
# 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 [321…

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

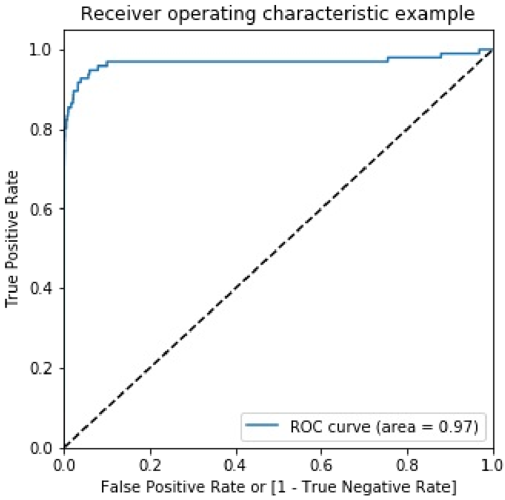
In [322…

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

Out[322… 0.9671573487086602

In [323…

```
# 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

Decision Tree

In [205…

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


# Instantiate the grid search model
```

```python
dtree = DecisionTreeClassifier()

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

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

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:  5.4min finished
```

Out[205…

```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(5, 15, 5),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=1)
```

In [206…

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

Out[206…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_leaf	param_min_samples_
0	9.992159	0.352046	0.098602	0.007498	5	50	
1	9.723238	0.012229	0.092537	0.001110	5	50	
2	9.719180	0.064421	0.087535	0.006825	5	100	
3	9.705453	0.014290	0.092735	0.001223	5	100	
4	17.367458	0.262487	0.104000	0.014708	10	50	
5	17.314552	0.315418	0.094069	0.000663	10	50	
6	17.106967	0.206823	0.104667	0.014260	10	100	
7	17.102270	0.148033	0.099734	0.006711	10	100	

In [207…

```python
# Printing the optimal sensitivity score and hyperparameters
print("Best roc_auc:-", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Best roc_auc:- 0.9414793563319087
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [324…

```python
# Model with optimal hyperparameters
dt_bal_adasyn_model = DecisionTreeClassifier(criterion = "gini",
```

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

dt_bal_adasyn_model.fit(X_train_adasyn, y_train_adasyn)
```

Out[324…

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=10, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=50,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=100, splitter='best')
```

Prediction on the train set

In [325…

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

In [326…

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

```
[[215929  11520]
 [  1118 226330]]
```

In [327…

```
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 [328…

```
# 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:- 0.9722178866864367
Sensitivity:- 0.9950845907636031
Specificity:- 0.9493512831447929
```

In [329…

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

```
              precision    recall  f1-score   support

           0       0.99      0.95      0.97    227449
           1       0.95      1.00      0.97    227448

    accuracy                           0.97    454897
   macro avg       0.97      0.97      0.97    454897
weighted avg       0.97      0.97      0.97    454897
```

In [330…

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

In [331…

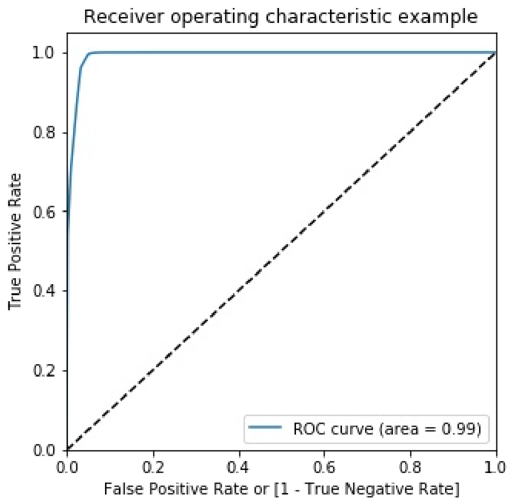
```
# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

Out[331…

```
0.9917591040224101
```

In [332…

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



Prediction on the test set

In [333…
```
# Predictions on the test set
y_test_pred = dt_bal_adasyn_model.predict(X_test)
```

In [334…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[53880  2986]
 [   15    81]]
```

In [335…
```
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 [336…
```
# 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:- 0.9473157543625575
Sensitivity:- 0.84375
Specificity:- 0.9474905919178419
```

In [337…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

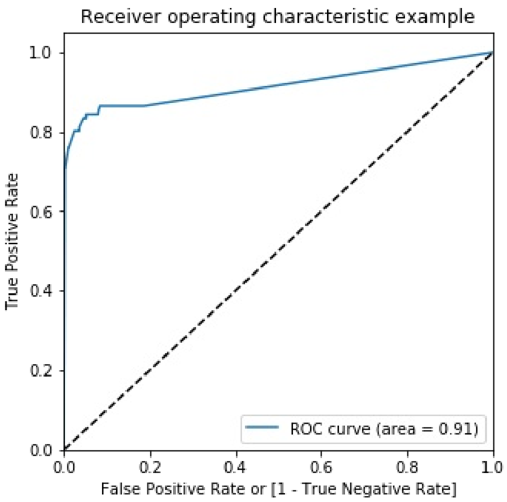
```
              precision    recall  f1-score   support

