In [48]:
```
# Importing python libraries :

import numpy as np
import pandas as pd
import time

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

# import machine learning and stats libraries:
from scipy import stats
from scipy.stats import norm, skew
from scipy.special import boxcox1p
from scipy.stats import boxcox_normmax

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

from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler, PowerTransformer

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.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
# Import:
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

#install scikit-optimize
!pip install scikit-optimize
from skopt import BayesSearchCV


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

```
Requirement already satisfied: scikit-optimize in /usr/local/lib/python3.6/dist-packages (0.7.4)
Requirement already satisfied: scipy>=0.18.0 in /usr/local/lib/python3.6/dist-packages (from scikit-optimize) (1.4.1)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (from scikit-optimize) (0.14.1)
Requirement already satisfied: pyaml>=16.9 in /usr/local/lib/python3.6/dist-packages (from scikit-optimize) (20.4.0)
Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.6/dist-packages (from scikit-optimize) (1.18.2)
Requirement already satisfied: scikit-learn>=0.19.1 in /usr/local/lib/python3.6/dist-packages (from scikit-optimize) (0.22.
2.post1)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.6/dist-packages (from pyaml>=16.9->scikit-optimize) (3.13)
```

EDA and Data Preparation

In [0]:
```
#To read csv File from locally stored file
df_credit = pd.read_csv('desktop/creditcard.csv')
df_credit.head()
```

In [50]:
```
#To download data directly from Kaggle and store it on Google Drive to User it with Google Colab
"""
! pip install -q kaggle
from google.colab import drive
drive.mount('/content/drive')
! mkdir ~/.kaggle
! cp /content/drive/'My Drive'/'Credit Card Fraud Assignment'/kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
!kaggle datasets download -d mlg-ulb/creditcardfraud
!unzip creditcardfraud.zip
#Create DataFrame from the csv file
df = pd.read_csv('creditcard.csv')
df.head()
"""
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=T
rue).
mkdir: cannot create directory ‘/root/.kaggle’: File exists
creditcardfraud.zip: Skipping, found more recently modified local copy (use --force to force download)
Archive:  creditcardfraud.zip
replace creditcard.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
```

Out[50]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.3111
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.1437
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.1659
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.2879
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.1196

In [51]:
```
# Examining the dataset imported:
df.shape
```

Out[51]:
```
(284807, 31)
```

In [52]:
```
# Lets check the numeric distribution of the data:
df.describe()
```

Out[52]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15	2.010663e-15	-1.694249e-15	-1.927028e-16	-3.137024e-
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+

In [53]:
```
#Examining the data frame for the shape, datatypes, NULls etc
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 [54]:
```
#Check the fraud/Non_Fraud related records
df['Class'].value_counts()
```

Out[54]:
```
0    284315
1       492
Name: Class, dtype: int64
```

In [55]:
```
#find % values of class
(df.groupby('Class')['Class'].count()/df['Class'].count()) *100
```

Out[55]:
```
Class
0    99.827251
1     0.172749
Name: Class, dtype: float64
```

In [56]:
```
#check if any null values
df.isnull().sum()
```

Out[56]:
```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

In [57]:
```
#observe the different feature type present in the data
#lets check data types of the features
df.dtypes
```

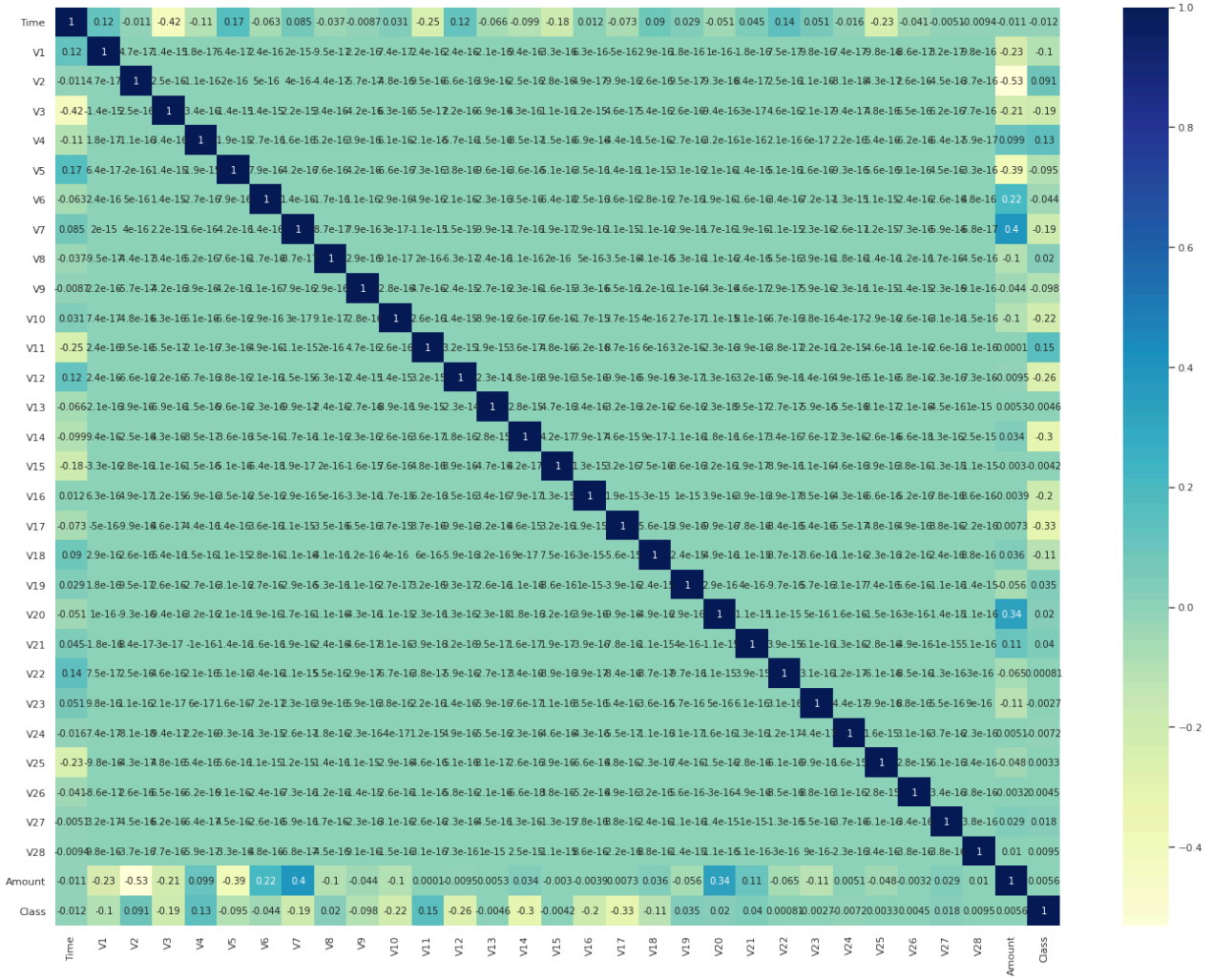
Out[57]:
```
Time      float64
V1        float64
V2        float64
V3        float64
V4        float64
V5        float64
V6        float64
V7        float64
V8        float64
V9        float64
V10       float64
V11       float64
V12       float64
V13       float64
V14       float64
V15       float64
V16       float64
V17       float64
V18       float64
V19       float64
V20       float64
V21       float64
V22       float64
V23       float64
V24       float64
V25       float64
V26       float64
V27       float64
V28       float64
Amount    float64
Class       int64
dtype: object
```

In [58]:
```
# Finding the initial full correlation in the dataset:

# correlation matrix
cor = df.corr()
cor
```

Out[58]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	
Time	1.000000	1.173963e-01	-1.059333e-02	-4.196182e-01	-1.052602e-01	1.730721e-01	-6.301647e-02	8.471437e-02	-3.694943e-02	-8.660434e-03	3.06
V1	0.117396	1.000000e+00	4.697350e-17	-1.424390e-15	1.755316e-17	6.391162e-17	2.398071e-16	1.991550e-15	-9.490675e-17	2.169581e-16	7.43
V2	-0.010593	4.697350e-17	1.000000e+00	2.512175e-16	-1.126388e-16	-2.039868e-16	5.024680e-16	3.966486e-16	-4.413984e-17	-5.728718e-17	-4.
V3	-0.419618	-1.424390e-15	2.512175e-16	1.000000e+00	-3.416910e-16	-1.436514e-15	1.431581e-15	2.168574e-15	3.433113e-16	-4.233770e-16	6.28
V4	-0.105260	1.755316e-17	-1.126388e-16	-3.416910e-16	1.000000e+00	-1.940929e-15	-2.712659e-16	1.556330e-16	5.195643e-16	3.859585e-16	6.05
V5	0.173072	6.391162e-17	-2.039868e-16	-1.436514e-15	-1.940929e-15	1.000000e+00	7.926364e-16	-4.209851e-16	7.589187e-16	4.205206e-16	-6.
V6	-0.063016	2.398071e-16	5.024680e-16	1.431581e-15	-2.712659e-16	7.926364e-16	1.000000e+00	1.429426e-16	-1.707421e-16	1.114447e-16	2.85
V7	0.084714	1.991550e-15	3.966486e-16	2.168574e-15	1.556330e-16	-4.209851e-16	1.429426e-16	1.000000e+00	-8.691834e-17	7.933251e-16	3.04
V8	-0.036949	-9.490675e-17	-4.413984e-17	3.433113e-16	5.195643e-16	7.589187e-16	-1.707421e-16	-8.691834e-17	1.000000e+00	2.900829e-16	9.05
V9	-0.008660	2.169581e-16	-5.728718e-17	-4.233770e-16	3.859585e-16	4.205206e-16	1.114447e-16	7.933251e-16	2.900829e-16	1.000000e+00	-2.
V10	0.030617	7.433820e-17	-4.782388e-16	6.289267e-16	6.055490e-16	-6.601716e-16	2.850776e-16	3.043333e-17	9.051847e-17	-2.771761e-16	1.00(
V11	-0.247689	2.438580e-16	9.468995e-16	-5.501758e-17	-2.083600e-16	7.342759e-16	4.865799e-16	-1.084105e-15	1.954747e-16	4.682341e-16	2.62
V12	0.124348	2.422086e-16	-6.588252e-16	2.206522e-16	-5.657963e-16	3.761033e-16	2.140589e-16	1.510045e-15	-6.266057e-17	-2.445230e-15	1.43
V13	-0.065902	-2.115458e-16	3.854521e-16	-6.883375e-16	-1.506129e-16	-9.578659e-16	-2.268061e-16	-9.892325e-17	-2.382948e-16	-2.650351e-16	-8.
V14	-0.098757	9.352582e-16	-2.541036e-16	4.271336e-16	-8.522435e-17	-3.634803e-16	3.452801e-16	-1.729462e-16	-1.131098e-16	2.343317e-16	2.62
V15	-0.183453	-3.252451e-16	2.831060e-16	1.122756e-16	-1.507718e-16	-5.132620e-16	-6.368111e-18	1.936832e-17	2.021491e-16	-1.588105e-15	7.61
V16	0.011903	6.308789e-16	4.934097e-17	1.183364e-15	-6.939204e-16	-3.517076e-16	-2.477917e-16	2.893672e-16	5.027192e-16	-3.251906e-16	-1.
V17	-0.073297	-5.011524e-16	-9.883008e-16	4.576619e-17	-4.397925e-16	1.425729e-16	3.567582e-16	1.149692e-15	-3.508777e-16	6.535992e-16	3.67
V18	0.090438	2.870125e-16	2.636654e-16	5.427965e-16	1.493667e-16	1.109525e-15	2.811474e-16	-1.116789e-16	-4.093852e-16	1.203843e-16	3.98
V19	0.028975	1.818128e-16	9.528280e-17	2.576773e-16	-2.656938e-16	-3.138234e-16	2.717167e-16	-2.874017e-16	-5.339821e-16	1.120752e-16	2.66
V20	-0.050866	1.036959e-16	-9.309954e-16	-9.429297e-16	-3.223123e-16	2.076048e-16	1.898638e-16	1.744242e-16	-1.095534e-16	-4.340941e-16	-1.
V21	0.044736	-1.755072e-16	8.444409e-17	-2.971969e-17	-9.976950e-17	-1.368701e-16	-1.575903e-16	1.938604e-16	-2.412439e-16	4.578389e-17	8.08
V22	0.144059	7.477367e-17	2.500830e-16	4.648259e-16	2.099922e-16	5.060029e-16	-3.362902e-16	-1.058131e-15	5.475559e-16	2.871855e-17	-6.
V23	0.051142	9.808705e-16	1.059562e-16	2.115206e-17	6.002528e-17	1.637596e-16	-7.232186e-17	2.327911e-16	3.897104e-16	5.929286e-16	3.80
V24	-0.016182	7.354269e-17	-8.142354e-18	-9.351637e-17	2.229738e-16	-9.286095e-16	-1.261867e-15	-2.589727e-17	-1.802967e-16	-2.346385e-16	-4.
V25	-0.233083	-9.805358e-16	-4.261894e-17	4.771164e-16	5.394585e-16	5.625102e-16	1.081933e-15	1.174169e-15	-1.390791e-16	1.099645e-15	-2.
V26	-0.041407	-8.621897e-17	2.601622e-16	6.521501e-16	-6.179751e-16	9.144690e-16	-2.378414e-16	-7.334507e-16	-1.209975e-16	-1.388725e-15	-2.
V27	-0.005135	3.208233e-17	-4.478472e-16	6.239832e-16	-6.403423e-17	4.465960e-16	-2.623818e-16	-5.886825e-16	1.733633e-16	-2.287414e-16	-3.
V28	-0.009413	9.820892e-16	-3.676415e-16	7.726948e-16	-5.863664e-17	-3.299167e-16	4.813155e-16	-6.836764e-17	-4.484325e-16	9.146779e-16	-1.
Amount	-0.010596	-2.277087e-01	-5.314089e-01	-2.108805e-01	9.873167e-02	-3.863563e-01	2.159812e-01	3.973113e-01	-1.030791e-01	-4.424560e-02	-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	1.987512e-02	-9.773269e-02	-2.

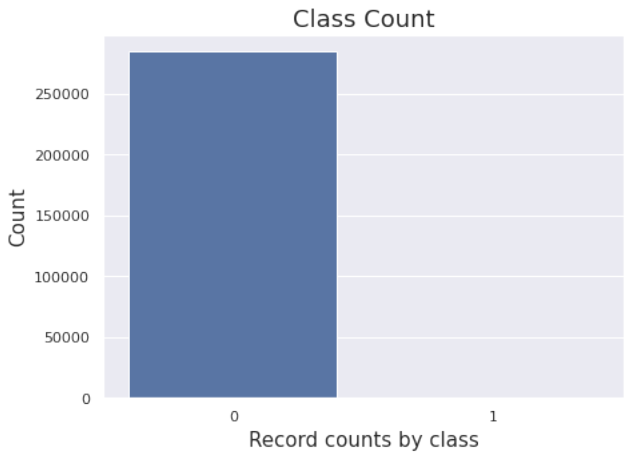
In [59]:
```
# plotting correlations on a heatmap

# figure size
plt.figure(figsize=(24,18))

# heatmap
sns.heatmap(cor, cmap="YlGnBu", annot=True)
plt.show()
```



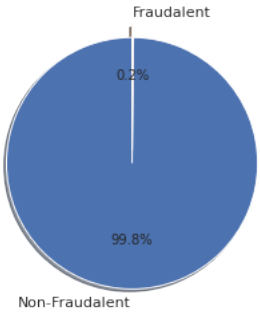
Here we will observe the distribution of our classes

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

In [61]:
```
# Create a bar plot for the number and percentage of fraudulent vs non-fraudulent transcations

