CREDIT CARD FRAUD DETECTION

Import Libraries

In [23]:

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

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

from scipy import stats
from scipy.stats import norm, skew
from scipy.special import boxcox1p
from scipy.stats import boxcox_normmax

from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

import sklearn
from sklearn import metrics
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.metrics import average_precision_score, precision_recall_curve

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

from sklearn.linear_model import Ridge, Lasso, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from xgboost import XGBClassifier
from xgboost import plot_importance
from sklearn.ensemble import AdaBoostClassifier

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

Explorating data analysis

In [3]:

```
df=pd.read_csv("./creditCard.csv")
df.head()
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	0.
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.9
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	0.1
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	0.1

5 rows × 31 columns

In [5]:

```
#checking the shape
df.shape
```

Out[5]:

```
(284807, 31)
```

In [7]:

```
#checking the datatypes and non-null/null distributions
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

In [10]:

```
#Checking distribution of numerical values in the dataset
df.describe()
```

Out[10]:

	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	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	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	
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

8 rows × 31 columns

In [14]:

```
# Checking for Correlation
corr= df.corr()
corr
```

Out[14]:

	Time	V1	V2	V3	V4	V5	V6	V7	
Time	1.000000	1.173963e-01	-1.059333e-02	-4.196182e-01	-1.052602e-01	1.730721e-01	-6.301647e-02	8.471437e-02	-3.
V1	0.117396	1.000000e+00	4.135835e-16	-1.227819e-15	-9.215150e-16	1.812612e-17	-6.506567e-16	-1.005191e-15	-2.
V2	-0.010593	4.135835e-16	1.000000e+00	3.243764e-16	-1.121065e-15	5.157519e-16	2.787346e-16	2.055934e-16	-5.
V3	-0.419618	-1.227819e-15	3.243764e-16	1.000000e+00	4.711293e-16	-6.539009e-17	1.627627e-15	4.895305e-16	-1.
V4	-0.105260	-9.215150e-16	-1.121065e-15	4.711293e-16	1.000000e+00	-1.719944e-15	-7.491959e-16	-4.104503e-16	5.69
V5	0.173072	1.812612e-17	5.157519e-16	-6.539009e-17	-1.719944e-15	1.000000e+00	2.408382e-16	2.715541e-16	7.43
V6	-0.063016	-6.506567e-16	2.787346e-16	1.627627e-15	-7.491959e-16	2.408382e-16	1.000000e+00	1.191668e-16	-1.
V7	0.084714	-1.005191e-15	2.055934e-16	4.895305e-16	-4.104503e-16	2.715541e-16	1.191668e-16	1.000000e+00	3.34
V8	-0.036949	-2.433822e-16	-5.377041e-17	-1.268779e-15	5.697192e-16	7.437229e-16	-1.104219e-16	3.344412e-16	1.00
V9	-0.008660	-1.513678e-16	1.978488e-17	5.568367e-16	6.923247e-16	7.391702e-16	4.131207e-16	1.122501e-15	4.35
V10	0.030617	7.388135e-17	-3.991394e-16	1.156587e-15	2.232685e-16	-5.202306e-16	5.932243e-17	-7.492834e-17	-2.
V11	-0.247689	2.125498e-16	1.975426e-16	1.576830e-15	3.459380e-16	7.203963e-16	1.980503e-15	1.425248e-16	2.48
V12	0.124348	2.053457e-16	-9.568710e-17	6.310231e-16	-5.625518e-16	7.412552e-16	2.375468e-16	-3.536655e-18	1.83
V13	-0.065902	-2.425603e-17	6.295388e-16	2.807652e-16	1.303306e-16	5.886991e-16	-1.211182e-16	1.266462e-17	-2.
V14	-0.098757	-5.020280e-16	-1.730566e-16	4.739859e-16	2.282280e-16	6.565143e-16	2.621312e-16	2.607772e-16	-8.
V15	-0.183453	3.547782e-16	-4.995814e-17	9.068793e-16	1.377649e-16	-8.720275e-16	-1.531188e-15	-1.690540e-16	4.12

	Time	V1	V2	V3	V4	V5	V6	V7	
V16	0.011903	7.212815e-17	1.177316e-17	8.299445e-16	-9.614528e-16	2.246261e-15	2.623672e-18	5.869302e-17	-5.
V17	-0.073297	-3.879840e-16	-2.685296e-16	7.614712e-16	-2.699612e-16	1.281914e-16	2.015618e-16	2.177192e-16	-2.
V18	0.090438	3.230206e-17	3.284605e-16	1.509897e-16	-5.103644e-16	5.308590e-16	1.223814e-16	7.604126e-17	-3.
V19	0.028975	1.502024e-16	-7.118719e-18	3.463522e-16	-3.980557e-16	-1.450421e-16	-1.865597e-16	-1.881008e-16	-3.
V20	-0.050866	4.654551e-16	2.506675e-16	-9.316409e-16	-1.857247e-16	-3.554057e-16	-1.858755e-16	9.379684e-16	2.03
V21	0.044736	-2.457409e-16	-8.480447e-17	5.706192e-17	-1.949553e-16	-3.920976e-16	5.833316e-17	-2.027779e-16	3.89
V22	0.144059	-4.290944e-16	1.526333e-16	-1.133902e-15	-6.276051e-17	1.253751e-16	-4.705235e-19	-8.898922e-16	2.02
V23	0.051142	6.168652e-16	1.634231e-16	-4.983035e-16	9.164206e-17	-8.428683e-18	1.046712e-16	-4.387401e-16	6.37
V24	-0.016182	-4.425156e-17	1.247925e-17	2.686834e-19	1.584638e-16	-1.149255e-15	-1.071589e-15	7.434913e-18	-1.
V25	-0.233083	-9.605737e-16	-4.478846e-16	-1.104734e-15	6.070716e-16	4.808532e-16	4.562861e-16	-3.094082e-16	-4.
V26	-0.041407	-1.581290e-17	2.057310e-16	-1.238062e-16	-4.247268e-16	4.319541e-16	-1.357067e-16	-9.657637e-16	-1.
V27	-0.005135	1.198124e-16	-4.966953e-16	1.045747e-15	3.977061e-17	6.590482e-16	-4.452461e-16	-1.782106e-15	1.29
V28	-0.009413	2.083082e-15	-5.093836e-16	9.775546e-16	-2.761403e-18	-5.613951e-18	2.594754e-16	-2.776530e-16	-6.
Amount	-0.010596	-2.277087e-01	-5.314089e-01	-2.108805e-01	9.873167e-02	-3.863563e-01	2.159812e-01	3.973113e-01	-1.
Class	-0.012323	-1.013473e-01	9.128865e-02	-1.929608e-01	1.334475e-01	-9.497430e-02	-4.364316e-02	-1.872566e-01	1.98

31 rows × 31 columns

In [9]:

```
# Checking the class distribution of the target variable
df['Class'].value_counts()
```

