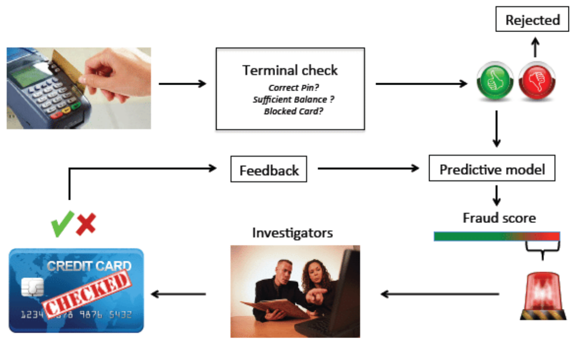
Credit Card Fraud Detection



Problem Statement:

- For many banks, retaining high profitable customers is the number one business goal. Banking fraud, however, poses a significant threat to this goal for different banks. In terms of substantial financial losses, trust and credibility, this is a concerning issue to both banks and customers alike.
- In the banking industry, credit card fraud detection using machine learning and deep learning is not only a trend but a necessity for them to put proactive monitoring and fraud prevention mechanisms in place. Machine learning is helping these institutions to reduce time-consuming manual reviews, costly chargebacks and fees as well as denials of legitimate transactions.
- In this project we will detect fraudulent credit card transactions with the help of Machine learning and deep learnig models.
- We will analyse customer-level data that has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group.

Theory about Credit Card Fraud

- **What is Credit Card Fraud Detection?**
  - Credit card fraud is a term that has been coined for unauthorized access of payment cards like credit cards or debit cards to pay for using services or goods
  - Hackers or fraudsters may obtain the confidential details of the card from unsecured websites. When a fraudster compromises an individual's credit/debit card, everyone involved in the process suffers, right from the individual whose confidential data has been leaked to the businesses (generally banks) who issue the credit card and the merchant who is finalizing the transaction with purchase
  - This makes it extremely essential to identify the fraudulent transactions at the onset. Financial institutions and businesses like e-commerce are taking firm steps to flag the fraudsters entering the system. Various advanced machine learning technologies are at play, assessing every transaction and stemming the fraud users in its nip using behavioral data and transaction patterns
  - The process of automatically differentiating between fraudulent and genuine users is known as “credit card fraud detection”

- **How does Credit Card Fraud work?**
  - A credit card is one of the most used financial products to make online purchases and payments such as gas, groceries, TVs, traveling, shopping bills, and so on because of the non-availability of funds at that instance. Credit cards are of most value that provide various benefits in the form of points while using them for different transactions. There are several categories of credit card fraud that are prevalent in today’s time:

    1. Lost/Stolen cards: People steal credit cards from the mail and use them illegally on behalf of the owner. The process of blocking credit cards that have been stolen and re-issuing them is a hassle for both customers and credit card companies. Some financial institutions keep the credit cards blocked until it is verified that the rightful owner has received the card.
    2. Card Abuse: The customer buys goods and items on the credit card but has no intention to pay back the amount charged by the bank for the same. These customers stop answering the calls as the deadline to settle the dues approaches. Sometimes they even declare bankruptcy—this type of fraud results in losses of millions every year.
    3. Identity Theft: The customers apply illegitimate information, and they might even steal the details of a genuine customer to apply for a credit card and then misuse it. In such cases, even card blocking can not stop the credit card from falling into the wrong hands.
    4. Merchant Abuse: Some merchants show illegal transactions (that never occurred) for money laundering. For performing these illicit transactions, legal information of genuine credit card users is stolen to generate replicas of the cards and use it for illegal work.
  - Many traditional old-school techniques have been used since time in-memorial for credit card fraud detection like CVV verification, geolocation tracking, IP Address verification, etc. But over time, the criminals are using more advanced techniques to commit crimes, and it is impossible to prevent them all using only traditional methods. Millions of transactions are processed every second in today’s world, which takes it beyond human intelligence to process all the data to identify the behavioral patterns of the fraudsters. This is where credit card fraud detection using machine learning plays a vital role.
  - Financial institutions increasingly depend upon automated machine learning systems to make intelligent decisions and protect businesses against substantial losses. These measures play a significant role in reducing the risk while doing online transactions. Like humans, machine learning algorithms learn from past transaction data and use that information to analyze future transactions with the same lens. While machines might not be as intelligent as humans are and might need some supervision on top of it, the advantage lies in the speed of data processing and computation. Also, machines can identify and remember more patterns in vast volumes of data compared to humans. Generally, these algorithms are known as anomaly detection. Let us delve into details in the next section.