           0       1.00      0.95      0.97     56866
           1       0.03      0.84      0.05        96

    accuracy                           0.95     56962
   macro avg       0.51      0.90      0.51     56962
weighted avg       1.00      0.95      0.97     56962
```

In [338…
```
# Predicted probability
y_test_pred_proba = dt_bal_adasyn_model.predict_proba(X_test)[:,1]
```

In [339…
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[339… 0.9141440147305362

In [340…
```
# Plot the ROC curve
draw_roc(y_test, y_test_pred_proba)
```



Model summary

- Train set
  - Accuracy = 0.97
  - Sensitivity = 0.99
  - Specificity = 0.95
  - ROC-AUC = 0.99
- Test set
  - Accuracy = 0.95
  - Sensitivity = 0.84
  - Specificity = 0.95
  - ROC-AUC = 0.91

XGBoost

In [221…

```
# 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
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed: 42.5min finished
```

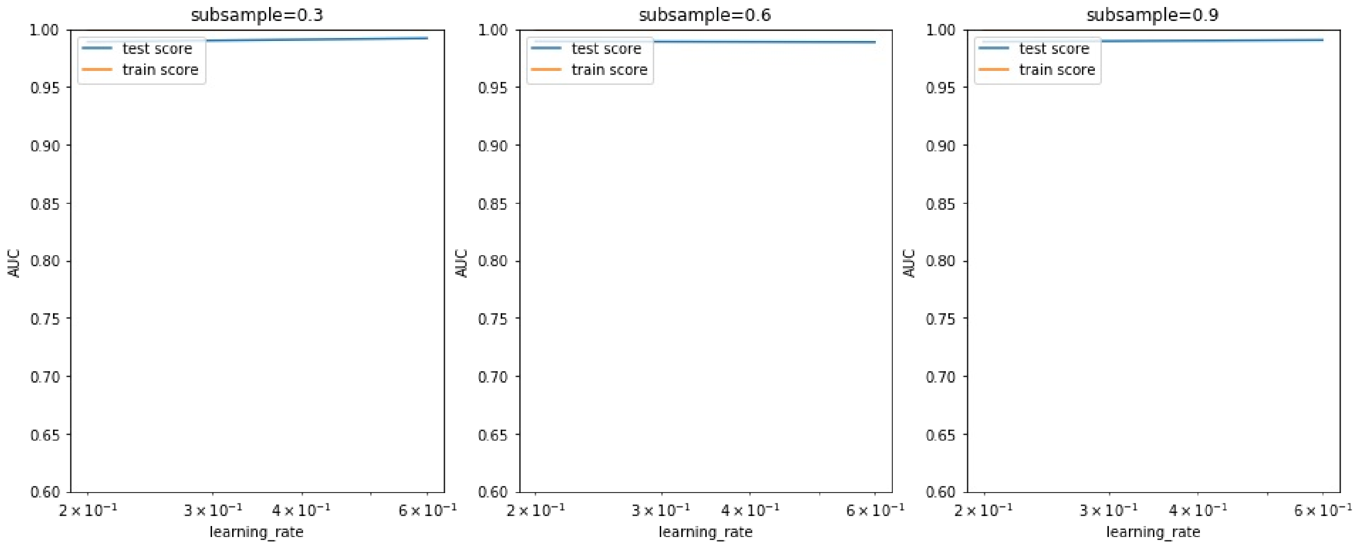
Out[221…
```
GridSearchCV(cv=3, error_score=nan,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=2, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimato...
                                     objective='binary:logistic',
                                     random_state=None, reg_alpha=None,
                                     reg_lambda=None, scale_pos_weight=None,
                                     subsample=None, tree_method=None,
                                     validate_parameters=False,

                                     verbosity=None),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': [0.2, 0.6],
                         'subsample': [0.3, 0.6, 0.9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