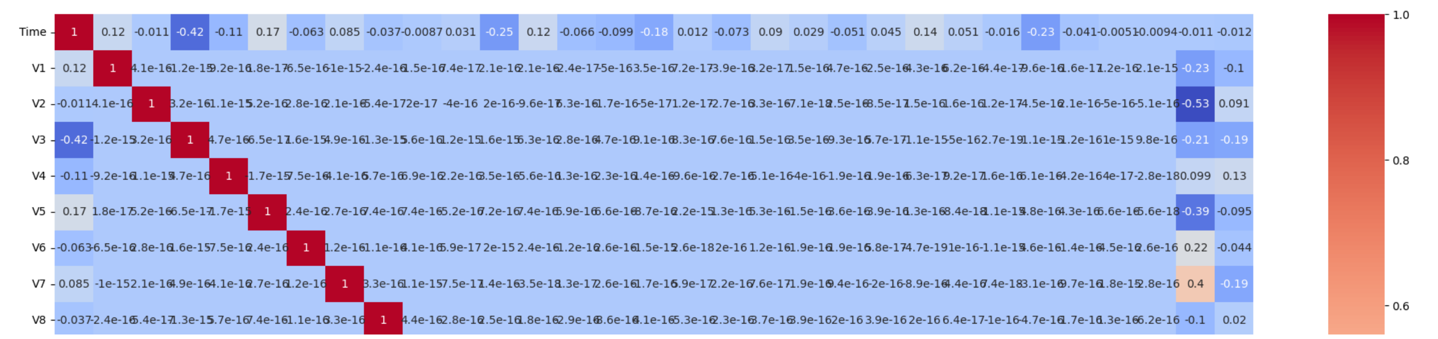
Out[9]:

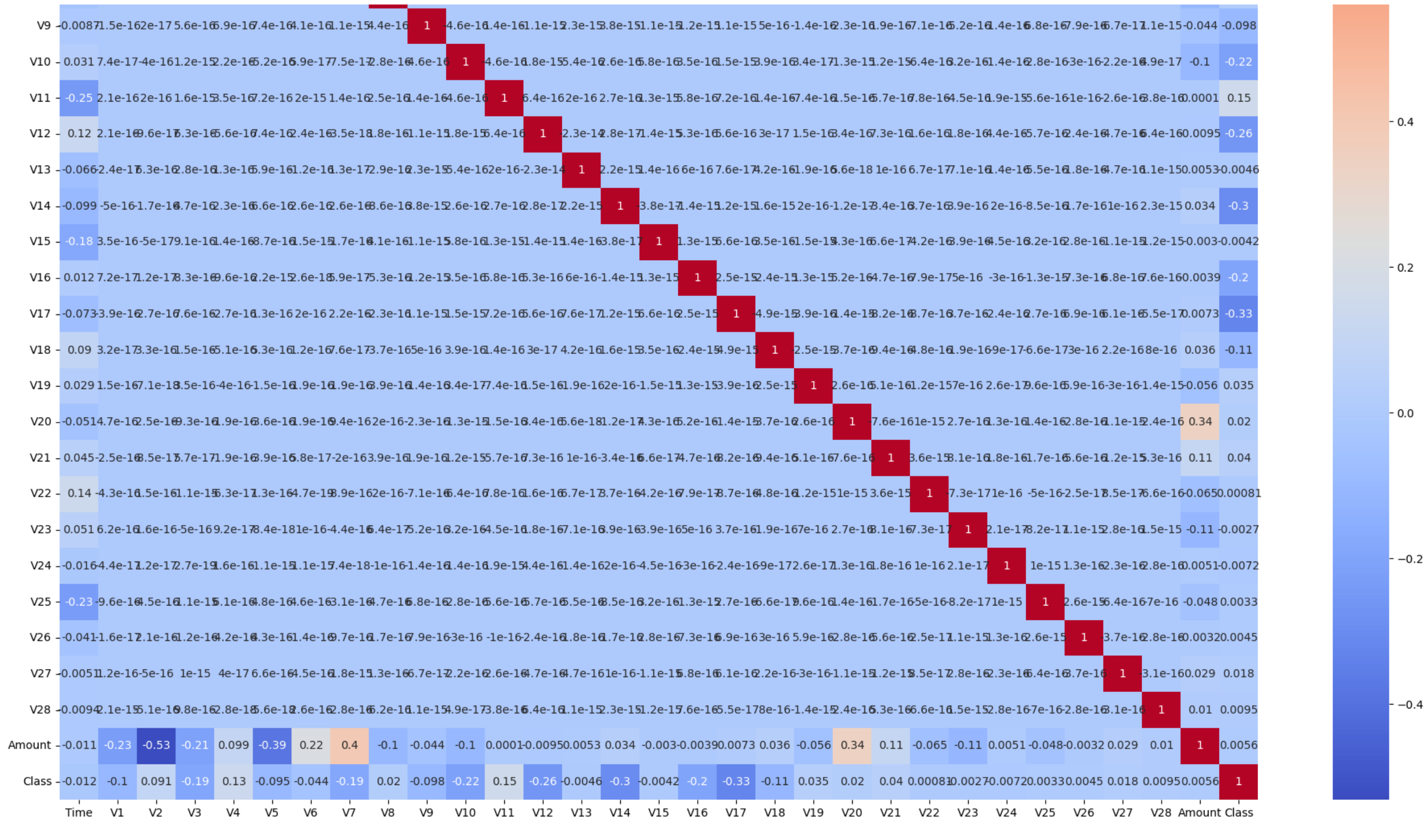
```
0    284315
1       492
Name: Class, dtype: int64
```

In [15]:

```
# Checking the correlation in heatmap
plt.figure(figsize=(24,18))

sns.heatmap(corr, cmap="coolwarm", annot=True)
plt.show()
```





Here we will observe the distribution of our classes

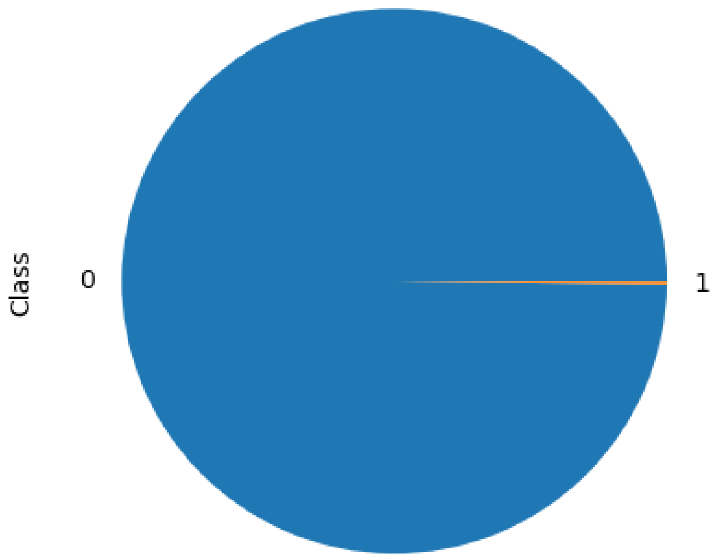
In [11]:

```
# Checking the class distribution of the target variable in percentage
print((df.groupby('Class')['Class'].count()/df['Class'].count()) *100)
((df.groupby('Class')['Class'].count()/df['Class'].count()) *100).plot.pie()
```

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

Out[11]:

<AxesSubplot: ylabel='Class'>



In [12]:

```
# Checking the % distribution of normal vs fraud
```

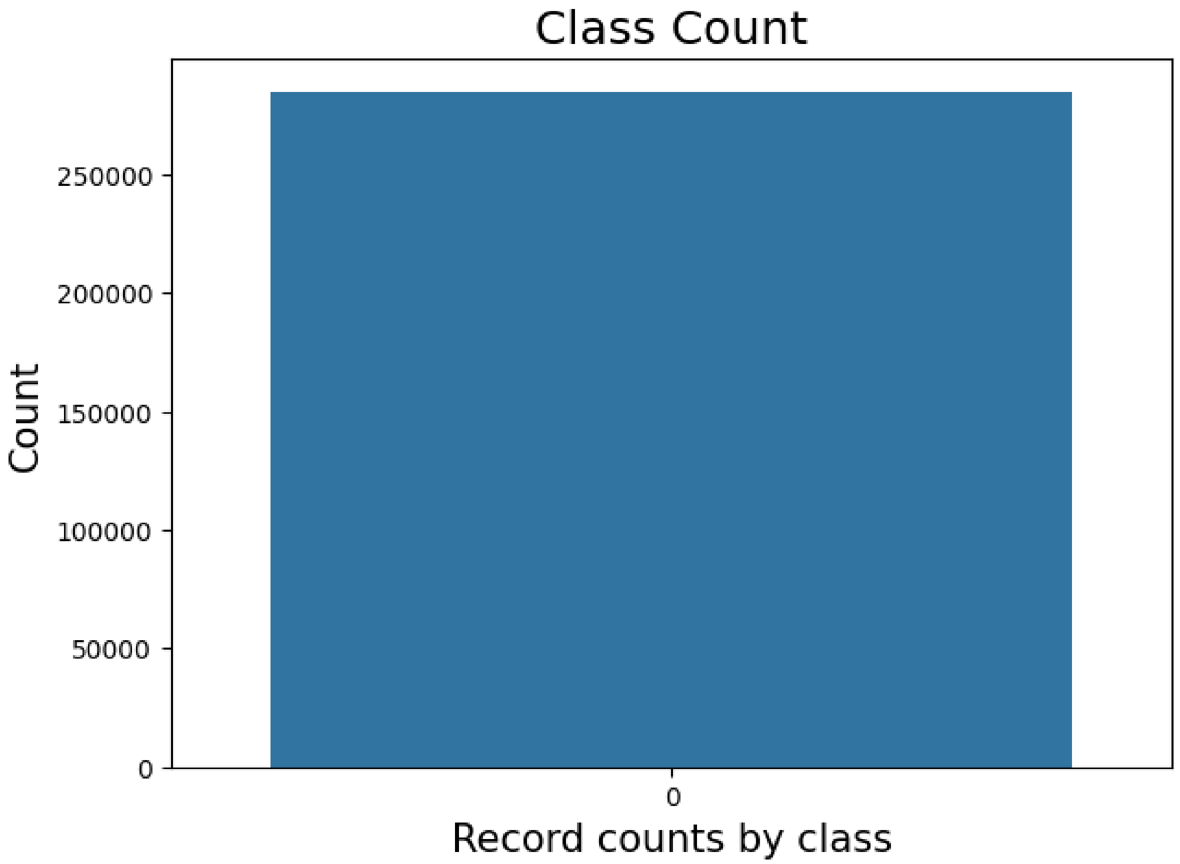
```
classes=df['Class'].value_counts()
normal_share=classes[0]/df['Class'].count()*100
fraud_share=classes[1]/df['Class'].count()*100

print(normal_share)
print(fraud_share)
```

```
99.82725143693798
0.1727485630620034
```

In [13]:

```
# Create a bar plot for the number and percentage of fraudulent vs non-fraudulent transca
tions
plt.figure(figsize=(7,5))
sns.countplot(df['Class'])
plt.title("Class Count", fontsize=18)
plt.xlabel("Record counts by class", fontsize=15)
plt.ylabel("Count", fontsize=15)
plt.show()
```



In [16]:

```
# As time is given in relative fashion, we are using pandas.Timedelta which Represents a
duration, the difference between two times or dates.
Delta_Time = pd.to_timedelta(df['Time'], unit='s')

#Create derived columns Mins and hours
df['Time_Day'] = (Delta_Time.dt.components.days).astype(int)
df['Time_Hour'] = (Delta_Time.dt.components.hours).astype(int)
df['Time_Min'] = (Delta_Time.dt.components.minutes).astype(int)
```

In [17]:

```
# Drop unnecessary columns
# We will drop Time,as we have derived the Day/Hour/Minutes from the time column
df.drop('Time', axis = 1, inplace= True)
# We will keep only derived column hour, as day/minutes might not be very useful
df.drop(['Time_Day', 'Time_Min'], axis = 1, inplace= True)
```

Splitting the data into train and test data

Splitting the data into train and test data

In [18]:

```
# Splitting the dataset into X and y
y= df['Class']
X = df.drop(['Class'], axis=1)
```

In [19]:

```
# Checking some rows of X
X.head()
```

Out[19]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.277838
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.638672
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.771679
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.005274
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.798278

5 rows × 30 columns

In [20]:

```
# Checking some rows of y
y.head()
```

Out[20]:

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

In [21]:

```
# Splitting the dataset using train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=100, test_size=0.
20)
```

Preserve X_test & y_test to evaluate on the test data once you build the model

In [22]:

```
# Checking the spread of data post split
print(np.sum(y))
print(np.sum(y_train))
print(np.sum(y_test))
```

```
492
396
96
```

Plotting the distribution of a variable

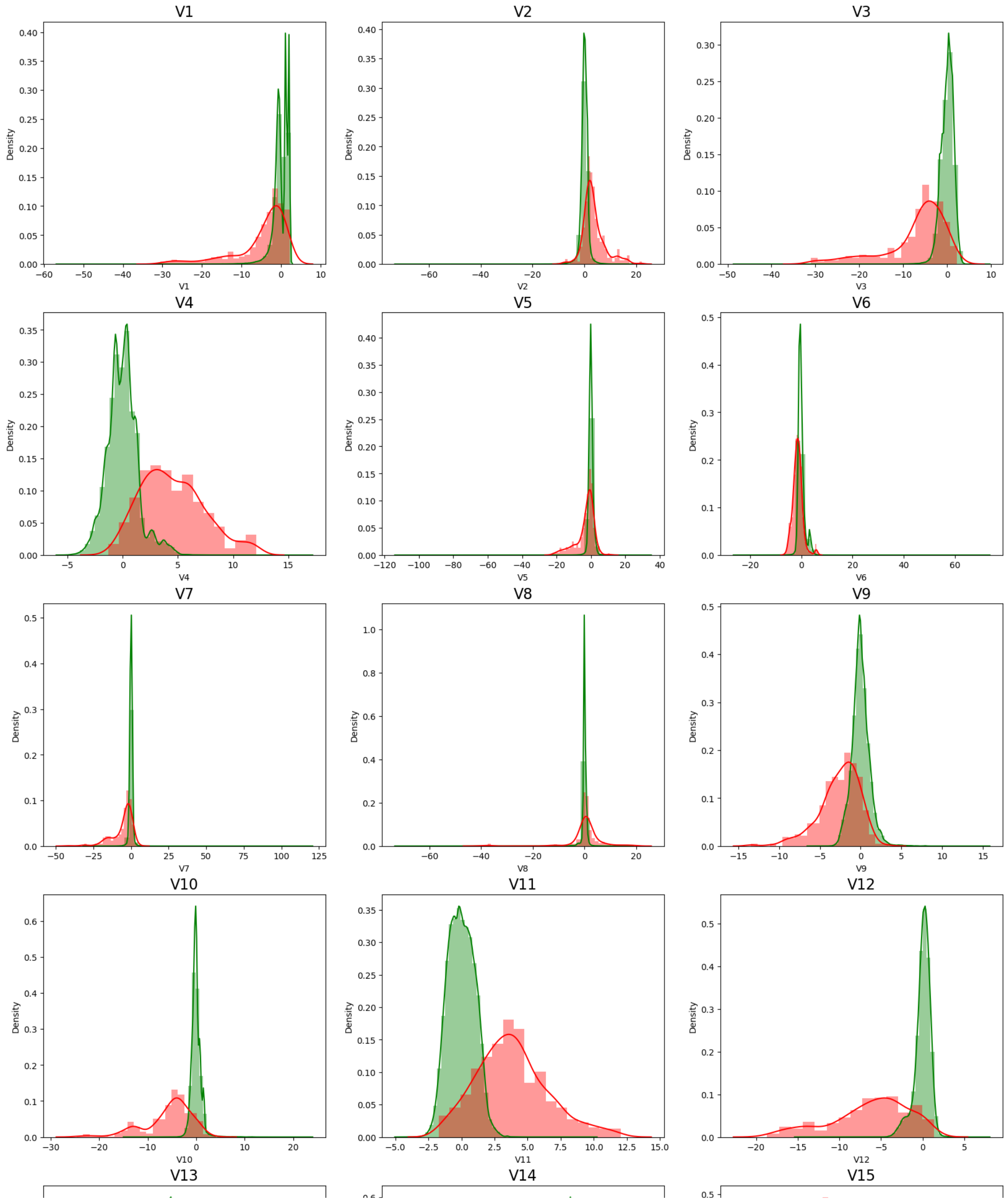
In [27]:

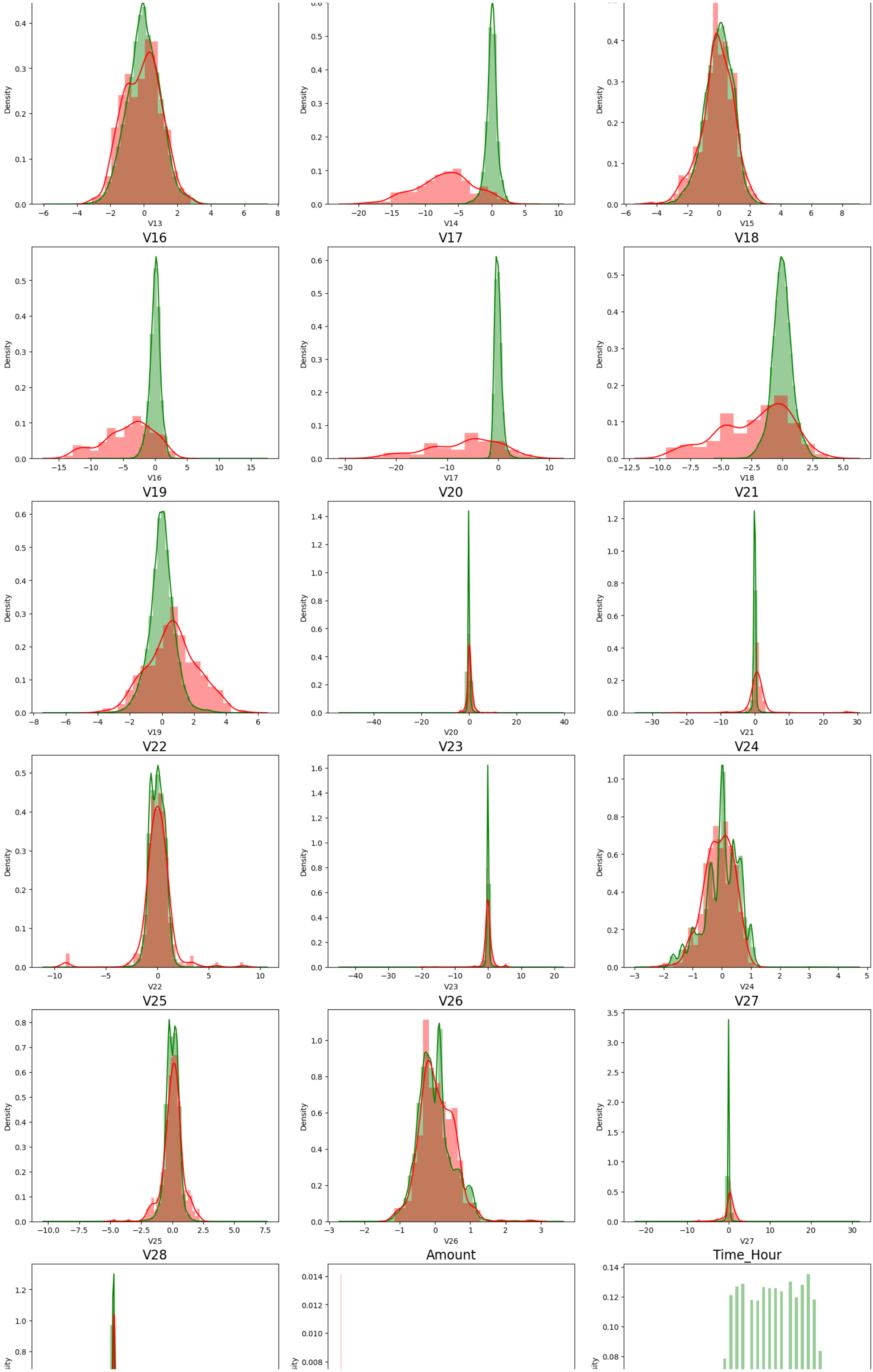
```python
# Accumulating all the column names under one variable
cols = list(X.columns.values)
```

In [28]:

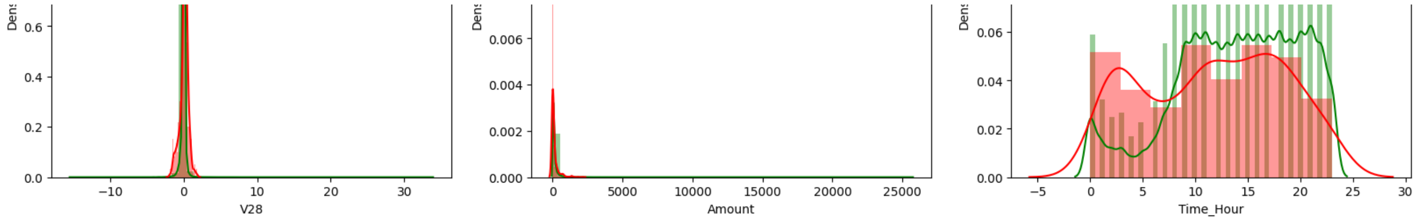
```python
# plot the histogram of a variable from the dataset to see the skewness
normal_records = df.Class == 0
fraud_records = df.Class == 1

plt.figure(figsize=(20, 60))
for n, col in enumerate(cols):
  plt.subplot(10,3,n+1)
  sns.distplot(X[col][normal_records], color='green')
  sns.distplot(X[col][fraud_records], color='red')
  plt.title(col, fontsize=17)
plt.show()
```