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 [62]:
```
#plt.title('Dsitribution of the Fraudalent vs Non-fraudalent transaction in Percentages')
classes=df['Class'].value_counts()
normal_share=classes[0]/df['Class'].count()*100
fraud_share=classes[1]/df['Class'].count()*100


labels = 'Non-Fraudalent', 'Fraudalent'
sizes = [normal_share, fraud_share]
explode = (0, 0.1)

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



In [63]:
```
print('The percentage without churn prediction is ', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the dataset
print('The percentage with churn prediction is ', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the dataset')
print('The ratio of imbalance is', round(df['Class'].value_counts()[1]/df['Class'].value_counts()[0] * 100,2))
```

```
The percentage without churn prediction is  99.83 % of the dataset
The percentage with churn prediction is  0.17 % of the dataset
The ratio of imbalance is 0.17
```

So we have 492 fraudalent transactions out of 284807 total credit card transactions.

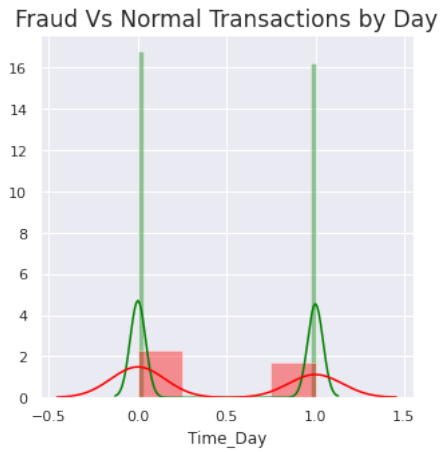
Target variable distribution shows that we are dealing with an highly imbalanced problem as there are many more genuine transactions class as compared to the fraudalent transactions. The model would achieve high accuracy as it would mostly predict majority class – transactions which are genuine in our example.
To overcome this we will use other metrics for model evaluation such as ROC-AUC , precision and recall etc

In [0]:
```
# Create a scatter plot to observe the distribution of classes with time
#As time is given in relative fashion, we will need to use pandas.Timedelta which Represents a duration, the difference betwe

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)
```

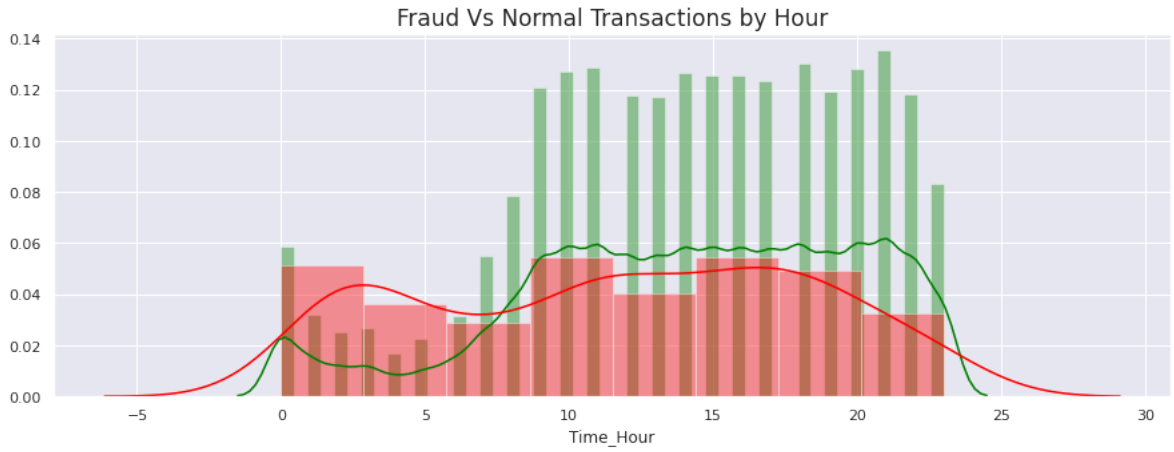
In [65]:
```
# Bivariate Analysis: Create a scatter plot to observe the distribution of classes with time

fig = plt.figure(figsize=(14, 18))
cmap = sns.color_palette('Set2')

# PLot the relation between the variables:

plt.subplot(3,1,1)
sns.scatterplot(x=df['Time'], y='Class', palette=cmap, data=df)
plt.xlabel('Time', size=18)
plt.ylabel('Class', size=18)
plt.tick_params(axis='x', labelsize=16)
plt.tick_params(axis='y', labelsize=16)
plt.title('Time vs Class Distribution', size=20, y=1.05)
```

Out[65]:
```
Text(0.5, 1.05, 'Time vs Class Distribution')
```

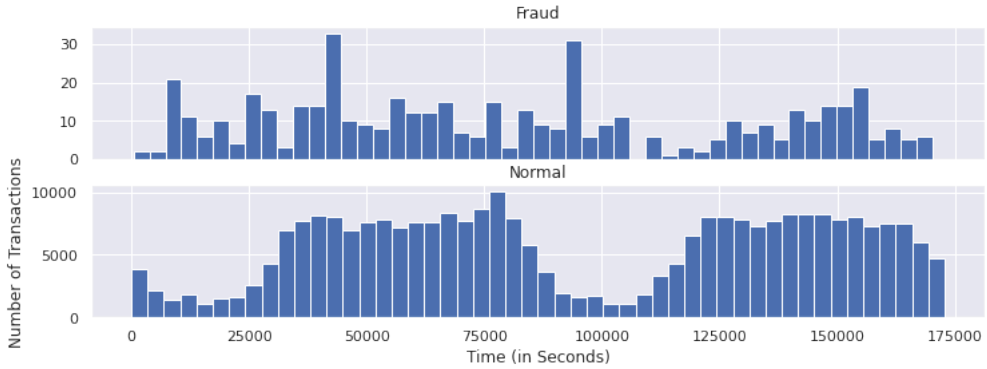
In [66]:
```
#The fraus Vs normal trasaction by day
plt.figure(figsize=(5,5))
sns.distplot(df[df['Class'] == 0]["Time_Day"], color='green')
sns.distplot(df[df['Class'] == 1]["Time_Day"], color='red')
plt.title('Fraud Vs Normal Transactions by Day', fontsize=17)
plt.show()
```



In [67]:
```
#The fraus Vs normal trasaction by hour
plt.figure(figsize=(15,5))
sns.distplot(df[df['Class'] == 0]["Time_Hour"], color='green')
sns.distplot(df[df['Class'] == 1]["Time_Hour"], color='red')
plt.title('Fraud Vs Normal Transactions by Hour', fontsize=17)
plt.show()
```



In [68]:
```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(12,4))

bins = 50

ax1.hist(df.Time[df.Class == 1], bins = bins)
ax1.set_title('Fraud')

ax2.hist(df.Time[df.Class == 0], bins = bins)
ax2.set_title('Normal')

plt.xlabel('Time (in Seconds)')
plt.ylabel('Number of Transactions')
plt.show()
```



In [69]:
```
# Create a scatter plot to observe the distribution of classes with Amount
#To clearly the data of frauds and no frauds
df_Fraud = df[df['Class'] == 1]
df_Regular = df[df['Class'] == 0]

# Fraud Transaction Amount Statistics
print(df_Fraud["Amount"].describe())
```

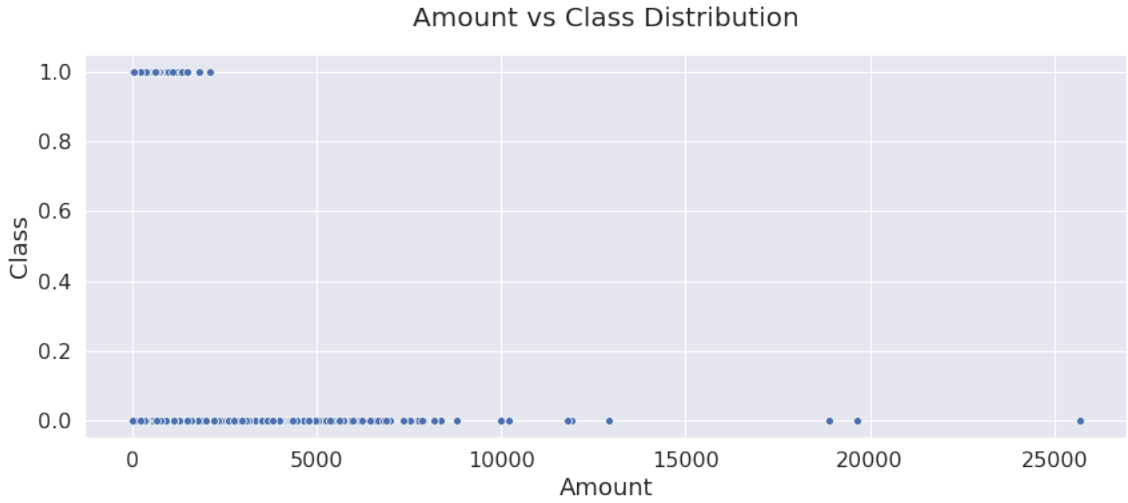
```
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

In [70]:
```
#Regular Transaction Amount Statistics
print(df_Regular["Amount"].describe())
```

```
count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
```

In [71]:
```
# Create a scatter plot to observe the distribution of classes with Amount

# Bivariate Analysis: Create a scatter plot to observe the distribution of classes with Amount


fig = plt.figure(figsize=(14, 18))
cmap = sns.color_palette('Set1')

# PLot the relation between the variables:

plt.subplot(3,1,1)
sns.scatterplot(x=df['Amount'], y='Class', palette=cmap, data=df)
plt.xlabel('Amount', size=18)
plt.ylabel('Class', size=18)
plt.tick_params(axis='x', labelsize=16)
plt.tick_params(axis='y', labelsize=16)
plt.title('Amount vs Class Distribution', size=20, y=1.05)
```

Out[71]: Text(0.5, 1.05, 'Amount vs Class Distribution')



In [72]:
```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(12,4))

bins = 30

ax1.hist(df.Amount[df.Class == 1], bins = bins)
ax1.set_title('Fraud')

ax2.hist(df.Amount[df.Class == 0], bins = bins)
ax2.set_title('Normal')

plt.xlabel('Amount ($)')
plt.ylabel('Number of Transactions')
plt.yscale('log')
plt.show()
```

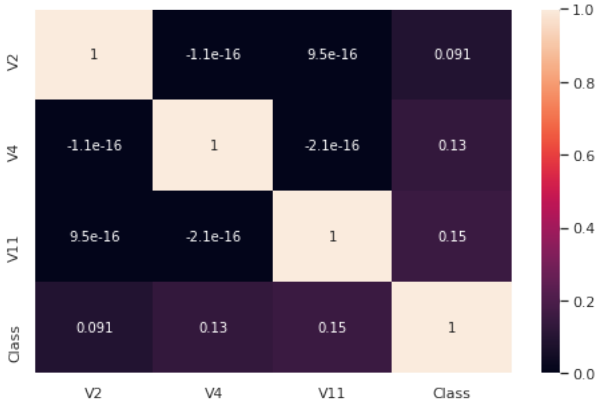
In [73]:
```
# Understanding more on the correlation in data:
print("Most important features relative to target variable Class")

corr_initial = df.corr()['Class']
# convert series to dataframe so it can be sorted
corr_initial = pd.DataFrame(corr_initial)
# correct column label from SalePrice to correlation
corr_initial.columns = ["Correlation"]
# sort correlation
corr_initial2 = corr_initial.sort_values(by=['Correlation'], ascending=False)
corr_initial2.head(5)
```

```
Most important features relative to target variable Class
```

Out[73]:

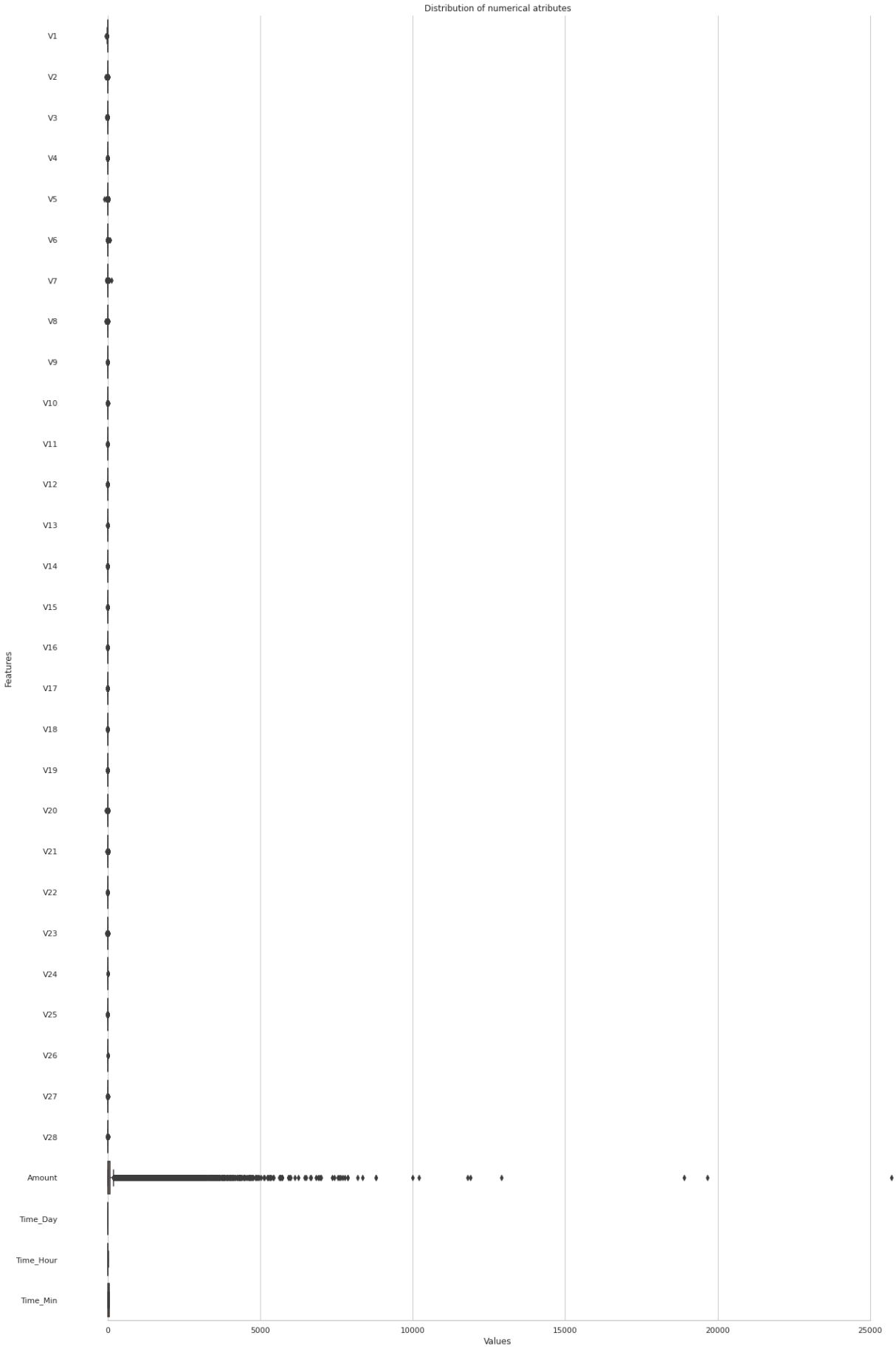
	Correlation
Class	1.000000
V11	0.154876
V4	0.133447
V2	0.091289
V21	0.040413