In [222…
```python
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[222…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_learning_rate	param_subsample	params	split0_test
0	107.725133	10.671068	0.794354	0.057250	0.2	0.3	{'learning_rate': 0.2, 'subsample': 0.3}	0.
1	138.776001	0.322162	0.785001	0.008165	0.2	0.6	{'learning_rate': 0.2, 'subsample': 0.6}	0.
2	157.356024	1.177755	0.809046	0.050527	0.2	0.9	{'learning_rate': 0.2, 'subsample': 0.9}	0.
3	108.153853	0.945556	0.795379	0.047079	0.6	0.3	{'learning_rate': 0.6, 'subsample': 0.3}	0.
4	139.687656	0.522447	0.793045	0.024834	0.6	0.6	{'learning_rate': 0.6, 'subsample': 0.6}	0.
5	184.802151	45.551483	0.767030	0.030258	0.6	0.9	{'learning_rate': 0.6, 'subsample': 0.9}	0.

In [223…
```python
# # plotting
plt.figure(figsize=(16,6))

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


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


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

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



In [224…
```
model_cv.best_params_
```

Out[224…
```
{'learning_rate': 0.6, 'subsample': 0.3}
```

In [341…
```
# 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[341…
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic',
              params={'learning_rate': 0.6, 'max_depth': 2, 'n_estimators': 200,
                      'objective': 'binary:logistic', 'subsample': 0.3},
              random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              subsample=1, tree_method=None, validate_parameters=False,
              verbosity=None)
```

Prediction on the train set

In [342…
```
# Predictions on the train set
y_train_pred = xgb_bal_adasyn_model.predict(X_train_adasyn)
```

In [343…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_adasyn, y_train_adasyn)
print(confusion)
```

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

In [344…
```
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 [345…
```
# 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:- 0.9999956034003302
Sensitivity:- 1.0
Specificity:- 1.0
```

In [346…
```
# classification_report
print(classification_report(y_train_adasyn, y_train_pred))
```

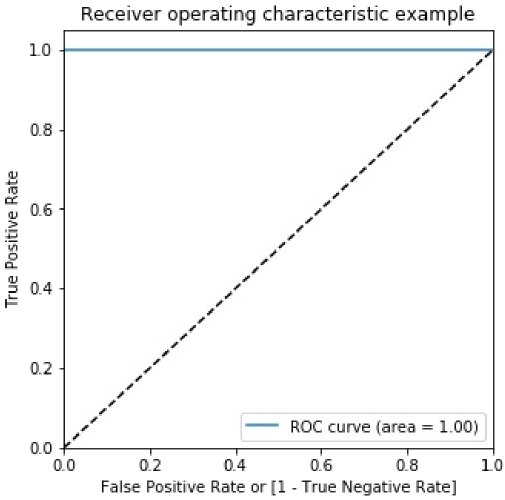
```
              precision    recall  f1-score   support

           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 [347…
```
# Predicted probability
y_train_pred_proba = xgb_bal_adasyn_model.predict_proba(X_train_adasyn)[:,1]
```

In [348…
```
# roc_auc
auc = metrics.roc_auc_score(y_train_adasyn, y_train_pred_proba)
auc
```

Out[348… 1.0

In [349…
```
# Plot the ROC curve
draw_roc(y_train_adasyn, y_train_pred_proba)
```



Prediction on the test set

In [350…
```
# Predictions on the test set
y_test_pred = xgb_bal_adasyn_model.predict(X_test)
```

In [351…
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[56825    41]
 [   21    75]]
```

In [352…
```
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 [353…
```
# 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:- 0.9989115550718023
Sensitivity:- 0.78125
Specificity:- 0.9992790067878873
```

In [354…
```
# classification_report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56866
           1       0.65      0.78      0.71        96

    accuracy                           1.00     56962
   macro avg       0.82      0.89      0.85     56962
weighted avg       1.00      1.00      1.00     56962
```

In [355…

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

In [356…

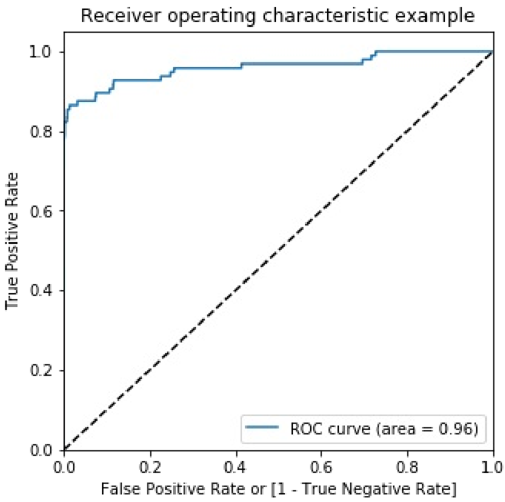
```
# roc_auc
auc = metrics.roc_auc_score(y_test, y_test_pred_proba)
auc
```

Out[356…

```
0.9599176499724499
```

In [357…

```
# 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 [92]:

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

```
Train auc = 0.9897539730968845
Threshold= 0.5311563613510013
```

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.