V13
V14
V15
V16
V17
V18
V19
V20
V21
V22
V23
V24
V25
V26
V27
V28
Amount
Time_Hour
Density



Model Building

In [29]:

```
#Create a dataframe to store results
df_Results = pd.DataFrame(columns=['Methodology','Model','Accuracy','roc_value','thresho
ld'])
```

In [30]:

```
# Created a common function to plot confusion matrix
def Plot_confusion_matrix(y_test, pred_test):
  cm = confusion_matrix(y_test, pred_test)
  plt.clf()
  plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Accent)
  categoryNames = ['Non-Fraudalent','Fraudalent']
  plt.title('Confusion Matrix - Test Data')
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  ticks = np.arange(len(categoryNames))
  plt.xticks(ticks, categoryNames, rotation=45)
  plt.yticks(ticks, categoryNames)
  s = [['TN','FP'], ['FN', 'TP']]

  for i in range(2):
      for j in range(2):
          plt.text(j,i, str(s[i][j])+" = "+str(cm[i][j]),fontsize=12)
  plt.show()
```

In [31]:

```
# # Created a common function to fit and predict on a Logistic Regression model for both
L1 and L2
def buildAndRunLogisticModels(df_Results, Methodology, X_train,y_train, X_test, y_test )
:

  # Logistic Regression
  from sklearn import linear_model
  from sklearn.model_selection import KFold

  num_C = list(np.power(10.0, np.arange(-10, 10)))
  cv_num = KFold(n_splits=10, shuffle=True, random_state=42)

  searchCV_l2 = linear_model.LogisticRegressionCV(
          Cs= num_C
          ,penalty='l2'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
          ,max_iter=10000
          ,fit_intercept=True
          ,solver='newton-cg'
          ,tol=10
      )

  searchCV_l1 = linear_model.LogisticRegressionCV(
          Cs=num_C
          ,penalty='l1'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
```

```
            ,max_iter=10000
            ,fit_intercept=True
            ,solver='liblinear'
            ,tol=10
        )

    searchCV_l1.fit(X_train, y_train)
    searchCV_l2.fit(X_train, y_train)
    print ('Max auc_roc for l1:', searchCV_l1.scores_[1].mean(axis=0).max())
    print ('Max auc_roc for l2:', searchCV_l2.scores_[1].mean(axis=0).max())

    print("Parameters for l1 regularisations")
    print(searchCV_l1.coef_)
    print(searchCV_l1.intercept_)
    print(searchCV_l1.scores_)

    print("Parameters for l2 regularisations")
    print(searchCV_l2.coef_)
    print(searchCV_l2.intercept_)
    print(searchCV_l2.scores_)


    #find predicted vallues
    y_pred_l1 = searchCV_l1.predict(X_test)
    y_pred_l2 = searchCV_l2.predict(X_test)


    #Find predicted probabilities
    y_pred_probs_l1 = searchCV_l1.predict_proba(X_test)[:,1]
    y_pred_probs_l2 = searchCV_l2.predict_proba(X_test)[:,1]

    # Accuaracy of L2/L1 models
    Accuracy_l2 = metrics.accuracy_score(y_pred=y_pred_l2, y_true=y_test)
    Accuracy_l1 = metrics.accuracy_score(y_pred=y_pred_l1, y_true=y_test)

    print("Accuarcy of Logistic model with l2 regularisation : {0}".format(Accuracy_l2))
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred_l2)
    print("classification Report")
    print(classification_report(y_test, y_pred_l2))

    print("Accuarcy of Logistic model with l1 regularisation : {0}".format(Accuracy_l1))
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred_l1)
    print("classification Report")
    print(classification_report(y_test, y_pred_l1))

    l2_roc_value = roc_auc_score(y_test, y_pred_probs_l2)
    print("l2 roc_value: {0}" .format(l2_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l2)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("l2 threshold: {0}".format(threshold))

    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logi
stic Regression with L2 Regularisation','Accuracy': Accuracy_l2,'roc_value': l2_roc_value
,'threshold': threshold}, index=[0]),ignore_index= True)

    l1_roc_value = roc_auc_score(y_test, y_pred_probs_l1)
    print("l1 roc_value: {0}" .format(l1_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l1)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("l1 threshold: {0}".format(threshold))

    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
```

```
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logi
stic Regression with L1 Regularisation','Accuracy': Accuracy_l1,'roc_value': l1_roc_value
,'threshold': threshold}, index=[0]),ignore_index= True)
    return df_Results
```

In [32]:

```
# Created a common function to fit and predict on a KNN model
def buildAndRunKNNModels(df_Results,Methodology, X_train,y_train, X_test, y_test ):

    #create KNN model and fit the model with train dataset
    knn = KNeighborsClassifier(n_neighbors = 5,n_jobs=16)
    knn.fit(X_train,y_train)
    score = knn.score(X_test,y_test)
    print("model score")
    print(score)

    #Accuracy
    y_pred = knn.predict(X_test)
    KNN_Accuracy = metrics.accuracy_score(y_pred=y_pred, y_true=y_test)
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred)
    print("classification Report")
    print(classification_report(y_test, y_pred))


    knn_probs = knn.predict_proba(X_test)[:, 1]

    # Calculate roc auc
    knn_roc_value = roc_auc_score(y_test, knn_probs)
    print("KNN roc_value: {0}" .format(knn_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, knn_probs)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("KNN threshold: {0}".format(threshold))

    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'KNN'
,'Accuracy': score,'roc_value': knn_roc_value,'threshold': threshold}, index=[0]),ignore
_index= True)

    return df_Results
```

In [ ]:

```
# Created a common function to fit and predict on a Tree models for both gini and entropy
criteria
def buildAndRunTreeModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
    #Evaluate Decision Tree model with 'gini' & 'entropy'
    criteria = ['gini', 'entropy']
    scores = {}

    for c in criteria:
        dt = DecisionTreeClassifier(criterion = c, random_state=42)
        dt.fit(X_train, y_train)
        y_pred = dt.predict(X_test)
        test_score = dt.score(X_test, y_test)
        tree_preds = dt.predict_proba(X_test)[:, 1]
        tree_roc_value = roc_auc_score(y_test, tree_preds)
        scores = test_score
        print(c + " score: {0}" .format(test_score))
        print("Confusion Matrix")
        Plot_confusion_matrix(y_test, y_pred)
        print("classification Report")
```

```
        print(classification_report(y_test, y_pred))
        print(c + " tree_roc_value: {0}" .format(tree_roc_value))
        fpr, tpr, thresholds = metrics.roc_curve(y_test, tree_preds)
        threshold = thresholds[np.argmax(tpr-fpr)]
        print("Tree threshold: {0}".format(threshold))
        roc_auc = metrics.auc(fpr, tpr)
        print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
        plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
        plt.legend(loc=4)
        plt.show()

        df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': '
Tree Model with {0} criteria'.format(c),'Accuracy': test_score,'roc_value': tree_roc_valu
e,'threshold': threshold}, index=[0]),ignore_index= True)

  return df_Results
```