In [74]:
```
# Lets plot the heatmap again for relatively strong correlation (i.e. >0.09) with the target variable:

top_feature = cor.index[abs(cor['Class']>0.09)]
plt.subplots(figsize=(8, 5))
top_corr = df[top_feature].corr()
sns.heatmap(top_corr, annot=True)
plt.show()
```



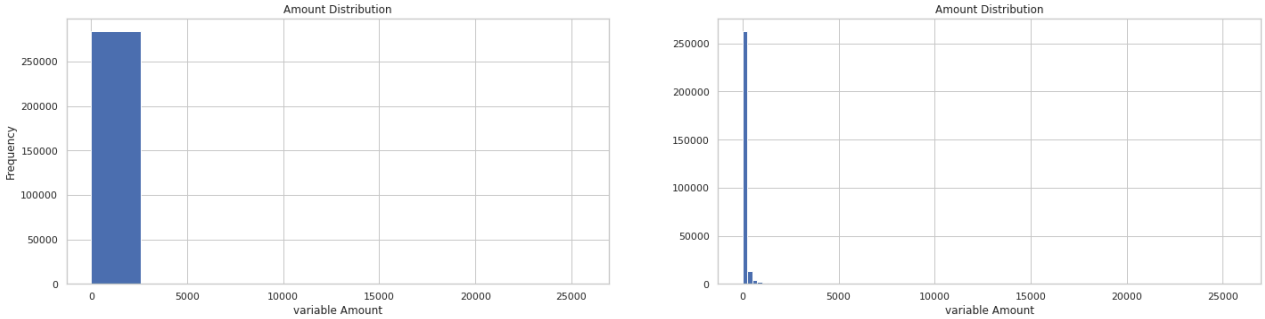
Plotting the distribution of a variable

In [75]:
```
# Boxplot to understand the distribution of numerical attributes :

# Selecting only numerical feature from the dataframe:
numeric_features = df.select_dtypes(include=[np.number]).columns.tolist()

# Excluding BINARY target feature and Time variable as its not needed for transformation :
li_not_plot = ['Class','Time']
li_transform_num_feats = [c for c in list(numeric_features) if c not in li_not_plot]

sns.set_style("whitegrid")
f, ax = plt.subplots(figsize=(22,34))
# Using log scale:
#ax.set_xscale("log")
ax = sns.boxplot(data=df[li_transform_num_feats] , orient="h", palette="Paired")
ax.set(ylabel="Features")
ax.set(xlabel="Values")
ax.set(title="Distribution of numerical atributes")
sns.despine(trim=True,left=True)
```



In [0]:
```
# Drop unnecessary columns
# As we have derived the Day/Hour/Minutes from the time column we will drop Time
df.drop('Time', axis = 1, inplace= True)
#also day/minutes might not be very useful as this is not time series data, we will keep only derived column hour
df.drop(['Time_Day', 'Time_Min'], axis = 1, inplace= True)
```

In [77]:
```
# Let's try to understand the Amount variable as it is not PCA transformed variable :

plt.figure(figsize=(24, 12))

plt.subplot(2,2,1)
plt.title('Amount Distribution')
df['Amount'].astype(int).plot.hist();
plt.xlabel("variable Amount")
#plt.ylabel("Frequency")

plt.subplot(2,2,2)
plt.title('Amount Distribution')
sns.set()
plt.xlabel("variable Amount")
plt.hist(df['Amount'],bins=100)
plt.show()
```



Splitting the data into train & test data

In [0]:
```
#Create X and y dataset for independent and dependent data
y= df['Class']
X = df.drop(['Class'], axis=1)
```

In [79]:
```
X.head()
```

Out[79]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235	0.489095	-0.143772	0
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084	0.717293	-0.165946	2
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228	0.507757	-0.287924	-0
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196	1.345852	-1.119670	0

In [0]:
```
from sklearn import model_selection

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 [81]:
```
print(np.sum(y))
print(np.sum(y_train))
print(np.sum(y_test))
```

```
492
396
96
```

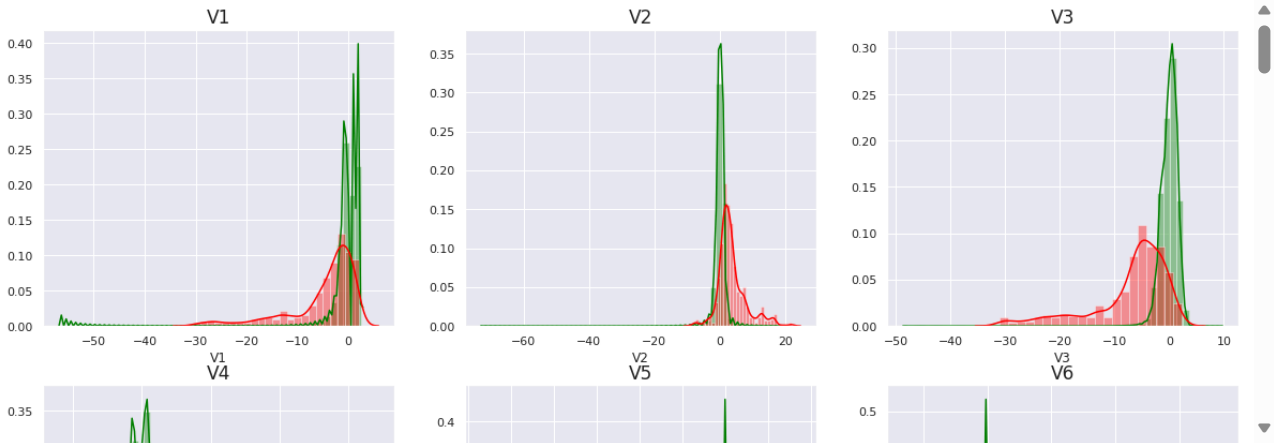
Plotting the distribution of a variable

In [0]:
```
cols = list(X.columns.values)
```

In [83]:
```
cols
```

Out[83]:
```
['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',
 'Time_Hour']
```

In [84]:
```
# 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()
```




```
#Create model functions for Logistic Regress, KNN, SVM, Decision Tree, Random Forest, XGBoost
```

In [0]:
```
#Create a dataframe to store results
df_Results = pd.DataFrame(columns=['Data_Imbalance_Handiling','Model','Accuracy','roc_value','threshold'])
```

In [0]:

```
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)
  classNames = ['Non-Fraudalent','Fraudalent']
  plt.title('Confusion Matrix - Test Data')
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  tick_marks = np.arange(len(classNames))
  plt.xticks(tick_marks, classNames, rotation=45)
  plt.yticks(tick_marks, classNames)
  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 [0]:

```
def buildAndRunLogisticModels(df_Results, DataImabalance, X_train,y_train, X_test, y_test ):

  # 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)

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

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

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


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


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

  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))

  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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'Logistic Regression with

  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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'Logistic Regression with
  return df_Results
```

In [0]:
```
def buildAndRunKNNModels(df_Results,DataImabalance, X_train,y_train, X_test, y_test ):
  #Evaluate KNN  model
  from sklearn.neighbors import KNeighborsClassifier
  from sklearn.metrics import roc_auc_score
  #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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'KNN','Accuracy': score,'

  return df_Results
```

In [0]:
```
def buildAndRunSVMModels(df_Results, DataImabalance, 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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'SVM','Accuracy': SVM_Sco

  return df_Results
```

In [0]:
```
def buildAndRunTreeModels(df_Results, DataImabalance, X_train,y_train, X_test, y_test ):
  #Evaluate Decision Tree model with 'gini' & 'entropy'
  from sklearn.tree import DecisionTreeClassifier
  from sklearn.metrics import roc_auc_score
  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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'Tree Model with {0} (

  return df_Results
```

In [0]:
```
def buildAndRunRandomForestModels(df_Results, DataImabalance, X_train,y_train, X_test, y_test ):
  #Evaluate Random Forest model

  from sklearn.ensemble import RandomForestClassifier
  from sklearn.metrics import roc_auc_score

  # 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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'Random Forest','Accuracy

  return df_Results
```

In [0]:
```
def buildAndRunXGBoostModels(df_Results, DataImabalance,X_train,y_train, X_test, y_test ):
  #Evaluate XGboost model
  from xgboost import XGBClassifier
  from sklearn.metrics import roc_auc_score
  # fit model no training data
  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({'Data_Imbalance_Handiling': DataImabalance,'Model': 'XGBoost','Accuracy': XGB_

  return df_Results
```

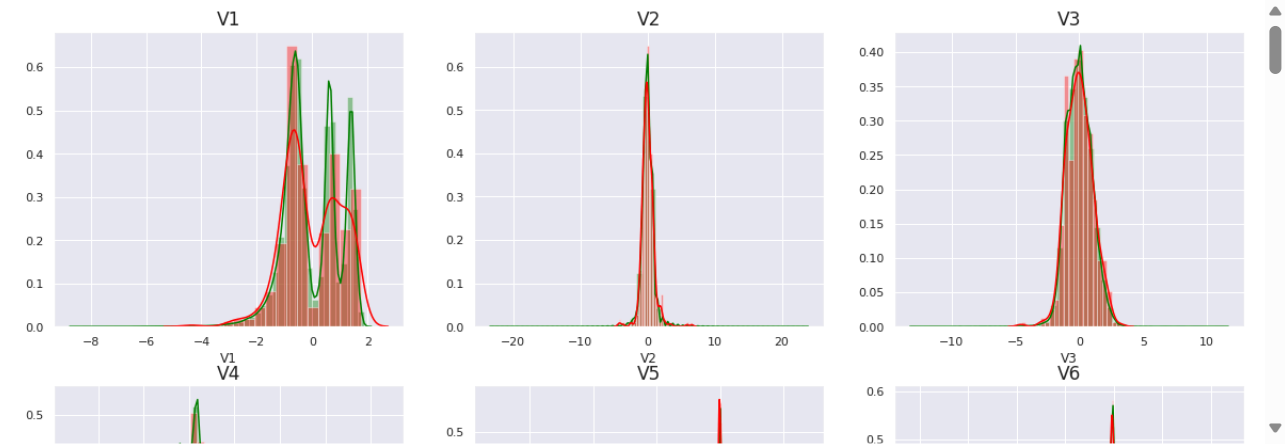
If there is skewness present in the distribution use:

- Power Transformer package present in the preprocessing library provided by sklearn to make distribution more gaussian

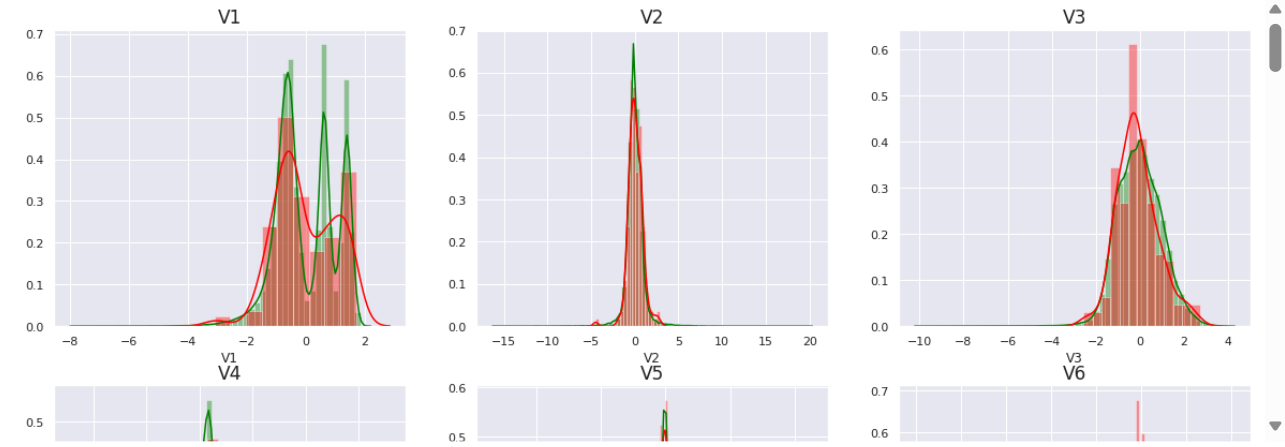
In [0]:
```
# - Apply : preprocessing.PowerTransformer(copy=False) to fit & transform the train & test data
from sklearn.preprocessing import PolynomialFeatures, PowerTransformer
pt = PowerTransformer()
pt.fit(X_train)                         ## Fit the PT on training data
X_train_pt = pt.transform(X_train)      ## Then apply on all data
X_test_pt = pt.transform(X_test)
```

In [0]:
```
#Create Dataframe
X_train_pt_df = pd.DataFrame(data=X_train_pt,   columns=cols)
X_test_pt_df = pd.DataFrame(data=X_test_pt,   columns=cols)
```