Credit Card Fraud Detection using Machine Learning can be done using

1. **Unsupervised Learning** -

   - Machine Learning Algorithms such as Isolation Forest, One-class SVM, LOF, etc., do not require labeled data for training the model. They identify patterns in the data and try to group the data points based on observed similarities in patterns.
2. **Supervised Learning**

   - Machine Learning and Deep Learning Algorithms such as Ensemble Models (RandomForest, XGBoost, LightGBM, etc.), KNN, Neural Networks, Autoencoders, etc. These algorithms are trained on labeled data, and the model learns to predict the labels for the unseen data. Labeled data can be expensive to gather.

Challenges in Credit Card Fraud Detection

- The challenges involved in credit card fraud detection project is primarily the data itself. The data is heavily imbalanced, i.e., the count of data labeled as fraudulent is way less than the data labeled as non-fraudulent data. This makes it extremely tricky to train the model as it tends to overfit for the majority class and underfit for the minority class. Techniques like oversampling, undersampling, cost-sensitive learning, etc. can be used to deal with this. The metrics used for the final model are different from standard evaluation metrics of accuracy, AUC-ROC, etc.
- Another prevalent faced challenge is the quality and quantity of the data. The startups in the early stage do not have much user history data to train extensive models, which makes it difficult to train a robust fraud detection model. A temporary solution to this problem can be sourcing data from an external third party, like scores from credit bureaus.

DATASET DESCRIPTION

- The dataset contains 284,807 transactions among which there are 492 i.e., 0.172% transactions are fraudulent transactions
- It also contains transactions made by a cardholder in 2 days in month of september 2013
- This dataset is highly unbalanced. Due to security reasons, most of the features in the dataset are transformed using principal component analysis (PCA). V1, V2, V3,…, V28 are PCA applied features and rest features include ‘time’, ‘amount’ and ‘class’ are non-PCA applied features

Table of Contents

Importing Dependencies

In [340]:
```
# Importing the libraries
import numpy as np
import pandas as pd
import time

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

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

from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

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

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

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


from tensorflow.keras import Sequential
import tensorflow as tf
from tensorflow.keras.layers import Dense,Dropout

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

Exploratory data analysis

In [341]:
```
# Mounting the google drive
from google.colab import drive
drive.mount('/content/drive')
import os
os.getcwd()
path = "/content/drive/MyDrive/Projects/Credit Card Defaulter"
os.chdir(path)
os.getcwd()
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Out[341]:
```
'/content/drive/MyDrive/Projects/Credit Card Defaulter'
```

In [342]:
```
# Loading the data
df = pd.read_csv('creditcard.csv')
df = df.sample(n=50000)
df.head()
```

Out[342]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
192316	129633.0	-0.541013	1.107931	-1.774693	-0.708025	-0.456628	-0.295740	2.360015	-0.293290	-0.597604	...	0.100783	0.308628	-0.177781	0.642998	0.037229	0.494459	-0.390304	-0.202842	300.00	0
91715	63596.0	-0.627415	1.123128	1.560804	-0.076211	0.223046	-0.904104	0.938718	-0.155628	-0.392705	...	0.067629	0.094912	-0.176009	0.359548	-0.058737	-0.666741	-0.019712	0.135237	4.65	0
140052	83511.0	-0.916656	0.996912	2.114384	1.082337	-1.104885	-0.049506	-0.411277	0.755500	0.369418	...	-0.006409	0.077001	-0.108486	0.345586	-0.031918	-0.303443	0.275835	0.123201	9.99	0
13800	24466.0	-0.886470	-0.126264	3.551005	3.751230	-0.563369	1.141458	-0.889331	0.234025	2.022399	...	-0.058542	0.915166	0.376580	0.375107	-0.396255	0.368435	0.028081	-0.129019	3.80	0
55873	47165.0	1.223111	-0.942174	0.196413	-0.420355	-1.078497	-0.451728	-0.572279	0.023307	-0.216677	...	0.099471	0.171263	-0.112221	0.065022	0.518764	-0.116556	-0.004219	0.012913	80.44	0