In [33]:

```
# Created a common function to fit and predict on a Random Forest model
def buildAndRunRandomForestModels(df_Results, Methodology, X_train,y_train, X_test, y_te
st ):
  #Evaluate Random Forest model

  # Create the model with 100 trees
  RF_model = RandomForestClassifier(n_estimators=100,
                                bootstrap = True,
                                max_features = 'sqrt', random_state=42)
  # Fit on training data
  RF_model.fit(X_train, y_train)
  RF_test_score = RF_model.score(X_test, y_test)
  RF_model.predict(X_test)

  print('Model Accuracy: {0}'.format(RF_test_score))


  # Actual class predictions
  rf_predictions = RF_model.predict(X_test)

  print("Confusion Matrix")
  Plot_confusion_matrix(y_test, rf_predictions)
  print("classification Report")
  print(classification_report(y_test, rf_predictions))

  # Probabilities for each class
  rf_probs = RF_model.predict_proba(X_test)[:, 1]

  # Calculate roc auc
  roc_value = roc_auc_score(y_test, rf_probs)

  print("Random Forest roc_value: {0}" .format(roc_value))
  fpr, tpr, thresholds = metrics.roc_curve(y_test, rf_probs)
  threshold = thresholds[np.argmax(tpr-fpr)]
  print("Random Forest threshold: {0}".format(threshold))
  roc_auc = metrics.auc(fpr, tpr)
  print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
  plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
  plt.legend(loc=4)
  plt.show()

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Rand
om Forest','Accuracy': RF_test_score,'roc_value': roc_value,'threshold': threshold}, inde
x=[0]),ignore_index= True)

  return df_Results
```

In [34]:

```
# Created a common function to fit and predict on a XGBoost model
def buildAndRunXGBoostModels(df_Results, Methodology,X_train,y_train, X_test, y_test ):
  #Evaluate XGboost model
```

```
    XGBmodel = XGBClassifier(random_state=42)
    XGBmodel.fit(X_train, y_train)
    y_pred = XGBmodel.predict(X_test)

    XGB_test_score = XGBmodel.score(X_test, y_test)
    print('Model Accuracy: {0}'.format(XGB_test_score))

    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred)
    print("classification Report")
    print(classification_report(y_test, y_pred))
    # Probabilities for each class
    XGB_probs = XGBmodel.predict_proba(X_test)[:, 1]

    # Calculate roc auc
    XGB_roc_value = roc_auc_score(y_test, XGB_probs)

    print("XGboost roc_value: {0}" .format(XGB_roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("XGBoost threshold: {0}".format(threshold))
    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'XGBo
ost','Accuracy': XGB_test_score,'roc_value': XGB_roc_value,'threshold': threshold}, index
=[0]),ignore_index= True)

    return df_Results
```

In [35]:

```
# Created a common function to fit and predict on a SVM model
def buildAndRunSVMModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
    #Evaluate SVM model with sigmoid kernel  model
    from sklearn.svm import SVC
    from sklearn.metrics import accuracy_score
    from sklearn.metrics import roc_auc_score

    clf = SVC(kernel='sigmoid', random_state=42)
    clf.fit(X_train,y_train)
    y_pred_SVM = clf.predict(X_test)
    SVM_Score = accuracy_score(y_test,y_pred_SVM)
    print("accuracy_score : {0}".format(SVM_Score))
    print("Confusion Matrix")
    Plot_confusion_matrix(y_test, y_pred_SVM)
    print("classification Report")
    print(classification_report(y_test, y_pred_SVM))

    # Run classifier
    classifier = SVC(kernel='sigmoid' , probability=True)
    svm_probs = classifier.fit(X_train, y_train).predict_proba(X_test)[:, 1]

    # Calculate roc auc
    roc_value = roc_auc_score(y_test, svm_probs)

    print("SVM roc_value: {0}" .format(roc_value))
    fpr, tpr, thresholds = metrics.roc_curve(y_test, svm_probs)
    threshold = thresholds[np.argmax(tpr-fpr)]
    print("SVM threshold: {0}".format(threshold))
    roc_auc = metrics.auc(fpr, tpr)
    print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
    plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
    plt.legend(loc=4)
    plt.show()

    df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'SVM'
,'Accuracy': SVM_Score,'roc_value': roc_value,'threshold': threshold}, index=[0]),ignore
```

```
_index= True)

  return df_Results
```

Build different models on the imbalanced dataset and see the result

Perform cross validation with RepeatedKFold

In [36]:

```
#Lets perfrom RepeatedKFold and check the results
from sklearn.model_selection import RepeatedKFold
rkf = RepeatedKFold(n_splits=5, n_repeats=10, random_state=None)
# X is the feature set and y is the target
for train_index, test_index in rkf.split(X,y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train_cv, X_test_cv = X.iloc[train_index], X.iloc[test_index]
    y_train_cv, y_test_cv = y.iloc[train_index], y.iloc[test_index]
```

```
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     4      9     19 ... 2
84790 284793 284799]
TRAIN: [      0      1      2 ... 284803 284805 284806] TEST: [     5     10     15 ... 2
84796 284797 284804]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     3     11     14 ... 2
84777 284792 284794]
TRAIN: [      2      3      4 ... 284803 284804 284805] TEST: [     0      1      6 ... 2
84795 284798 284806]
TRAIN: [      0      1      3 ... 284799 284804 284806] TEST: [     2     12     18 ... 2
84802 284803 284805]
TRAIN: [      0      1      2 ... 284802 284805 284806] TEST: [     8      9     12 ... 2
84801 284803 284804]
TRAIN: [      1      3      4 ... 284803 284804 284805] TEST: [     0      2      5 ... 2
84776 284791 284806]
TRAIN: [      0      2      4 ... 284804 284805 284806] TEST: [     1      3     20 ... 2
84783 284794 284798]
TRAIN: [      0      1      2 ... 284803 284804 284806] TEST: [    16     19     27 ... 2
84795 284800 284805]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     4      6     10 ... 2
84796 284799 284802]
TRAIN: [      0      1      2 ... 284803 284804 284805] TEST: [     4      5      9 ... 2
84787 284790 284806]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [    21     30     32 ... 2
84792 284794 284801]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     3     12     15 ... 2
84795 284799 284802]
TRAIN: [      2      3      4 ... 284801 284802 284806] TEST: [     0      1      6 ... 2
84803 284804 284805]
TRAIN: [      0      1      3 ... 284804 284805 284806] TEST: [     2      7      8 ... 2
84793 284797 284800]
TRAIN: [      0      1      3 ... 284804 284805 284806] TEST: [     2      4     10 ... 2
84789 284790 284802]
TRAIN: [      0      1      2 ... 284803 284804 284806] TEST: [     9     12     21 ... 2
84799 284801 284805]
TRAIN: [      0      1      2 ... 284803 284804 284805] TEST: [     5      6      7 ... 2
84792 284796 284806]
TRAIN: [      2      3      4 ... 284802 284805 284806] TEST: [     0      1      8 ... 2
84800 284803 284804]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     3     16     22 ... 2
84782 284787 284797]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     3      9     24 ... 2
84790 284795 284803]
TRAIN: [      0      2      3 ... 284804 284805 284806] TEST: [     1      5      6 ... 2
84793 284796 284799]
TRAIN: [      0      1      3 ... 284803 284804 284806] TEST: [     2      4      7 ... 2
84797 284801 284805]
TRAIN: [      1      2      3 ... 284803 284804 284805] TEST: [     0      8     12 ... 2
84792 284794 284806]
TRAIN: [      0      1      2 ... 284803 284805 284806] TEST: [    11     18     27 ... 2
84800 284802 284804]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     8      9     10 ... 2
```

```
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [      8      9     10 ... 2
84793 284797 284803]
TRAIN: [      0      4      5 ... 284804 284805 284806] TEST: [      1      2      3 ... 2
84781 284783 284789]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [      4     12     18 ... 2
84790 284794 284795]
TRAIN: [      0      1      2 ... 284799 284802 284803] TEST: [      7     21     25 ... 2
84804 284805 284806]
TRAIN: [      1      2      3 ... 284804 284805 284806] TEST: [      0      5      6 ... 2
84785 284799 284802]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     10     13     15 ... 2
84790 284793 284797]
TRAIN: [      0      1      3 ... 284804 284805 284806] TEST: [      2      6      9 ... 2
84794 284796 284801]
TRAIN: [      2      3      4 ... 284803 284804 284805] TEST: [      0      1      7 ... 2
84786 284795 284806]
TRAIN: [      0      1      2 ... 284803 284805 284806] TEST: [      8     26     36 ... 2
84798 284799 284804]
TRAIN: [      0      1      2 ... 284801 284804 284806] TEST: [      3      4      5 ... 2
84802 284803 284805]
TRAIN: [      0      1      3 ... 284804 284805 284806] TEST: [      2      6      8 ... 2
84793 284798 284803]
TRAIN: [      0      1      2 ... 284803 284804 284806] TEST: [     16     18     29 ... 2
84800 284801 284805]
TRAIN: [      1      2      3 ... 284802 284803 284805] TEST: [      0      7     14 ... 2
84796 284804 284806]
TRAIN: [      0      2      3 ... 284804 284805 284806] TEST: [      1      9     25 ... 2
84779 284782 284788]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [      3      4      5 ... 2
84797 284799 284802]
TRAIN: [      1      3      4 ... 284803 284804 284805] TEST: [      0      2      6 ... 2
84792 284795 284806]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     11     18     26 ... 2
84787 284798 284803]
TRAIN: [      0      2      4 ... 284803 284805 284806] TEST: [      1      3     10 ... 2
84794 284799 284804]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [      5      9     12 ... 2
84780 284784 284801]
TRAIN: [      0      1      2 ... 284803 284804 284806] TEST: [      4     16     20 ... 2
84800 284802 284805]
TRAIN: [      1      2      3 ... 284804 284805 284806] TEST: [      0      8      9 ... 2
84778 284782 284798]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [     16     22     27 ... 2
84795 284799 284802]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [      4     10     12 ... 2
84793 284794 284800]
TRAIN: [      0      1      2 ... 284800 284802 284806] TEST: [      3      5      6 ... 2
84803 284804 284805]
TRAIN: [      0      3      4 ... 284803 284804 284805] TEST: [      1      2     18 ... 2
84789 284791 284806]
```