In [95]:
```
# plot the histogram of a variable from the train dataset again to see the result

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



In [96]:
```
# plot the histogram of a variable from the test dataset again to see the result
plt.figure(figsize=(20, 60))
for n, col in enumerate(cols):
  plt.subplot(10,3,n+1)
  sns.distplot(X_test_pt_df[col][normal_records], color='green')
  sns.distplot(X_test_pt_df[col][fraud_records], color='red')
  plt.title(col, fontsize=17)
plt.show()
```



Model Building

- Build different models on the imbalanced dataset and see the result

In [97]:
```
#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,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, 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,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, 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,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, 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,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,"Power Transformer",X_train_pt_df,y_train, X_test_pt_df, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.983760023970567
Max auc_roc for l1: 0.9854770239257755
Parameters for l2 regularisations
[[-0.09276541 -0.06828297 -0.28652331  0.55493742  0.02789795  0.01396825
  -0.06824953 -0.20408006 -0.18143666 -0.20371291  0.17699968 -0.50183428
  -0.11614819 -0.66830552 -0.0815942  -0.18013738 -0.08164992  0.03866812
  -0.03807332 -0.02310099  0.10465199  0.10090233 -0.03198749 -0.02939204
   0.01001787  0.00785392 -0.04474827 -0.02229527  0.02208228  0.08474082]]
[-8.07230494]
{1: array([[0.90398235, 0.90383704, 0.90383704, 0.90383704, 0.90383704,
        0.90419481, 0.96804377, 0.99255279, 0.99488378, 0.99521346,
        0.99521346, 0.99521346, 0.99521346, 0.99521346, 0.99521346,
        0.99521346, 0.99521346, 0.99521346, 0.99521346, 0.99521346],
       [0.9306611 , 0.9306611 , 0.9306611 , 0.9306611 , 0.9306611 ,
        0.93091836, 0.96502028, 0.98005776, 0.97844439, 0.97856203,
        0.97856203, 0.97856203, 0.97856203, 0.97856203, 0.97856203,
        0.97856203, 0.97856203, 0.97856203, 0.97856203, 0.97856203],
       [0.92531067, 0.92518706, 0.92518706, 0.92518706, 0.92518706,
```

In [98]:
```
df_Results.head()
```

Out[98]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998947	0.970246	0.002676
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998806	0.859015	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999052	0.864337	1.000000

Perform cross validation with RepeatedKFold

In [99]:
```
#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 ... 284803 284804 284806] TEST: [    14     16     20 ... 284795 284797 284805]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    21     23     25 ... 284767 284800 284803]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     3      4      8 ... 284788 284796 284799]
TRAIN: [     1      2      3 ... 284804 284805 284806] TEST: [     0      9     13 ... 284790 284794 284801]
TRAIN: [     0      3      4 ... 284801 284803 284805] TEST: [     1      2      5 ... 284802 284804 284806]
TRAIN: [     0      2      3 ... 284803 284804 284806] TEST: [     1     30     35 ... 284789 284801 284805]
TRAIN: [     1      4      5 ... 284804 284805 284806] TEST: [     0      2      3 ... 284782 284787 284800]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     4      8     13 ... 284796 284798 284802]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     7     14     15 ... 284794 284795 284804]
TRAIN: [     0      1      2 ... 284802 284804 284805] TEST: [     5      9     10 ... 284799 284803 284806]
TRAIN: [     0      3      5 ... 284804 284805 284806] TEST: [     1      2      4 ... 284786 284791 284792]
TRAIN: [     0      1      2 ... 284802 284805 284806] TEST: [     5     11     21 ... 284777 284803 284804]
TRAIN: [     0      1      2 ... 284803 284804 284806] TEST: [     6      9     12 ... 284799 284800 284805]
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     3      7     14 ... 284796 284802 284806]
TRAIN: [     1      2      3 ... 284804 284805 284806] TEST: [     0      8     10 ... 284781 284797 284801]
TRAIN: [     1      2      3 ... 284802 284804 284806] TEST: [     0      6      8 ... 284800 284803 284805]
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     4     11     12 ... 284784 284801 284806]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     3     18     26 ... 284792 284793 284795]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [    13     14     19 ... 284785 284802 284804]
TRAIN: [     0      3      4 ... 284804 284805 284806] TEST: [     1      2      5 ... 284790 284796 284798]
TRAIN: [     0      1      3 ... 284803 284804 284805] TEST: [     2     10     12 ... 284799 284800 284806]
TRAIN: [     0      2      3 ... 284804 284805 284806] TEST: [     1      9     13 ... 284788 284798 284802]
TRAIN: [     1      2      3 ... 284804 284805 284806] TEST: [     0     15     20 ... 284793 284794 284803]
TRAIN: [     0      1      2 ... 284803 284804 284806] TEST: [     3      4      5 ... 284796 284797 284805]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     6      8     11 ... 284795 284801 284804]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     5      7     14 ... 284794 284796 284798]
TRAIN: [     1      3      4 ... 284799 284802 284805] TEST: [     0      2     10 ... 284803 284804 284806]
TRAIN: [     0      1      2 ... 284803 284804 284806] TEST: [     3      4      6 ... 284795 284799 284805]
TRAIN: [     0      2      3 ... 284804 284805 284806] TEST: [     1     17     22 ... 284784 284797 284802]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     9     19     21 ... 284780 284791 284793]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     4     10     27 ... 284790 284793 284801]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     6      8     12 ... 284782 284789 284792]
TRAIN: [     0      1      2 ... 284802 284804 284805] TEST: [     3      7     16 ... 284800 284803 284806]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     9     11     13 ... 284794 284802 284804]
TRAIN: [     3      4      6 ... 284803 284804 284806] TEST: [     0      1      2 ... 284796 284797 284805]
TRAIN: [     1      2      3 ... 284803 284804 284806] TEST: [     0     12     13 ... 284768 284789 284805]
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     4      8      9 ... 284795 284798 284806]
TRAIN: [     0      3      4 ... 284804 284805 284806] TEST: [     1      2     17 ... 284799 284800 284803]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [     6      7     18 ... 284797 284801 284802]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3      5     10 ... 284788 284791 284804]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [    10     15     17 ... 284792 284797 284801]
TRAIN: [     0      2      3 ... 284804 284805 284806] TEST: [     1     14     29 ... 284780 284781 284796]
TRAIN: [     0      1      3 ... 284802 284804 284805] TEST: [     2      5      8 ... 284800 284803 284806]
TRAIN: [     1      2      3 ... 284803 284804 284806] TEST: [     0      6      9 ... 284798 284799 284805]
TRAIN: [     0      1      2 ... 284803 284805 284806] TEST: [     3      4      7 ... 284793 284802 284804]
TRAIN: [     0      1      2 ... 284803 284804 284805] TEST: [     7      8     19 ... 284795 284796 284806]
TRAIN: [     0      1      2 ... 284802 284804 284806] TEST: [    13     17     28 ... 284801 284803 284805]
TRAIN: [     0      2      3 ... 284803 284805 284806] TEST: [     1      9     14 ... 284798 284802 284804]
TRAIN: [     1      2      5 ... 284804 284805 284806] TEST: [     0      3      4 ... 284790 284794 284797]
TRAIN: [     0      1      3 ... 284804 284805 284806] TEST: [     2      5      6 ... 284786 284788 284793]
```

In [100]:
```
#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_train_cv,y_train_cv, X_test_cv, 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,"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('-'*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,"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('-'*80 )
#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_t
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#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_c
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#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))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.9823679464176831
Max auc_roc for l1: 0.9600809175324396
Parameters for l2 regularisations
[[ 7.30096846e-03  4.39054392e-02 -8.72384853e-02  2.48274466e-01
   8.16770838e-02 -4.19062655e-02 -3.87041341e-02 -1.19258883e-01
  -8.65625031e-02 -1.65607922e-01  1.35357109e-01 -2.03120494e-01
  -3.08378489e-02 -3.69908918e-01 -1.09913944e-02 -9.68525219e-02
  -7.98382418e-02 -1.48886429e-03  8.76947314e-03 -7.87305770e-03
   3.67118920e-02  2.76845651e-02 -1.15533969e-04 -1.00789740e-02
  -1.17617950e-02  9.05954632e-03 -1.01752068e-02 -4.58286372e-03
   2.96453431e-04  6.56025172e-03]]
[-7.55834996]
{1: array([[0.65496784, 0.65628466, 0.67098181, 0.78804377, 0.92948127,
        0.97865512, 0.99147781, 0.99253096, 0.99391445, 0.9929901 ,
        0.9929901 , 0.9929901 , 0.9929901 , 0.9929901 , 0.9929901 ,
        0.9929901 , 0.9929901 , 0.9929901 , 0.9929901 , 0.9929901 ],
       [0.52570898, 0.52780612, 0.55312518, 0.74757549, 0.94855858,
        0.98984071, 0.99374768, 0.99032249, 0.98494479, 0.98764585,
```

In [101]:
```
df_Results
```

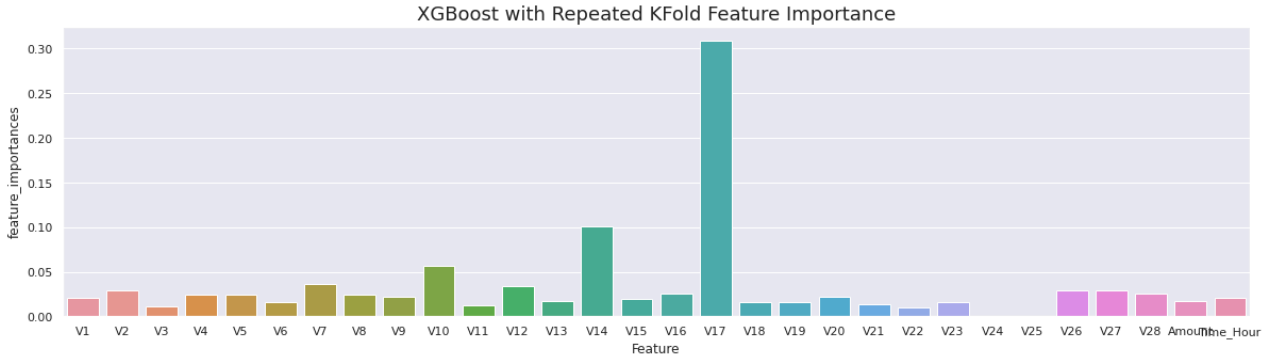
Out[101]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998947	0.970246	0.002676
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998806	0.859015	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999052	0.864337	1.000000
5	Power Transformer	Random Forest	0.999350	0.940564	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891322	0.001064
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998876	0.984196	0.002033
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998894	0.909238	0.022686
10	RepeatedKFold Cross Validation	KNN	0.999070	0.860102	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999034	0.900611	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999017	0.900602	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999526	0.949019	0.020000
14	RepeatedKFold Cross Validation	XGBoost	0.999491	0.984352	0.001197
15	RepeatedKFold Cross Validation	SVM	0.998174	0.514644	0.002859

It seems XGBOost with Repeated KFold cross validation has provided us wih best results with ROC_Value of 0.984352

In [0]:
```
#Evaluate XGboost model
from xgboost import XGBClassifier
# fit model no training data
XGBmodel = XGBClassifier(random_state=42)
XGBmodel.fit(X_train_cv,y_train_cv)

coefficients = pd.concat([pd.DataFrame(X.columns),pd.DataFrame(np.transpose(XGBmodel.feature_importances_))], axis = 1)
coefficients.columns = ['Feature','feature_importances']
```

In [103]:
```
plt.figure(figsize=(20,5))
sns.barplot(x='Feature', y='feature_importances', data=coefficients)
plt.title("XGBoost with Repeated KFold Feature Importance", fontsize=18)

plt.show()
```



Perform cross validation with StratifiedKFold

In [0]:
```
#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 ... 57017 57018 57019]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [ 30473  30496  31002 ... 113964 113965 113966]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [ 81609  82400  83053 ... 170946 170947 170948]
TRAIN: [     0      1      2 ... 284804 284805 284806] TEST: [150654 150660 150661 ... 227866 227867 227868]
TRAIN: [     0      1      2 ... 227866 227867 227868] TEST: [212516 212644 213092 ... 284804 284805 284806]
```

Similarly explore other algorithms by building models like:

- KNN
- SVM
- Decision Tree
- Random Forest
- XGBoost

In [0]:
```
#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_train_SKF_cv,y_train_SKF_cv, X_test_S
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,"StratifiedKFold Cross Validation",X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv
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,"StratifiedKFold Cross Validation",X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv
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,"StratifiedKFold Cross Validation",X_train_SKF_cv,y_train_SKF_cv, X_tes
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#Run XGBoost Modela
print("XGBoost Model")
start_time = time.time()
df_Results = buildAndRunXGBoostModels(df_Results,"StratifiedKFold Cross Validation",X_train_SKF_cv,y_train_SKF_cv, X_test_SKF
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#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_SKF_cv,y_train_SKF_cv, X_test_SKF_cv
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.982552383813862
Max auc_roc for l1: 0.9532044417995286
Parameters for l2 regularisations
[[ 6.85139233e-03  6.17754487e-02 -6.71812892e-02  2.18613707e-01
   5.41818363e-02 -3.20377133e-02 -2.28974175e-02 -1.14988071e-01
  -6.93795526e-02 -1.63673872e-01  1.28016335e-01 -1.92133511e-01
  -3.69083488e-02 -3.60608541e-01  3.48475228e-03 -8.79624763e-02
  -7.27253829e-02 -7.05776486e-04  3.90161817e-03 -2.80743299e-02
   3.81909632e-02  2.33588614e-02 -1.16098485e-02 -9.98603232e-03
  -6.55263654e-03  6.29741877e-03 -1.01442016e-02 -1.67911649e-03
   4.00150036e-04  3.10464020e-04]]
[-7.42734858]
{1: array([[0.69244522, 0.69421053, 0.71448798, 0.83769702, 0.95579612,
        0.97136042, 0.9891556 , 0.99473675, 0.99129968, 0.99109226,
        0.99109226, 0.99109226, 0.99109226, 0.99109226, 0.99109226,
        0.99109226, 0.99109226, 0.99109226, 0.99109226, 0.99109226],
       [0.60698465, 0.60863434, 0.6291165 , 0.76881516, 0.90899478,
        0.94877915, 0.9603719 , 0.98184219, 0.98547281, 0.98327947,
```

In [0]:
```
df_Results
```

Out[52]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998947	0.970246	0.002676
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998806	0.859015	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999052	0.864337	1.000000
5	Power Transformer	Random Forest	0.999350	0.940564	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891341	0.000934
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.999210	0.989580	0.001313
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.999228	0.907356	0.065160
10	RepeatedKFold Cross Validation	KNN	0.999263	0.884847	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999140	0.896279	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999122	0.873317	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999526	0.946466	0.010000
14	RepeatedKFold Cross Validation	XGBoost	0.999544	0.991970	0.000733
15	RepeatedKFold Cross Validation	SVM	0.997858	0.680293	0.002100
16	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983339	0.001569
17	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998631	0.923413	0.004244
18	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
19	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998841	0.826249	1.000000
20	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999017	0.821244	1.000000
21	StratifiedKFold Cross Validation	Random Forest	0.999438	0.946472	0.010000
22	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
23	StratifiedKFold Cross Validation	SVM	0.998280	0.401770	0.002843

****As the results show Logistic Regression with L2 Regularisation for StratifiedFold 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 [0]:

```
  # 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)

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


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


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


#Find predicted probabilities
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)


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))
```

```
Max auc_roc for l2: 0.9860157961461928
Parameters for l2 regularisations
[[ 2.05494512e-02  3.27890625e-02 -8.96514836e-02  2.38302750e-01
   8.29640042e-02 -4.90363237e-02 -3.31411706e-02 -1.08612881e-01
  -9.26053454e-02 -1.75808446e-01  1.35492913e-01 -2.08141147e-01
  -4.34399635e-02 -3.78889848e-01 -1.11305988e-02 -1.04749939e-01
  -1.14544376e-01 -8.46144242e-03  1.49894309e-02 -5.86707388e-03
   3.98051253e-02  2.46455931e-02  1.95896795e-03 -1.57936078e-02
  -7.89511943e-03  5.19255364e-03 -3.25772934e-03 -2.37002046e-04
   3.01640888e-04  4.53251460e-03]]
[-7.50114736]
{1: array([[0.582166  , 0.58434192, 0.6102234 , 0.77826009, 0.94270322,
        0.98595298, 0.99602302, 0.9940266 , 0.9941426 , 0.99499001,
        0.99342462, 0.99342462, 0.99342462, 0.99342462, 0.99342462,
        0.99342462, 0.99342462, 0.99342462, 0.99342462, 0.99342462],
       [0.56080922, 0.56256092, 0.57926348, 0.72051132, 0.88967906,
        0.93826774, 0.9754206 , 0.97876629, 0.97975913, 0.98089289,
        0.98089289, 0.98089289, 0.98089289, 0.98089289, 0.98089289,
        0.98089289, 0.98089289, 0.98089289, 0.98089289, 0.98089289],
       [0.5401307 , 0.54247929, 0.56922576, 0.75371242, 0.93262702,
        0.98512147, 0.99661995, 0.99703611, 0.9937055 , 0.98232513,
        0.9797664 , 0.9797664 , 0.9797664 , 0.9797664 , 0.9797664 ,
        0.9797664 , 0.9797664 , 0.9797664 , 0.9797664 , 0.9797664 ],
       [0.56287574, 0.56501433, 0.59079147, 0.76878989, 0.92492242,
        0.96178458, 0.98736331, 0.98449166, 0.98138238, 0.97157814,
        0.97157814, 0.97157814, 0.97157814, 0.97157814, 0.97157814,
        0.97157814, 0.97157814, 0.97157814, 0.97157814, 0.97157814],
       [0.57606537, 0.57861898, 0.60586271, 0.78864886, 0.93777312,
        0.96845846, 0.98689793, 0.99082986, 0.98956008, 0.98803297,
        0.9845427 , 0.9845427 , 0.9845427 , 0.9845427 , 0.9845427 ,
        0.9845427 , 0.9845427 , 0.9845427 , 0.9845427 , 0.9845427 ],
       [0.56499936, 0.56692437, 0.58791931, 0.7349557 , 0.89257268,
        0.93400274, 0.97421774, 0.97846134, 0.97306703, 0.95797152,
        0.95797152, 0.95797152, 0.95797152, 0.95797152, 0.95797152,
        0.95797152, 0.95797152, 0.95797152, 0.95797152, 0.95797152],
       [0.70428794, 0.70626649, 0.72861964, 0.85880012, 0.96370361,
        0.9789362 , 0.99335869, 0.99669803, 0.99503276, 0.99251341,
        0.99251341, 0.99251341, 0.99251341, 0.99251341, 0.99251341,
        0.99251341, 0.99251341, 0.99251341, 0.99251341, 0.99251341],
       [0.58738842, 0.58894134, 0.60804454, 0.75409172, 0.9139239 ,
        0.94380251, 0.97609189, 0.98190079, 0.97498334, 0.95869963,
        0.95869963, 0.95869963, 0.95869963, 0.95869963, 0.95869963,
        0.95869963, 0.95869963, 0.95869963, 0.95869963, 0.95869963],
       [0.57477819, 0.5764715 , 0.59684348, 0.74503359, 0.89752669,
        0.94644418, 0.96724364, 0.97202308, 0.96507554, 0.95909501,
        0.95909501, 0.95909501, 0.95909501, 0.95909501, 0.95909501,
        0.95909501, 0.95909501, 0.95909501, 0.95909501, 0.95909501],
       [0.63001167, 0.63177706, 0.65226395, 0.79417736, 0.93932845,
        0.96440469, 0.98043301, 0.98592421, 0.98611924, 0.98401903,
        0.98401903, 0.98401903, 0.98401903, 0.98401903, 0.98401903,
        0.98401903, 0.98401903, 0.98401903, 0.98401903, 0.98401903]])}
Accuarcy of Logistic model with l2 regularisation : 0.9988764439450862
l2 roc_value: 0.9754199199287213
l2 threshold: 0.0013463004464288415
```

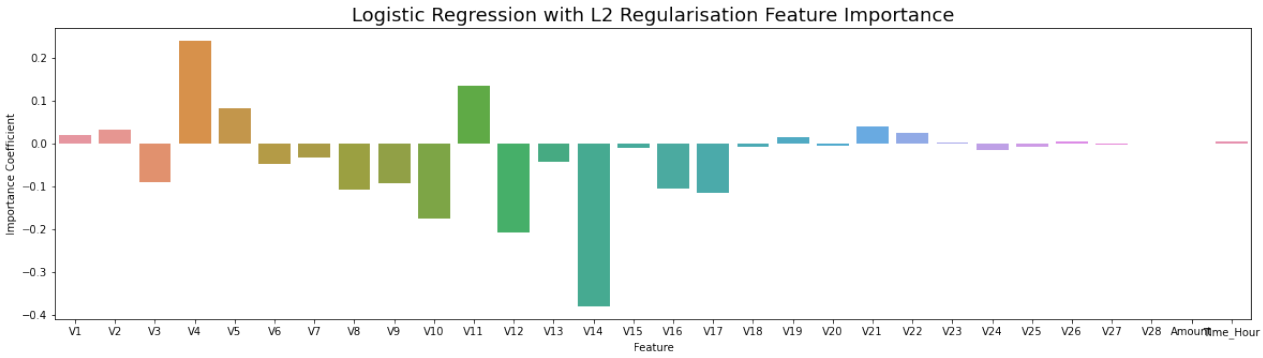
In [0]:
```
searchCV_l2.coef_
```

Out[55]:
```
array([[ 2.05494512e-02,  3.27890625e-02, -8.96514836e-02,
         2.38302750e-01,  8.29640042e-02, -4.90363237e-02,
        -3.31411706e-02, -1.08612881e-01, -9.26053454e-02,
        -1.75808446e-01,  1.35492913e-01, -2.08141147e-01,
        -4.34399635e-02, -3.78889848e-01, -1.11305988e-02,
        -1.04749939e-01, -1.14544376e-01, -8.46144242e-03,
         1.49894309e-02, -5.86707388e-03,  3.98051253e-02,
         2.46455931e-02,  1.95896795e-03, -1.57936078e-02,
        -7.89511943e-03,  5.19255364e-03, -3.25772934e-03,
        -2.37002046e-04,  3.01640888e-04,  4.53251460e-03]])
```

In [0]:
```
coefficients = pd.concat([pd.DataFrame(X.columns),pd.DataFrame(np.transpose(searchCV_l2.coef_))], axis = 1)
coefficients.columns = ['Feature','Importance Coefficient']
```

In [0]:
```
coefficients
```

In [0]:
```
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()
```



****Its is evident that V4, V11, v5 has + ve imporatnce whereas V14, V12, V10 seems to have -ve impact on the predictaions****

```
#As the models Oversampling data, take significantly longer time to run.
We will try with undersampling methods
```

In [0]:
```
# Undersampling

from imblearn.under_sampling import RandomUnderSampler
#Define Oversampler
RUS = RandomUnderSampler(sampling_strategy=0.5)
# fit and apply the transform
X_Under, y_Under = RUS.fit_resample(X_train, y_train)
#Create Dataframe
X_Under = pd.DataFrame(data=X_Under,   columns=cols)
```

In [47]:
```
#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,"Random Undersampling", X_Under, y_Under , 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,"Random Undersampling",X_Under, y_Under , 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, "Random Undersampling",X_Under, y_Under , 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, "Random Undersampling",X_Under, y_Under , 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, "Random Undersampling",X_Under, y_Under , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results, "Random Undersampling",X_Under, y_Under , X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.9811231353749233
Max auc_roc for l1: 0.5
Parameters for l2 regularisations
[[-0.04909258  0.02137459 -0.1450031   0.38611892  0.05326705 -0.10295461
  -0.06266579 -0.04305795 -0.10154318 -0.21250084  0.15336315 -0.28315858
  -0.12352161 -0.41942831 -0.01982848 -0.09155073 -0.01971507  0.03501985
   0.00504673 -0.03573556  0.06653707  0.01519606  0.02317174 -0.02672027
   0.00214569 -0.01066237  0.01381849  0.02922295  0.00158951 -0.05258998]]
[-2.23378235]
{1: array([[0.5599359 , 0.56025641, 0.56634615, 0.60512821, 0.76442308,
        0.90608974, 0.96057692, 0.97083333, 0.98237179, 0.98269231,
        0.98269231, 0.98269231, 0.98269231, 0.98269231, 0.98269231,
        0.98269231, 0.98269231, 0.98269231, 0.98269231, 0.98269231],
       [0.47108512, 0.47173489, 0.48050682, 0.53768681, 0.71539961,
        0.86419753, 0.9217024 , 0.94769331, 0.95224172, 0.95711501,
        0.95711501, 0.95711501, 0.95711501, 0.95711501, 0.95711501,
        0.95711501, 0.95711501, 0.95711501, 0.95711501, 0.95711501],
       [0.50494397, 0.50494397, 0.51285432, 0.57218194, 0.78707976,
```

In [0]:
```
df_Results
```

Out[75]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998947	0.970246	0.002676
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998806	0.859015	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999052	0.864337	1.000000
5	Power Transformer	Random Forest	0.999350	0.940564	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891341	0.000934
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.999210	0.989580	0.001313
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.999228	0.907356	0.065160
10	RepeatedKFold Cross Validation	KNN	0.999263	0.884847	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999140	0.896279	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999122	0.873317	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999526	0.946466	0.010000
14	RepeatedKFold Cross Validation	XGBoost	0.999544	0.991970	0.000733
15	RepeatedKFold Cross Validation	SVM	0.997858	0.680293	0.002100
16	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983339	0.001569
17	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998631	0.923413	0.004244
18	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
19	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998841	0.826249	1.000000
20	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999017	0.821244	1.000000
21	StratifiedKFold Cross Validation	Random Forest	0.999438	0.946472	0.010000
22	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
23	StratifiedKFold Cross Validation	SVM	0.998280	0.401770	0.002843
24	Random Undersampling	Logistic Regression with L2 Regularisation	0.998227	0.970043	0.153747
25	Random Undersampling	Logistic Regression with L1 Regularisation	0.998315	0.500000	1.500000
26	Random Undersampling	KNN	0.978336	0.941358	0.400000
27	Random Undersampling	Tree Model with gini criteria	0.941909	0.882513	1.000000
28	Random Undersampling	Tree Model with entropy criteria	0.938240	0.901474	1.000000
29	Random Undersampling	Random Forest	0.992521	0.973638	0.270000
30	Random Undersampling	XGBoost	0.984955	0.982122	0.153101
31	Random Undersampling	SVM	0.765089	0.310644	0.258133

****It seems Undersampling has impoved the XGBoost Results****

Model building with balancing Classes

Perform class balancing with :
- Random Oversampling
- SMOTE
- ADASYN

Oversampling with RandomOverSampler and StratifiedKFold Cross Validation

****We will use Random Oversampling method to handle the class imbalance****

1. First we will display class distibution with and without the Random Oversampling.

2. Then We will use the oversampled with StratifiedKFold cross validation method to genearte Train And test datasets.

Once we have train and test dataset we will feed the data to below models:
1. Logistic Regression with L2 Regularisation
2. Logistic Regression with L1 Regularisation
3. KNN
4. Decision tree model with Gini criteria
5. Decision tree model with Entropy criteria
6. Random Forest
7. XGBoost

3. We did try SVM (support vector Machine) model , but due to extensive processive power requirement we avoided useing the model.

4. Once we get results for above model, we will compare the results and select model which provided best results for the Random oversampling techinique

In [0]:
```
"""
from imblearn.over_sampling import RandomOverSampler
#Define Oversampler
ROS = RandomOverSampler(sampling_strategy=0.5)
# fit and apply the transform
X_over, y_over = ROS.fit_resample(X_train, y_train)
"""
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/externals/six.py:31: FutureWarning: The module is deprecated in version 0.21
and will be removed in version 0.23 since we've dropped support for Python 2.7. Please rely on the official version of six
(https://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:144: FutureWarning: The sklearn.neighbors.base module i
s  deprecated in version 0.22 and will be removed in version 0.24. The corresponding classes / functions should instead be
imported from sklearn.neighbors. Anything that cannot be imported from sklearn.neighbors is now part of the private API.
  warnings.warn(message, FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
```

In [0]:
```
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_sample(X_train, y_train)