In [ ]:

```
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results,"RepeatedKFold Cross Validation", X_tra
in_cv,y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,
y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
```

```
df_Results = buildAndRunTreeModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv
,y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results,"RepeatedKFold Cross Validation",X_
train_cv,y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"RepeatedKFold Cross Validation",X_train
_cv,y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,"RepeatedKFold Cross Validation",X_train_cv,
y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

In [ ]:

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Results for cross validation with RepeatedKFold:

Looking at Accuracy and ROC value we have "Logistic Regression with L2 Regularisation" which has provided best results for cross validation with RepeatedKFold technique

Perform cross validation with StratifiedKFold

In [39]:

```
#Lets perfrom StratifiedKFold and check the results
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, random_state=None)
# X is the feature set and y is the target
for train_index, test_index in skf.split(X,y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train_SKF_cv, X_test_SKF_cv = X.iloc[train_index], X.iloc[test_index]
    y_train_SKF_cv, y_test_SKF_cv = y.iloc[train_index], y.iloc[test_index]
```

```
TRAIN: [ 30473  30496  31002 ... 284804 284805 284806] TEST: [     0      1      2 ... 5701
7 57018 57019]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [ 30473  30496  31002 ... 1
13964 113965 113966]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [ 81609  82400  83053 ... 1
70946 170947 170948]
TRAIN: [      0      1      2 ... 284804 284805 284806] TEST: [150654 150660 150661 ... 2
27866 227867 227868]
TRAIN: [      0      1      2 ... 227866 227867 227868] TEST: [212516 212644 213092 ... 2
84804 284805 284806]
```

In [ ]:

```
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results,"StratifiedKFold Cross Validation", X_t
```

```
rain_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results,"StratifiedKFold Cross Validation",X_train_S
KF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results,"StratifiedKFold Cross Validation",X_train_
SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results,"StratifiedKFold Cross Validation",
X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"StratifiedKFold Cross Validation",X_tra
in_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,"StratifiedKFold Cross Validation",X_train_S
KF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

In [ ]:

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Results for cross validation with StratifiedKFold:

Looking at the ROC value we have Logistic Regression with L2 Regularisation has provided best results for cross validation with StratifiedKFold technique

Conclusion :

As the results show Logistic Regression with L2 Regularisation for StratifiedKFold cross validation provided best results

Proceed with the model which shows the best result

Apply the best hyperparameter on the model Predict on the test dataset

In [ ]:

```
# Logistic Regression
from sklearn import linear_model #import the package
```

```
from sklearn.model_selection import KFold

num_C = list(np.power(10.0, np.arange(-10, 10)))
cv_num = KFold(n_splits=10, shuffle=True, random_state=42)

clf = linear_model.LogisticRegressionCV(
          Cs= num_C
          ,penalty='l2'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
          ,max_iter=10000
          ,fit_intercept=True
          ,solver='newton-cg'
          ,tol=10
      )

clf.fit(X_train_SKF_cv, y_train_SKF_cv)
print ('Max auc_roc for l2:', clf.scores_[1].mean(axis=0).max())


print("Parameters for l2 regularisations")
print(clf.coef_)
print(clf.intercept_)
print(clf.scores_)


#find predicted vallues
y_pred_l2 = clf.predict(X_test)


#Find predicted probabilities
y_pred_probs_l2 = clf.predict_proba(X_test)[:,1]


# Accuaracy of L2/L1 models
Accuracy_l2 = metrics.accuracy_score(y_pred=y_pred_l2, y_true=y_test)


print("Accuarcy of Logistic model with l2 regularisation : {0}".format(Accuracy_l2))


from sklearn.metrics import roc_auc_score
l2_roc_value = roc_auc_score(y_test, y_pred_probs_l2)
print("l2 roc_value: {0}" .format(l2_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_probs_l2)
threshold = thresholds[np.argmax(tpr-fpr)]
print("l2 threshold: {0}".format(threshold))
```

In [ ]:

```
# Checking for the coefficient values
clf.coef_
```

In [ ]:

```
# Creating a dataframe with the coefficient values
coefficients = pd.concat([pd.DataFrame(X.columns),pd.DataFrame(np.transpose(clf.coef_))]
, axis = 1)
coefficients.columns = ['Feature','Importance Coefficient']
```

In [ ]:

```
coefficients
```

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 [ ]:

```
# Plotting the coefficient values
plt.figure(figsize=(20,5))
sns.barplot(x='Feature', y='Importance Coefficient', data=coefficients)
plt.title("Logistic Regression with L2 Regularisation Feature Importance", fontsize=18)

plt.show()
```