#Create Dataframe for X_over
X_over = pd.DataFrame(data=X_over,   columns=cols)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/externals/six.py:31: FutureWarning: The module is deprecated in version 0.21
and will be removed in version 0.23 since we've dropped support for Python 2.7. Please rely on the official version of six
(https://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:144: FutureWarning: The sklearn.neighbors.base module i
s  deprecated in version 0.22 and will be removed in version 0.24. The corresponding classes / functions should instead be
imported from sklearn.neighbors. Anything that cannot be imported from sklearn.neighbors is now part of the private API.
  warnings.warn(message, FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecate
d; safe_indexing is deprecated in version 0.22 and will be removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
```

In [0]:
```
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('-'*80 )
#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('-'*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_over, y_over, 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_over, y_over, 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_over, y_over, X_test, y_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )
#Run SVM Model with Sigmoid Kernel
#print("SVM Model with Sigmoid Kernel")
#start_time = time.time()
#df_Results = buildAndRunSVMModels(df_Results , Data_Imbalance_Handiling,X_over, y_over, X_test, y_test)
#print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.9865025591927999
Max auc_roc for l1: 0.5
Parameters for l2 regularisations
[[ 0.36886344  0.38435425  0.28081628  0.84058165  0.34636546 -0.48593156
  -0.33886921 -0.33046818 -0.22278712 -0.66609549  0.29429241 -0.90549267
  -0.31357142 -0.90089451  0.01401429 -0.5415035  -0.30379454 -0.11348605
   0.11850648 -0.70473518  0.0672293   0.51170164  0.12992618 -0.41528494
   0.20106506 -0.09681546 -0.32603165  0.16442407  0.00549619  0.03956361]]
[-5.12758767]
{1: array([[0.63316119, 0.76122851, 0.90738112, 0.9566734 , 0.97744747,
        0.98500112, 0.98648266, 0.98681054, 0.9870233 , 0.98710244,
        0.98710244, 0.98710244, 0.98711965, 0.98711965, 0.98711965,
        0.98711965, 0.98711965, 0.98711965, 0.98711965, 0.98711965],
       [0.63241139, 0.75999538, 0.90711032, 0.9571945 , 0.97724342,
        0.98428955, 0.98538597, 0.9857937 , 0.98599362, 0.98604023,
        0.98604023, 0.98604023, 0.98604023, 0.98604023, 0.98604023,
        0.98604023, 0.98604023, 0.98604023, 0.98604023, 0.98604023],
       [0.63177623, 0.75996798, 0.90777497, 0.95707584, 0.97753824,
```

In [0]:
```
df_Results
```

Out[279]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998982	0.967170	0.003298
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998947	0.869484	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999070	0.869545	1.000000
5	Power Transformer	Random Forest	0.999350	0.930340	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891340	0.001346
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.999034	0.997121	0.001853
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.999052	0.949325	0.027425
10	RepeatedKFold Cross Validation	KNN	0.999333	0.901774	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999192	0.896839	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999087	0.896786	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999631	0.979573	0.020000
14	RepeatedKFold Cross Validation	XGBoost	0.999614	0.997407	0.001658
15	RepeatedKFold Cross Validation	SVM	0.998315	0.469528	0.004318
16	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983339	0.001569
17	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998631	0.933642	0.005137
18	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
19	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998982	0.831413	1.000000
20	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999052	0.816168	1.000000
21	StratifiedKFold Cross Validation	Random Forest	0.999438	0.940527	0.010000
22	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
23	StratifiedKFold Cross Validation	SVM	0.998280	0.401770	0.002409
24	Random Undersampling	Logistic Regression with L2 Regularisation	0.984639	0.963450	0.246874
25	Random Undersampling	Logistic Regression with L1 Regularisation	0.998315	0.500000	1.500000
26	Random Undersampling	KNN	0.975931	0.942016	0.400000
27	Random Undersampling	Tree Model with gini criteria	0.937169	0.890538	1.000000
28	Random Undersampling	Tree Model with entropy criteria	0.954303	0.888721	1.000000
29	Random Undersampling	Random Forest	0.992047	0.976800	0.270000
30	Random Undersampling	XGBoost	0.986535	0.981135	0.108230
31	Random Undersampling	SVM	0.640129	0.551141	0.331891
32	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.986359	0.980812	0.425031
33	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
34	Random Oversampling with StratifiedKFold CV	KNN	0.998069	0.805604	0.200000
35	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.999017	0.816151	1.000000
36	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.999122	0.831483	1.000000
37	Random Oversampling with StratifiedKFold CV	Random Forest	0.999456	0.956230	0.010000
38	Random Oversampling with StratifiedKFold CV	XGBoost	0.996436	0.982501	0.233910

Results for Random Oversampling:

****Random Oversampling seems to have +ve change in prediction for XGBoost****
Looking at Accuracy and ROC value we have XGBoost which has provided best results for SMOTE oversampling technique

Similarly explore other algorithms on balanced dataset by building models like:
- KNN
- SVM
- Decision Tree
- Random Forest
- XGBoost

Oversampling with SMOTE Oversampling

****We will use SMOTE Oversampling method to handle the class imbalance****

1. First we will display class distibution with and without the SMOTE Oversampling.

2. Then We will use the oversampled with StratifiedKFold cross validation method to genearte Train And test datasets.

Once we have train and test dataset we will feed the data to below models:
1. Logistic Regression with L2 Regularisation
2. Logistic Regression with L1 Regularisation
3. KNN
4. Decision tree model with Gini criteria

5. Decision tree model with Entropy criteria
6. Random Forest
7. XGBoost

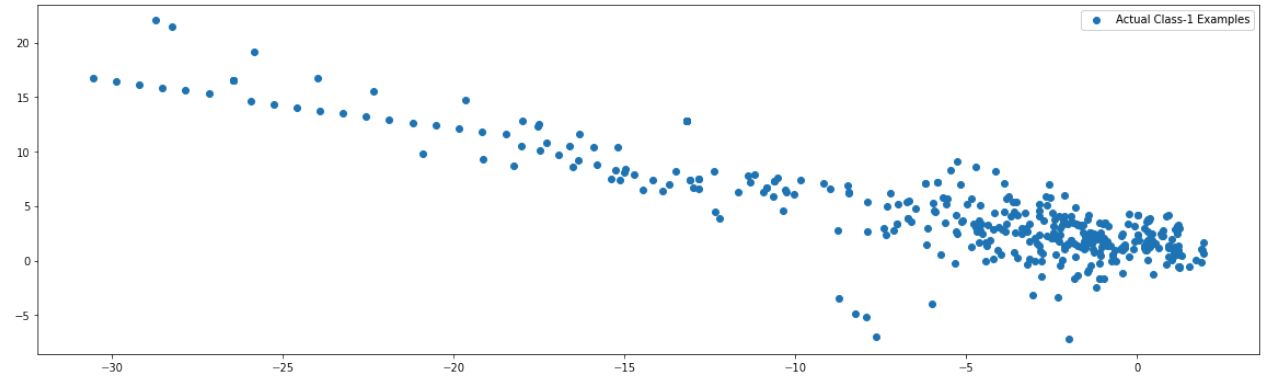
3. We did try SVM (support vector Machine) model , but due to extensive processive power requirement we avoided useing the model.

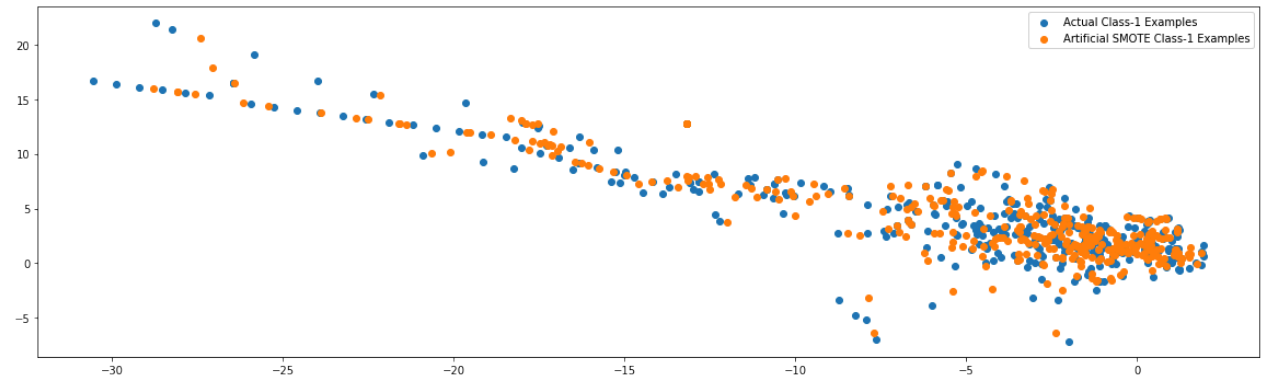
4. Once we get results for above model, we will compare the results and select model which provided best results for the SMOTE oversampling techinique

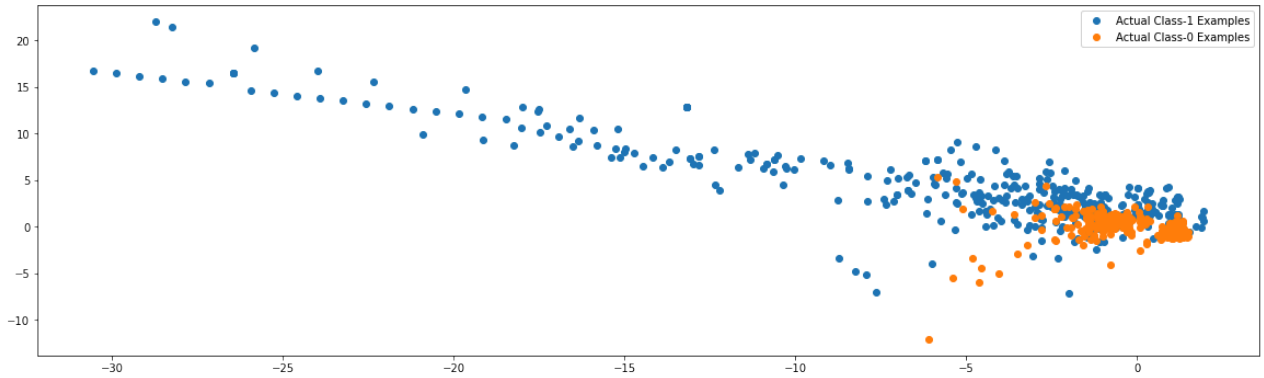
Print the class distribution after applying SMOTE

In [0]:

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

from imblearn import over_sampling
SMOTE = over_sampling.SMOTE(random_state=0)

X_train_Smote, y_train_Smote= SMOTE.fit_sample(X_train, y_train)

#Create dataframe
#X_train_Smote = pd.DataFrame(data=X_train_Smote,   columns=cols)
# Artificial minority samples and corresponding minority labels from SMOTE are appended
# below X_train and y_train respectively
# So to exclusively get the artificial minority samples from SMOTE, we do
X_train_smote_1 = X_train_Smote[X_train.shape[0]:]

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


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

plt.subplot(3, 1, 1)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.legend()

plt.subplot(3, 1, 2)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.scatter(X_train_smote_1[:X_train_1.shape[0], 0], X_train_smote_1[:X_train_1.shape[0], 1],
            label='Artificial SMOTE Class-1 Examples')
plt.legend()

plt.subplot(3, 1, 3)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.scatter(X_train_0[:X_train_1.shape[0], 0], X_train_0[:X_train_1.shape[0], 1], label='Actual Class-0 Examples')
plt.legend()


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







In [0]:
```
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_sample(X_train, y_train)

#Create Dataframe for X_over
X_train_Smote = pd.DataFrame(data=X_train_Smote,   columns=cols)
```

In [0]:
```
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_Smote, 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_te
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 )
#Run SVM Model with Sigmoid Kernel
#print("SVM Model with Sigmoid Kernel")
#start_time = time.time()
#df_Results = buildAndRunSVMModels(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))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.9931097764111401
Max auc_roc for l1: 0.5
Parameters for l2 regularisations
[[ 0.7732628   0.71963287  0.67449012  0.93817716  0.52161111 -0.92453533
  -0.69757323 -0.50072878 -0.39639681 -0.91309049  0.62482619 -1.29304859
  -0.21550507 -1.5144778   0.10885543 -0.77976472 -0.83913558 -0.42652265
   0.22281993 -1.08033762 -0.01509549  0.5741215   0.4155427  -0.89802921
   0.50130269  0.03836659 -0.19066733  0.8462861   0.00910223  0.03330456]]
[-5.5895278]
{1: array([[0.64885856, 0.7848513 , 0.9172895 , 0.96622762, 0.9856963 ,
        0.98987474, 0.99085632, 0.99170264, 0.99204533, 0.9921428 ,
        0.9921527 , 0.9921527 , 0.9921527 , 0.9921527 , 0.9921527 ,
        0.9921527 , 0.9921527 , 0.9921527 , 0.9921527 , 0.9921527 ],
       [0.64564205, 0.78391169, 0.91896288, 0.96868714, 0.98727477,
        0.99110178, 0.99198637, 0.99283415, 0.99319852, 0.99329555,
        0.99331059, 0.99331059, 0.99331059, 0.99331059, 0.99331059,
        0.99331059, 0.99331059, 0.99331059, 0.99331059, 0.99331059],
       [0.64521227, 0.78384875, 0.91816475, 0.96762898, 0.98673962,
```

Build models on other algorithms to see the better performing on SMOTE

In [0]:
```
df_Results
```

Out[287]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998982	0.967170	0.003298
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998947	0.869484	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999070	0.869545	1.000000
5	Power Transformer	Random Forest	0.999350	0.930340	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891340	0.001346
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.999034	0.997121	0.001853
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.999052	0.949325	0.027425
10	RepeatedKFold Cross Validation	KNN	0.999333	0.901774	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999192	0.896839	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999087	0.896786	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999631	0.979573	0.020000
14	RepeatedKFold Cross Validation	XGBoost	0.999614	0.997407	0.001658
15	RepeatedKFold Cross Validation	SVM	0.998315	0.469528	0.004318
16	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983339	0.001569
17	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998631	0.933642	0.005137
18	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
19	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998982	0.831413	1.000000
20	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999052	0.816168	1.000000
21	StratifiedKFold Cross Validation	Random Forest	0.999438	0.940527	0.010000
22	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
23	StratifiedKFold Cross Validation	SVM	0.998280	0.401770	0.002409
24	Random Undersampling	Logistic Regression with L2 Regularisation	0.984639	0.963450	0.246874
25	Random Undersampling	Logistic Regression with L1 Regularisation	0.998315	0.500000	1.500000
26	Random Undersampling	KNN	0.975931	0.942016	0.400000
27	Random Undersampling	Tree Model with gini criteria	0.937169	0.890538	1.000000
28	Random Undersampling	Tree Model with entropy criteria	0.954303	0.888721	1.000000
29	Random Undersampling	Random Forest	0.992047	0.976800	0.270000
30	Random Undersampling	XGBoost	0.986535	0.981135	0.108230
31	Random Undersampling	SVM	0.640129	0.551141	0.331891
32	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.986359	0.980812	0.425031
33	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
34	Random Oversampling with StratifiedKFold CV	KNN	0.998069	0.805604	0.200000
35	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.999017	0.816151	1.000000
36	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.999122	0.831483	1.000000
37	Random Oversampling with StratifiedKFold CV	Random Forest	0.999456	0.956230	0.010000
38	Random Oversampling with StratifiedKFold CV	XGBoost	0.996436	0.982501	0.233910
39	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.981654	0.974723	0.446641
40	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
41	SMOTE Oversampling with StratifiedKFold CV	KNN	0.994470	0.860213	0.400000
42	SMOTE Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.997964	0.841089	1.000000
43	SMOTE Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.992890	0.869108	1.000000
44	SMOTE Oversampling with StratifiedKFold CV	Random Forest	0.999491	0.959446	0.070000
45	SMOTE Oversampling with StratifiedKFold CV	XGBoost	0.993030	0.978184	0.273181

Results for SMOTE Oversampling:

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

Oversampling with ADASYN Oversampling

We will use ADASYN Oversampling method to handle the class imbalance

1. First we will display class distibution with and without the ADASYN Oversampling.

2. Then We will use the oversampled with StratifiedKFold cross validation method to genearte Train And test datasets.

Once we have train and test dataset we will feed the data to below models:
1. Logistic Regression with L2 Regularisation
2. Logistic Regression with L1 Regularisation

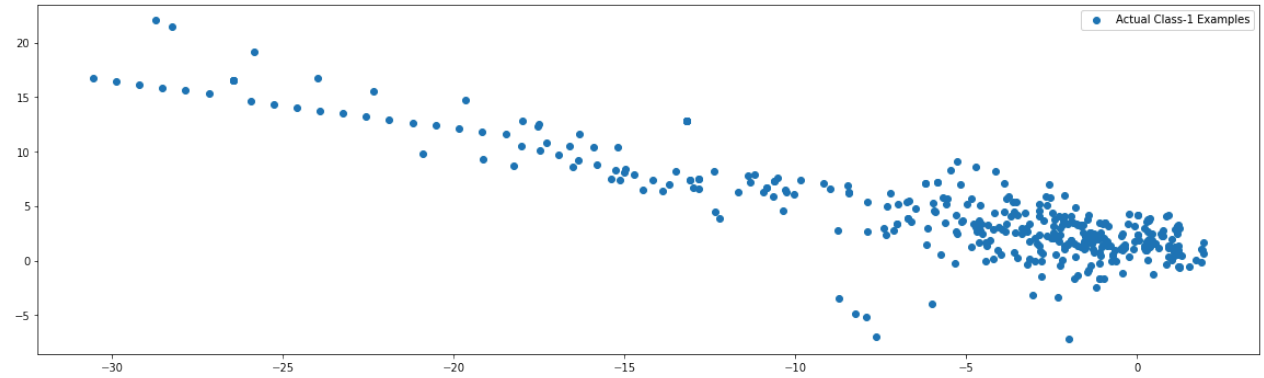
3. KNN
4. Decision tree model with Gini criteria
5. Decision tree model with Entropy criteria
6. Random Forest
7. XGBoost

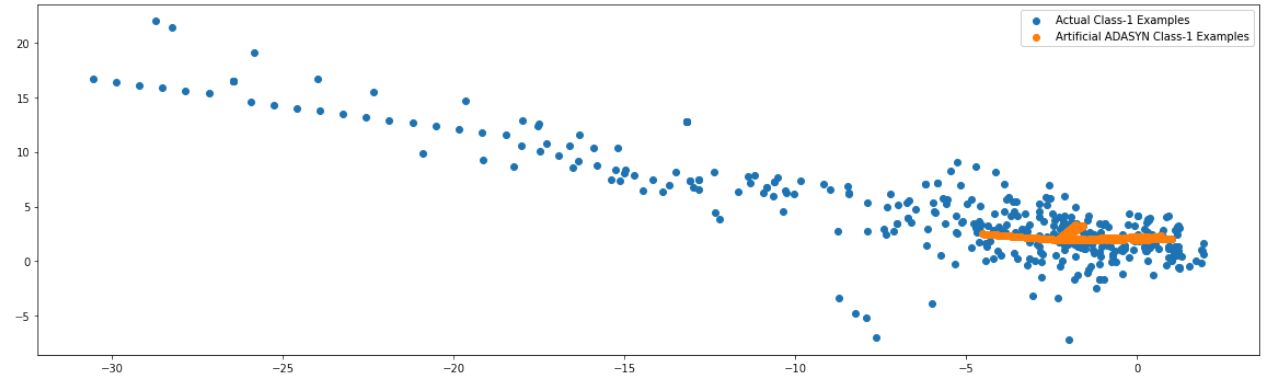
3. We did try SVM (support vector Machine) model , but due to extensive processive power requirement we avoided useing the model.

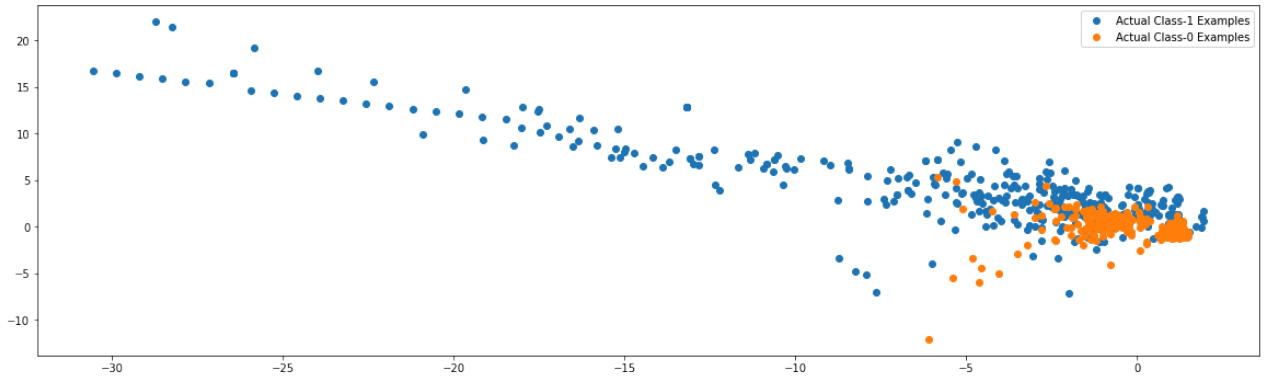
4. Once we get results for above model, we will compare the results and select model which provided best results for the oversampling techinique

In [0]:

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

from imblearn import over_sampling

ADASYN = over_sampling.ADASYN(random_state=0)
X_train_ADASYN, y_train_ADASYN = ADASYN.fit_sample(X_train, y_train)

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

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


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

plt.subplot(3, 1, 1)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.legend()

plt.subplot(3, 1, 2)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.scatter(X_train_adasyn_1[:X_train_1.shape[0], 0], X_train_adasyn_1[:X_train_1.shape[0], 1],
            label='Artificial ADASYN Class-1 Examples')
plt.legend()

plt.subplot(3, 1, 3)
plt.scatter(X_train_1[:, 0], X_train_1[:, 1], label='Actual Class-1 Examples')
plt.scatter(X_train_0[:X_train_1.shape[0], 0], X_train_0[:X_train_1.shape[0], 1], label='Actual Class-0 Examples')
plt.legend()
```







In [0]:
```
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_ADASYN, y_train_ADASYN= ADASYN.fit_sample(X_train, y_train)

#Create Dataframe for X_over
X_train_ADASYN = pd.DataFrame(data=X_train_ADASYN,   columns=cols)
```

Build models on other algorithms to see the better performing on ADASYN

In [0]:
```
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_ADASYN, 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_ADASYN, y_train_ADASYN , X_test, y_t
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 )
#Run SVM Model with Sigmoid Kernel
#print("SVM Model with Sigmoid Kernel")
#start_time = time.time()
#df_Results = buildAndRunSVMModels(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))
```

```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l2: 0.9837840925609461
Max auc_roc for l1: 0.5
Parameters for l2 regularisations
[[ 0.99536558  1.040472    0.86054532  0.95478253  0.78892654 -1.1802698
  -0.92251805 -0.58391691 -0.33968518 -0.90295945  0.80087087 -1.49823958
  -0.14145136 -1.97554486  0.13217482 -0.99877928 -1.125551   -0.50111868
   0.37693768 -1.40758233 -0.18341086  0.72687459  0.65838876 -1.08083968
   0.69686219  0.19660966  0.13556606  1.2859248   0.01275172  0.04680201]]
[-5.43870501]
{1: array([[0.61021517, 0.63159046, 0.72080484, 0.87495728, 0.96155685,
        0.97622443, 0.97996651, 0.98264108, 0.98380927, 0.98416981,
        0.98422199, 0.98422199, 0.98422199, 0.98422199, 0.98422199,
        0.98422199, 0.98422199, 0.98422199, 0.98422199, 0.98422199],
       [0.61045493, 0.63159708, 0.71938814, 0.87086492, 0.9587103 ,
        0.97444715, 0.97873341, 0.98183648, 0.98311067, 0.98346534,
        0.98351997, 0.98351997, 0.98351997, 0.98351997, 0.98351997,
        0.98351997, 0.98351997, 0.98351997, 0.98351997, 0.98351997],
       [0.61406318, 0.63553209, 0.72392735, 0.8757245 , 0.96148026,
```

In [0]:
```
df_Results
```

Out[291]:

	Data_Imbalance_Handiling	Model	Accuracy	roc_value	threshold
0	Power Transformer	Logistic Regression with L2 Regularisation	0.998999	0.966806	0.001289
1	Power Transformer	Logistic Regression with L1 Regularisation	0.998982	0.967170	0.003298
2	Power Transformer	KNN	0.999298	0.900851	0.200000
3	Power Transformer	Tree Model with gini criteria	0.998947	0.869484	1.000000
4	Power Transformer	Tree Model with entropy criteria	0.999070	0.869545	1.000000
5	Power Transformer	Random Forest	0.999350	0.930340	0.010000
6	Power Transformer	XGBoost	0.999315	0.975037	0.000756
7	Power Transformer	SVM	0.998560	0.891340	0.001346
8	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.999034	0.997121	0.001853
9	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.999052	0.949325	0.027425
10	RepeatedKFold Cross Validation	KNN	0.999333	0.901774	0.200000
11	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.999192	0.896839	1.000000
12	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.999087	0.896786	1.000000
13	RepeatedKFold Cross Validation	Random Forest	0.999631	0.979573	0.020000
14	RepeatedKFold Cross Validation	XGBoost	0.999614	0.997407	0.001658
15	RepeatedKFold Cross Validation	SVM	0.998315	0.469528	0.004318
16	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.998771	0.983339	0.001569
17	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.998631	0.933642	0.005137
18	StratifiedKFold Cross Validation	KNN	0.999192	0.805746	0.200000
19	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.998982	0.831413	1.000000
20	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.999052	0.816168	1.000000
21	StratifiedKFold Cross Validation	Random Forest	0.999438	0.940527	0.010000
22	StratifiedKFold Cross Validation	XGBoost	0.999386	0.978148	0.002443
23	StratifiedKFold Cross Validation	SVM	0.998280	0.401770	0.002409
24	Random Undersampling	Logistic Regression with L2 Regularisation	0.984639	0.963450	0.246874
25	Random Undersampling	Logistic Regression with L1 Regularisation	0.998315	0.500000	1.500000
26	Random Undersampling	KNN	0.975931	0.942016	0.400000
27	Random Undersampling	Tree Model with gini criteria	0.937169	0.890538	1.000000
28	Random Undersampling	Tree Model with entropy criteria	0.954303	0.888721	1.000000
29	Random Undersampling	Random Forest	0.992047	0.976800	0.270000
30	Random Undersampling	XGBoost	0.986535	0.981135	0.108230
31	Random Undersampling	SVM	0.640129	0.551141	0.331891
32	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.986359	0.980812	0.425031
33	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
34	Random Oversampling with StratifiedKFold CV	KNN	0.998069	0.805604	0.200000
35	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.999017	0.816151	1.000000
36	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.999122	0.831483	1.000000
37	Random Oversampling with StratifiedKFold CV	Random Forest	0.999456	0.956230	0.010000
38	Random Oversampling with StratifiedKFold CV	XGBoost	0.996436	0.982501	0.233910
39	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.981654	0.974723	0.446641
40	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
41	SMOTE Oversampling with StratifiedKFold CV	KNN	0.994470	0.860213	0.400000
42	SMOTE Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.997964	0.841089	1.000000
43	SMOTE Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.992890	0.869108	1.000000
44	SMOTE Oversampling with StratifiedKFold CV	Random Forest	0.999491	0.959446	0.070000
45	SMOTE Oversampling with StratifiedKFold CV	XGBoost	0.993030	0.978184	0.273181
46	ADASYN Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.949509	0.974374	0.651157
47	ADASYN Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.998280	0.500000	1.500000
48	ADASYN Oversampling with StratifiedKFold CV	KNN	0.994294	0.860157	0.400000
49	ADASYN Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.996998	0.830419	1.000000
50	ADASYN Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.998034	0.856404	1.000000
51	ADASYN Oversampling with StratifiedKFold CV	Random Forest	0.999491	0.948065	0.040000
52	ADASYN Oversampling with StratifiedKFold CV	XGBoost	0.983041	0.978625	0.322957

Results for ADASYN Oversampling:

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

Overall conclusion after running models on Oversampled data:

Looking at above results it seems XGBOOST model with Random Oversampling with StratifiedKFold CV has provided best results.
So we can try to tune the hyperparameters of this model to get best results

But looking at the results Logistic Regression with L2 Regularisation with RepeatedKFold Cross Validation has been provided best results without any oversampling.

#Parameter Tuning for Final Model by Handling class imbalance

In [0]:
```
#Evaluate XGboost model
from xgboost import XGBClassifier
# fit model no training data
XGBmodel = XGBClassifier(random_state=42)
XGBmodel.fit(X_over, y_over)

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

# 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))
```

In [0]:
```
#Evaluate XGboost model
from xgboost import XGBClassifier
# fit model no training data
XGBmodel = XGBClassifier(random_state=42)
```

In [0]:
```
#Lets tune XGBoost Model for max_depth and min_child_weight
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2)
}
gsearch1 = GridSearchCV(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=3,
              min_child_weight=1, missing=None, n_estimators=100, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is deprec
ated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[48]:
```
({'mean_fit_time': array([154.378017  , 152.94472551, 153.29736295, 254.89239678,
         253.53505139, 255.34319501, 353.03661404, 347.02156057,
         344.20010471, 423.01080499, 409.8352325 , 354.94625111]),
  'mean_score_time': array([0.5772203 , 0.5686718 , 0.51954436, 0.84202261, 0.84573722,
         0.83635464, 0.96595922, 0.9154521 , 0.93311915, 1.02788877,
         1.00881209, 0.70457125]),
  'mean_test_score': array([0.99896593, 0.99908345, 0.99918029, 0.99552672, 0.9963417 ,
         0.99497632, 0.99394382, 0.99426184, 0.99675047, 0.99475186,
         0.99887264, 0.99915323]),
  'param_max_depth': masked_array(data=[3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9],
               mask=[False, False, False, False, False, False, False, False,
                     False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_min_child_weight': masked_array(data=[1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5],
               mask=[False, False, False, False, False, False, False, False,
                     False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'max_depth': 3, 'min_child_weight': 1},
   {'max_depth': 3, 'min_child_weight': 3},
   {'max_depth': 3, 'min_child_weight': 5},
   {'max_depth': 5, 'min_child_weight': 1},
   {'max_depth': 5, 'min_child_weight': 3},
   {'max_depth': 5, 'min_child_weight': 5},
   {'max_depth': 7, 'min_child_weight': 1},
   {'max_depth': 7, 'min_child_weight': 3},
   {'max_depth': 7, 'min_child_weight': 5},
   {'max_depth': 9, 'min_child_weight': 1},
   {'max_depth': 9, 'min_child_weight': 3},
   {'max_depth': 9, 'min_child_weight': 5}],
  'rank_test_score': array([ 4,  3,  1,  8,  7,  9, 12, 11,  6, 10,  5,  2], dtype=int32),
  'split0_test_score': array([0.99929168, 0.99931126, 0.99929972, 0.99953766, 0.99947106,
         0.99949342, 0.99947142, 0.99951276, 0.99954324, 0.99946615,
         0.99949479, 0.9995233 ]),
  'split1_test_score': array([0.99983802, 0.99985202, 0.99985937, 0.99999339, 0.99999854,
         0.99999599, 1.        , 1.        , 1.        , 1.        ,
         1.        , 1.        ]),
  'split2_test_score': array([0.99660508, 0.99722304, 0.99771923, 0.97819516, 0.98232552,
         0.97548333, 0.97029647, 0.971869  , 0.98428185, 0.97434316,
         0.99492806, 0.99630903]),
  'split3_test_score': array([0.99950464, 0.99944473, 0.999438  , 0.99992781, 0.99992418,
         0.99992276, 0.9999571 , 0.99992983, 0.99993409, 0.99995406,
         0.9999404 , 0.99993605]),
  'split4_test_score': array([0.99959022, 0.99958621, 0.99958513, 0.99997957, 0.99998918,
         0.9999861 , 0.99999409, 0.99999762, 0.99999316, 0.99999594,
         0.99999996, 0.99999777]),
  'std_fit_time': array([ 1.2864572 ,  0.83377006,  0.38877925,  1.23775132,  0.56333368,
          1.58396372,  3.0904619 ,  4.68574492,  2.56649638,  1.76590515,
          5.58894243, 48.11743496]),
  'std_score_time': array([0.01580999, 0.01339917, 0.10017221, 0.04097969, 0.03644023,
         0.02938585, 0.05922849, 0.05416364, 0.0490002 , 0.07055777,
         0.0804285 , 0.25936635]),
  'std_test_score': array([0.00119333, 0.00094731, 0.00075371, 0.0086674 , 0.00701081,
         0.00974826, 0.01182535, 0.01119788, 0.0062366 , 0.01020633,
         0.00198135, 0.00143314])},
 {'max_depth': 3, 'min_child_weight': 5},
 0.9991802915100555)
```

In [0]:
```
#Lets tune XGBoost Model for n_estimators
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
 'n_estimators':range(60,150,20)
}
gsearch1 = GridSearchCV(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=3,
              min_child_weight=5, missing=None, n_estimators=100, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is deprec
ated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[50]:
```
({'mean_fit_time': array([ 93.55748796, 124.16368618, 155.04023719, 184.51453319,
         185.81143909]),
  'mean_score_time': array([0.36964664, 0.48924289, 0.56491547, 0.64109282, 0.54340262]),
  'mean_test_score': array([0.9971648 , 0.99880929, 0.99918029, 0.99934999, 0.99934307]),
  'param_n_estimators': masked_array(data=[60, 80, 100, 120, 140],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'n_estimators': 60},
   {'n_estimators': 80},
   {'n_estimators': 100},
   {'n_estimators': 120},
   {'n_estimators': 140}],
  'rank_test_score': array([5, 4, 3, 1, 2], dtype=int32),
  'split0_test_score': array([0.99793942, 0.99897899, 0.99929972, 0.99943555, 0.99945592]),
  'split1_test_score': array([0.99937066, 0.99972665, 0.99985937, 0.99992757, 0.99995287]),
  'split2_test_score': array([0.99100365, 0.99689359, 0.99771923, 0.99802611, 0.99777926]),
  'split3_test_score': array([0.99860233, 0.99910894, 0.999438  , 0.99963404, 0.99973366]),
  'split4_test_score': array([0.99890793, 0.99933828, 0.99958513, 0.99972669, 0.99979365]),
  'std_fit_time': array([ 0.40770138,  0.67545008,  0.83700936,  0.64218854, 41.21988455]),
  'std_score_time': array([0.01439796, 0.02122805, 0.0567845 , 0.06306428, 0.21097391]),
  'std_test_score': array([0.00311546, 0.00099095, 0.00075371, 0.00068061, 0.00079821])},
 {'n_estimators': 120},
 0.9993499912901148)
```

In [0]:
```
# We will narror down the tunned parameters of max_depth ,  min_child_weight and n_estimators
#Lets tune XGBoost Model for n_estimators
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
 'n_estimators':[110,120,130],
  'max_depth':[2,3,4],
 'min_child_weight':[4,5,6]
}
gsearch1 = GridSearchCV(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=3,
              min_child_weight=5, missing=None, n_estimators=120, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is depr
ecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[51]:
```
({'mean_fit_time': array([116.83571134, 126.12999382, 137.36606145, 116.03834352,
         126.23316689, 137.03901792, 116.3757113 , 126.62710047,
         137.45390463, 169.88939152, 186.13391142, 201.37337914,
         170.35435586, 188.40744205, 206.86028023, 174.40270581,
         188.25000806, 202.75115585, 228.22967377, 248.92124991,
         274.73475127, 232.47118106, 252.43309417, 274.43020811,
         233.22154098, 252.81742902, 244.71753373]),
  'mean_score_time': array([0.46612463, 0.52484007, 0.55369239, 0.49137659, 0.5310689 ,
         0.56432767, 0.4824707 , 0.53389244, 0.57116213, 0.65584898,
         0.70415812, 0.75159774, 0.65239506, 0.70001698, 0.75196462,
         0.65334458, 0.69881234, 0.74790444, 0.83101544, 0.86137023,
         0.92298226, 0.78676491, 0.87344117, 0.89475231, 0.80407505,
         0.90554757, 0.68143554]),
  'mean_test_score': array([0.99688598, 0.99726727, 0.99774231, 0.99689773, 0.99747423,
         0.99772741, 0.99688569, 0.99745897, 0.99762666, 0.99920592,
         0.99934461, 0.99948304, 0.99921098, 0.99934999, 0.99929426,
```

In [0]:
```
#With 'max_depth': 4, 'min_child_weight': 4, 'n_estimators': 130 parameters tuned above we will now check learning rate