Hence it implies that V4, v5,V11 has + ve importance whereas V10, V12, V14 seems to have -ve impact on the predictions

Model building with balancing Classes

Perform class balancing with :

Random Oversampling

SMOTE

ADASYN

Oversampling with RandomOverSampler with StratifiedKFold Cross Validation

We will use Random Oversampling method to handle the class imbalance

In [ ]:

```
# Creating the dataset with RandomOverSampler and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn.over_sampling import RandomOverSampler

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    ROS = RandomOverSampler(sampling_strategy=0.5)
    X_over, y_over= ROS.fit_resample(X_train, y_train)

X_over = pd.DataFrame(data=X_over, columns=cols)
```

In [ ]:

```
Data_Imbalance_Handiling  = "Random Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results , Data_Imbalance_Handiling , X_over, y
_over, X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
```

```
df_Results = buildAndRunKNNModels(df_Results , Data_Imbalance_Handiling,X_over, y_over,
X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results , Data_Imbalance_Handiling,X_over, y_over,
X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results , Data_Imbalance_Handiling,X_over,
y_over, X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results , Data_Imbalance_Handiling,X_over, y_ove
r, X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )
```

In [ ]:

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Results for Random Oversampling with StratifiedKFold technique:

Looking at the Accuracy and ROC value we have XGBoost which has provided best results for Random Oversampling and StratifiedKFold technique

Oversampling with SMOTE Oversampling

We will use SMOTE Oversampling method to handle the class imbalance

In [ ]:

```
# Creating dataframe with Smote and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    SMOTE = over_sampling.SMOTE(random_state=0)
    X_train_Smote, y_train_Smote= SMOTE.fit_resample(X_train, y_train)

X_train_Smote = pd.DataFrame(data=X_train_Smote,   columns=cols)
```

In [ ]:

```
Data_Imbalance_Handiling  = "SMOTE Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
```

```
df_Results = buildAndRunLogisticModels(df_Results, Data_Imbalance_Handiling, X_train_Smot
e, y_train_Smote , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y_
train_Smote , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results, Data_Imbalance_Handiling, X_train_Smote, y
_train_Smote , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results, Data_Imbalance_Handiling, X_train_
Smote, y_train_Smote , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results, Data_Imbalance_Handiling, X_train_Smote
, y_train_Smote , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
```

In [ ]:

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Results for SMOTE Oversampling with StratifiedKFold:

Looking at Accuracy and ROC value we have XGBoost which has provided best results for SMOTE Oversampling with StratifiedKFold technique

Oversampling with ADASYN Oversampling

We will use ADASYN Oversampling method to handle the class imbalance

In [ ]:

```
# Creating dataframe with ADASYN and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

skf = StratifiedKFold(n_splits=5, random_state=None)

for fold, (train_index, test_index) in enumerate(skf.split(X,y), 1):
    X_train = X.loc[train_index]
    y_train = y.loc[train_index]
    X_test = X.loc[test_index]
    y_test = y.loc[test_index]
    ADASYN = over_sampling.ADASYN(random_state=0)
    X_train_ADASYN, y_train_ADASYN= ADASYN.fit_resample(X_train, y_train)
```

```
X_train_ADASYN = pd.DataFrame(data=X_train_ADASYN,   columns=cols)
```

In [ ]:

```
Data_Imbalance_Handiling  = "ADASYN Oversampling with StratifiedKFold CV "
#Run Logistic Regression with L1 And L2 Regularisation
print("Logistic Regression with L1 And L2 Regularisation")
start_time = time.time()
df_Results = buildAndRunLogisticModels(df_Results, Data_Imbalance_Handiling, X_train_ADAS
YN, y_train_ADASYN , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run KNN Model
print("KNN Model")
start_time = time.time()
df_Results = buildAndRunKNNModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN, y_
train_ADASYN , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN, y
_train_ADASYN , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run Random Forest Model
print("Random Forest Model")
start_time = time.time()
df_Results = buildAndRunRandomForestModels(df_Results, Data_Imbalance_Handiling,X_train_A
DASYN, y_train_ADASYN , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

#Run XGBoost Model
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results, Data_Imbalance_Handiling,X_train_ADASYN
, y_train_ADASYN , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
```

In [ ]:

```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Results for ADASYN Oversampling with StratifiedKFold:

Looking at Accuracy and ROC value we have XGBoost which has provided best results for ADASYN Oversampling with StratifiedKFold technique

Overall conclusion after running the models on Oversampled data :

Looking at above results it seems XGBOOST model with Random Oversampling with StratifiedKFold CV has provided the best results under the category of all oversampling techniques. So we will try to tune the hyperparameters of this model to get best results.

Hyperparameter Tuning

HPT - Xgboost Regression

In [ ]:

```
# Performing Hyperparameter tuning
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
param_test = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2),
 'n_estimators':range(60,130,150),
 'learning_rate':[0.05,0.1,0.125,0.15,0.2],
 'gamma':[i/10.0 for i in range(0,5)],
 'subsample':[i/10.0 for i in range(7,10)],
 'colsample_bytree':[i/10.0 for i in range(7,10)]
}

gsearch1 = RandomizedSearchCV(estimator = XGBClassifier(base_score=0.5, booster='gbtree',
colsample_bylevel=1,
              colsample_bynode=1,max_delta_step=0,
              missing=None, n_jobs=-1,
              nthread=None, objective='binary:logistic', random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, verbosity=1),
              param_distributions = param_test, n_iter=5,scoring='roc_auc',n_jobs=-1, cv
=5)

gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

In [ ]:

```
# Creating XGBoost model with selected hyperparameters
from xgboost import XGBClassifier

clf = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.7, gamma=0.2,
              learning_rate=0.125, max_delta_step=0, max_depth=7,
              min_child_weight=5, missing=None, n_estimators=60, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=0.8, verbosity=1)

# fit on the dataset
clf.fit(X_over, y_over )
XGB_test_score = clf.score(X_test, y_test)
print('Model Accuracy: {0}'.format(XGB_test_score))

# Probabilities for each class
XGB_probs = clf.predict_proba(X_test)[:, 1]

# Calculate roc auc
XGB_roc_value = roc_auc_score(y_test, XGB_probs)

print("XGboost roc_value: {0}" .format(XGB_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
threshold = thresholds[np.argmax(tpr-fpr)]
print("XGBoost threshold: {0}".format(threshold))
```

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

In [ ]:

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

In [ ]:

```
# Calculate roc auc
XGB_roc_value = roc_auc_score(y_test, XGB_probs)

print("XGboost roc_value: {0}" .format(XGB_roc_value))
fpr, tpr, thresholds = metrics.roc_curve(y_test, XGB_probs)
threshold = thresholds[np.argmax(tpr-fpr)]
print("XGBoost threshold: {0}".format(threshold))
```

Conclusion

In [ ]:

```
In the oversample cases, of all the models we build found that the XGBOOST model with Ra
ndom Oversampling with StratifiedKFold CV gave us the best accuracy and ROC on oversample
d data. Post that we performed hyperparameter tuning and got the below metrices :

XGboost roc_value: 0.9815403079438694
XGBoost threshold: 0.01721232570707798

However, of all the models we created we found Logistic Regression with L2 Regularisatio
n for StratifiedKFold cross validation (without any oversampling or undersampling) gave
us the best result.
```