# We will narror down the tunned parameters of max_depth ,  min_child_weight and n_estimators
#Lets tune XGBoost Model for n_estimators
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
 'learning_rate':[0.05,0.1,0.125,0.15,0.2]
}
gsearch1 = GridSearchCV(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=4,
              min_child_weight=4, missing=None, n_estimators=130, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is deprec
ated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[53]:
```
({'mean_fit_time': array([266.2077878 , 268.72819576, 273.50645761, 270.80638585,
        232.40842614]),
  'mean_score_time': array([0.83613429, 0.92765174, 0.94825087, 0.95323839, 0.75981684]),
  'mean_test_score': array([0.99572378, 0.99956123, 0.999078  , 0.9996335 , 0.99985175]),
  'param_learning_rate': masked_array(data=[0.05, 0.1, 0.125, 0.15, 0.2],
               mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
  'params': [{'learning_rate': 0.05},
   {'learning_rate': 0.1},
   {'learning_rate': 0.125},
   {'learning_rate': 0.15},
   {'learning_rate': 0.2}],
  'rank_test_score': array([5, 3, 4, 2, 1], dtype=int32),
  'split0_test_score': array([0.99916385, 0.99947664, 0.99953927, 0.9996356 , 0.99956154]),
  'split1_test_score': array([0.99989976, 0.99998867, 0.99999163, 0.99999856, 1.        ]),
  'split2_test_score': array([0.98018465, 0.9984703 , 0.99596634, 0.99862185, 0.99977268]),
  'split3_test_score': array([0.99958451, 0.99989125, 0.99991553, 0.99992904, 0.99992971]),
  'split4_test_score': array([0.99978615, 0.99997931, 0.99997724, 0.99998248, 0.99999485]),
  'std_fit_time': array([ 0.65685616,  1.15777493,  2.09103134,  0.90751007, 73.64750969]),
  'std_score_time': array([0.02818235, 0.01327884, 0.03287893, 0.0169792 , 0.27732037]),
  'std_test_score': array([0.00777362, 0.00057684, 0.00156461, 0.00052265, 0.00016674])},
 {'learning_rate': 0.2},
 0.9998517547335155)
```

In [0]:
```
#With 'max_depth': 4, 'min_child_weight': 4, 'n_estimators': 130 parameters tuned above we will now check learning rate

# We will narror down the tunned parameters of max_depth ,  min_child_weight and n_estimators
#Lets tune XGBoost Model for n_estimators
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
'gamma':[i/10.0 for i in range(0,5)]
}
gsearch1 = GridSearchCV(estimator = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0,
              learning_rate=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=4, missing=None, n_estimators=130, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is deprec
ated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[54]:
```
({'mean_fit_time': array([268.17632442, 262.58947592, 264.15084901, 267.92146974,
        226.95248747]),
  'mean_score_time': array([0.98027482, 0.97626629, 0.95794263, 0.94473982, 0.68396888]),
  'mean_test_score': array([0.99985175, 0.99986873, 0.99983861, 0.99986593, 0.9998595 ]),
  'param_gamma': masked_array(data=[0.0, 0.1, 0.2, 0.3, 0.4],
               mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
  'params': [{'gamma': 0.0},
   {'gamma': 0.1},
   {'gamma': 0.2},
   {'gamma': 0.3},
   {'gamma': 0.4}],
  'rank_test_score': array([4, 1, 5, 2, 3], dtype=int32),
  'split0_test_score': array([0.99956154, 0.99962305, 0.99962606, 0.99957899, 0.99961592]),
  'split1_test_score': array([1., 1., 1., 1., 1.]),
  'split2_test_score': array([0.99977268, 0.99978411, 0.99965472, 0.99983794, 0.99975305]),
  'split3_test_score': array([0.99992971, 0.99994122, 0.99991725, 0.99991918, 0.99993802]),
  'split4_test_score': array([0.99999485, 0.99999525, 0.99999501, 0.99999355, 0.99999051]),
  'std_fit_time': array([ 2.76890853,  1.25461177,  0.86014662,  2.18647134, 69.16904991]),
  'std_score_time': array([0.05558509, 0.05672296, 0.03313441, 0.03286944, 0.25840669]),
  'std_test_score': array([0.00016674, 0.00014561, 0.00016473, 0.00015506, 0.00015082])},
 {'gamma': 0.1},
 0.9998687267297413)
```

In [0]:
```
#Evaluate XGboost model
from xgboost import XGBClassifier
# fit model no training data
XGBmodel = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0.1,
              learning_rate=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=4, missing=None, n_estimators=130, 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=1, verbosity=1)
XGBmodel.fit(X_over, y_over)

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

# 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))
```

```
Model Accuracy: 0.9993855444953564
XGboost roc_value: 0.9852138347557161
XGBoost threshold: 0.005087878089398146
```

In [0]:
```
#With 'max_depth': 4, 'min_child_weight': 4, 'n_estimators': 130 , gamma: 0.1 parameters tuned above we will now check learn

# We will narror down the tunned parameters of max_depth ,  min_child_weight and n_estimators
#Lets tune XGBoost Model for n_estimators
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
param_test = {
 'subsample':[i/10.0 for i in range(7,10)],
 'colsample_bytree':[i/10.0 for i in range(7,10)]
}

gsearch1 = GridSearchCV(estimator = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0.1,
              learning_rate=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=4, missing=None, n_estimators=130, 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=1, verbosity=1),
 param_grid = param_test, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(X_over, y_over)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter 'iid' is deprec
ated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

Out[56]:
```
({'mean_fit_time': array([188.89507856, 191.38894272, 195.58882852, 208.04385567,
        211.51613936, 215.31259031, 228.27863607, 233.30221796,
        204.55559034]),
  'mean_score_time': array([0.95504174, 0.94060907, 0.97315516, 0.94686146, 0.96284485,
        1.01250434, 0.95457482, 0.9673882 , 0.66616488]),
  'mean_test_score': array([0.9998548 , 0.99981136, 0.99985984, 0.99982328, 0.99988522,
        0.99987986, 0.99965533, 0.99987715, 0.99986361]),
  'param_colsample_bytree': masked_array(data=[0.7, 0.7, 0.7, 0.8, 0.8, 0.8, 0.9, 0.9, 0.9],
               mask=[False, False, False, False, False, False, False, False,
                     False],
        fill_value='?',
             dtype=object),
  'param_subsample': masked_array(data=[0.7, 0.8, 0.9, 0.7, 0.8, 0.9, 0.7, 0.8, 0.9],
               mask=[False, False, False, False, False, False, False, False,
                     False],
        fill_value='?',
             dtype=object),
  'params': [{'colsample_bytree': 0.7, 'subsample': 0.7},
   {'colsample_bytree': 0.7, 'subsample': 0.8},
   {'colsample_bytree': 0.7, 'subsample': 0.9},
   {'colsample_bytree': 0.8, 'subsample': 0.7},
   {'colsample_bytree': 0.8, 'subsample': 0.8},
   {'colsample_bytree': 0.8, 'subsample': 0.9},
   {'colsample_bytree': 0.9, 'subsample': 0.7},
   {'colsample_bytree': 0.9, 'subsample': 0.8},
   {'colsample_bytree': 0.9, 'subsample': 0.9}],
  'rank_test_score': array([6, 8, 5, 7, 1, 2, 9, 3, 4], dtype=int32),
  'split0_test_score': array([0.99962137, 0.99964956, 0.99957462, 0.99964822, 0.99965182,
        0.99961341, 0.99967807, 0.99969066, 0.99954874]),
  'split1_test_score': array([1.        , 0.9999988 , 0.99999773, 0.99999967, 1.        ,
        1.        , 1.        , 1.        , 1.        ]),
  'split2_test_score': array([0.99971956, 0.99947355, 0.99981036, 0.99952629, 0.99983837,
        0.99985525, 0.99869168, 0.99975767, 0.99984246]),
  'split3_test_score': array([0.99994542, 0.99994723, 0.99992327, 0.99995161, 0.99994318,
        0.99993658, 0.99991877, 0.99994657, 0.9999317 ]),
  'split4_test_score': array([0.99998766, 0.99998766, 0.99999324, 0.99999059, 0.99999271,
        0.99999407, 0.99998812, 0.99999085, 0.99999516]),
  'std_fit_time': array([ 0.872237  ,  0.92977773,  0.89993588,  0.82873869,  1.49347485,
         1.38696147,  0.94581146,  0.56854859, 54.84654672]),
  'std_score_time': array([0.04734074, 0.05044447, 0.06115994, 0.02327855, 0.03995797,
        0.08029122, 0.02557596, 0.0244135 , 0.25582243]),
  'std_test_score': array([0.00015474, 0.00021212, 0.00015788, 0.00019719, 0.0001302 ,
        0.00014303, 0.0004956 , 0.00012798, 0.00016741])},
 {'colsample_bytree': 0.8, 'subsample': 0.8},
 0.9998852168905572)
```

In [0]:
```
#Evaluate XGboost model
from xgboost import XGBClassifier
# fit model no training data
XGBmodel = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=0.8,
              colsample_bynode=1, colsample_bytree=1, gamma=0.1,
              learning_rate=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=4, missing=None, n_estimators=130, 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)
XGBmodel.fit(X_over, y_over)

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

# 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))
```

```
Model Accuracy: 0.999350432752234
XGboost roc_value: 0.9765293381478648
XGBoost threshold: 0.0018976784776896238
```

****As the roc value has dropped we will take not consider new values of colsample_bytree': 0.8, 'subsample': 0.8****

In [0]:
```
# perform the best oversampling method on X_train & y_train

clf = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0.1,
              learning_rate=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=4, missing=None, n_estimators=130, 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=1, verbosity=1)
clf.fit(X_over, y_over ) # fit on the balanced dataset
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))
```

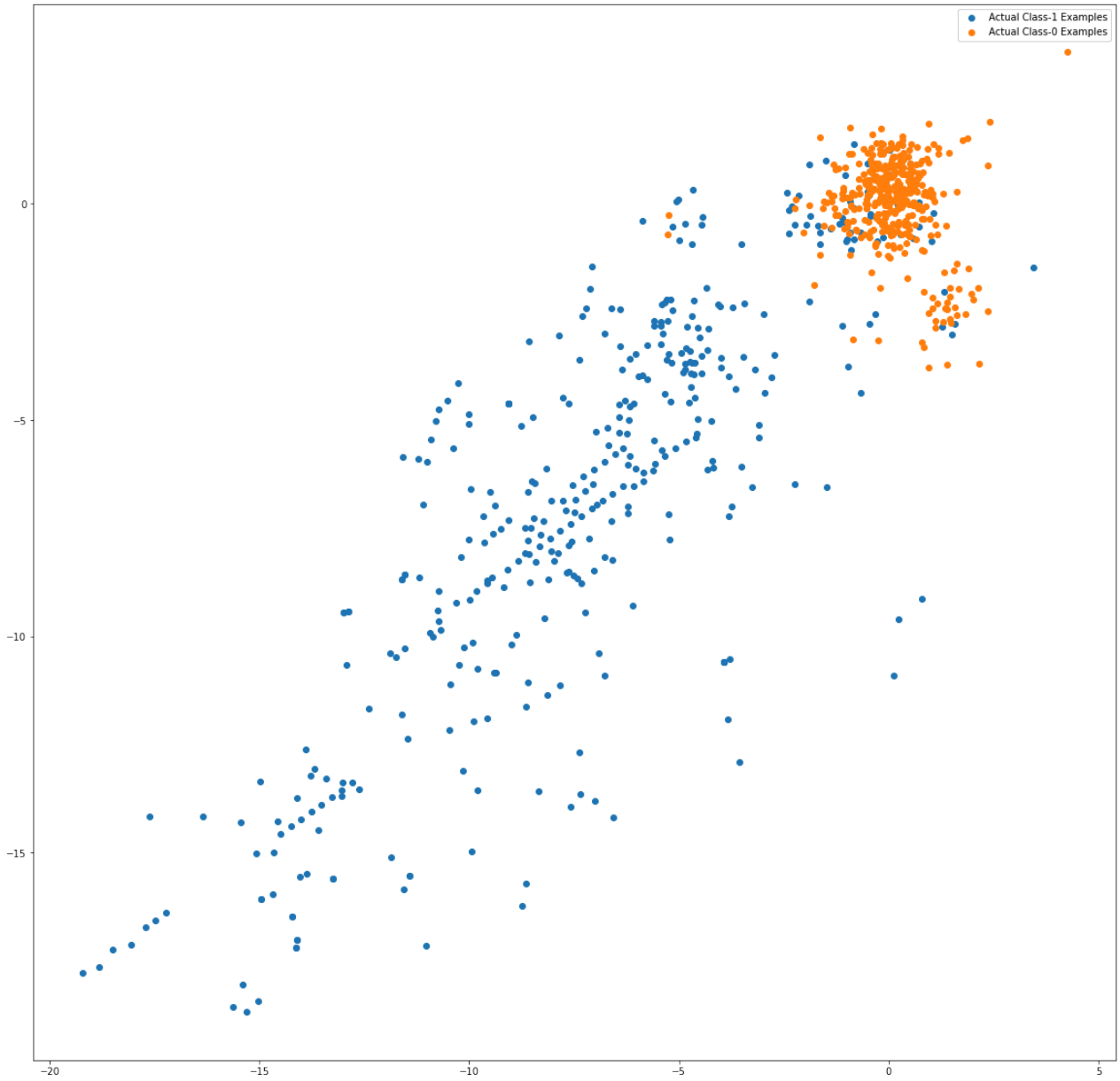
```
Model Accuracy: 0.9993855444953564
XGboost roc_value: 0.9852138347557161
XGBoost threshold: 0.005087878089398146
```

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

In [0]:
```
var_imp = []
for i in clf.feature_importances_:
    var_imp.append(i)
print('Top var =', var_imp.index(np.sort(clf.feature_importances_)[-1])+1)
print('2nd Top var =', var_imp.index(np.sort(clf.feature_importances_)[-2])+1)
print('3rd Top var =', var_imp.index(np.sort(clf.feature_importances_)[-3])+1)

# Variable on Index-13 and Index-9 seems to be the top 2 variables
top_var_index = var_imp.index(np.sort(clf.feature_importances_)[-1])
second_top_var_index = var_imp.index(np.sort(clf.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 = 14
2nd Top var = 12
3rd Top var = 4
```

Out[59]:
```
<matplotlib.legend.Legend at 0x7f179dbe00b8>
```



In [0]:
```
#### Print the FPR,TPR & select the best threshold from the roc curve
```

In [0]:
```
# 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))
```

```
ERROR! Session/line number was not unique in database. History logging moved to new session 59
XGboost roc_value: 0.9852138347557161
XGBoost threshold: 0.005087878089398146
```

Model Selection:
Overall conclusion after running models on Oversampled data:

Looking at above results it seems XGBOOST model with Random Oversampling with StratifiedKFold CV has provided best results.
So we can try to tune the hyperparameters of this model to get best results

We have selected XGBOOST model with Random Oversampling and StratifiedKFold CV

Model Accuracy: 0.9993855444953564

XGboost roc_value: 0.9852138347557161

XGBoost threshold: 0.005087878089398146

We also noticed by looking at the results Logistic Regression with L2 Regularisation with RepeatedKFold Cross Validation has been provided best results without any oversampling.