5 rows × 31 columns

In [343]:
```
# Checking the shape
df.shape
```

Out[343]:
```
(50000, 31)
```

In [344]:
```
# Checking the datatypes and null/non-null distribution
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 50000 entries, 192316 to 240041
Data columns (total 31 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Time    50000 non-null  float64
 1   V1      50000 non-null  float64
 2   V2      50000 non-null  float64
 3   V3      50000 non-null  float64
 4   V4      50000 non-null  float64
 5   V5      50000 non-null  float64
 6   V6      50000 non-null  float64
 7   V7      50000 non-null  float64
 8   V8      50000 non-null  float64
 9   V9      50000 non-null  float64
 10  V10     50000 non-null  float64
 11  V11     50000 non-null  float64
 12  V12     50000 non-null  float64
 13  V13     50000 non-null  float64
 14  V14     50000 non-null  float64
 15  V15     50000 non-null  float64
 16  V16     50000 non-null  float64
 17  V17     50000 non-null  float64
 18  V18     50000 non-null  float64
 19  V19     50000 non-null  float64
 20  V20     50000 non-null  float64
 21  V21     50000 non-null  float64
 22  V22     50000 non-null  float64
 23  V23     50000 non-null  float64
 24  V24     50000 non-null  float64
 25  V25     50000 non-null  float64
 26  V26     50000 non-null  float64
 27  V27     50000 non-null  float64
 28  V28     50000 non-null  float64
 29  Amount  50000 non-null  float64
 30  Class   50000 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 12.2 MB
```

In [345]:
```
# Checking distribution of numerical values in the dataset
df.describe()
```

Out[345]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	...	50000.000000	50000.000000	50000.000000	50000.000000	50000.00
mean	95027.093240	0.006869	0.000014	0.002052	0.006169	0.003853	0.000950	0.004717	0.004668	0.008224	...	0.003217	-0.002990	0.000050	-0.002233	-0.00
std	47608.097497	1.940516	1.625471	1.504460	1.410004	1.364510	1.322307	1.212862	1.170605	1.091409	...	0.726534	0.725087	0.601028	0.606238	0.51
min	0.000000	-40.470142	-42.172688	-31.103685	-5.683171	-40.427726	-23.496714	-31.197329	-50.943369	-9.462573	...	-22.665685	-8.887017	-23.222016	-2.740677	-4.93
25%	54267.750000	-0.912608	-0.597673	-0.883402	-0.841195	-0.683034	-0.763299	-0.549844	-0.204852	-0.636216	...	-0.229721	-0.546591	-0.160259	-0.356284	-0.31
50%	85203.000000	0.017080	0.072429	0.176216	-0.011505	-0.053596	-0.270196	0.042177	0.022762	-0.046710	...	-0.029802	0.004218	-0.009554	0.041472	0.01
75%	139631.500000	1.315053	0.808100	1.024045	0.759097	0.609536	0.399615	0.573382	0.327584	0.607525	...	0.187433	0.531716	0.149508	0.437811	0.34
max	172792.000000	2.451888	16.713389	3.920275	12.699542	34.099309	23.917837	44.054461	20.007208	10.392889	...	27.202839	7.220158	15.426351	3.998294	4.82

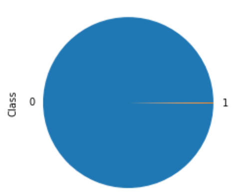
8 rows × 31 columns

In [346]:
```
# Checking the class distribution of the target variable
df['Class'].value_counts()
```

Out[346]:
```
0    49916
1       84
Name: Class, dtype: int64
```

In [347]:
```
# Checking the class distribution of the target variable in percentage
df['Class'].value_counts(normalize = True).plot.pie()
```

Out[347]: <matplotlib.axes._subplots.AxesSubplot at 0x7f53a19b95e0>



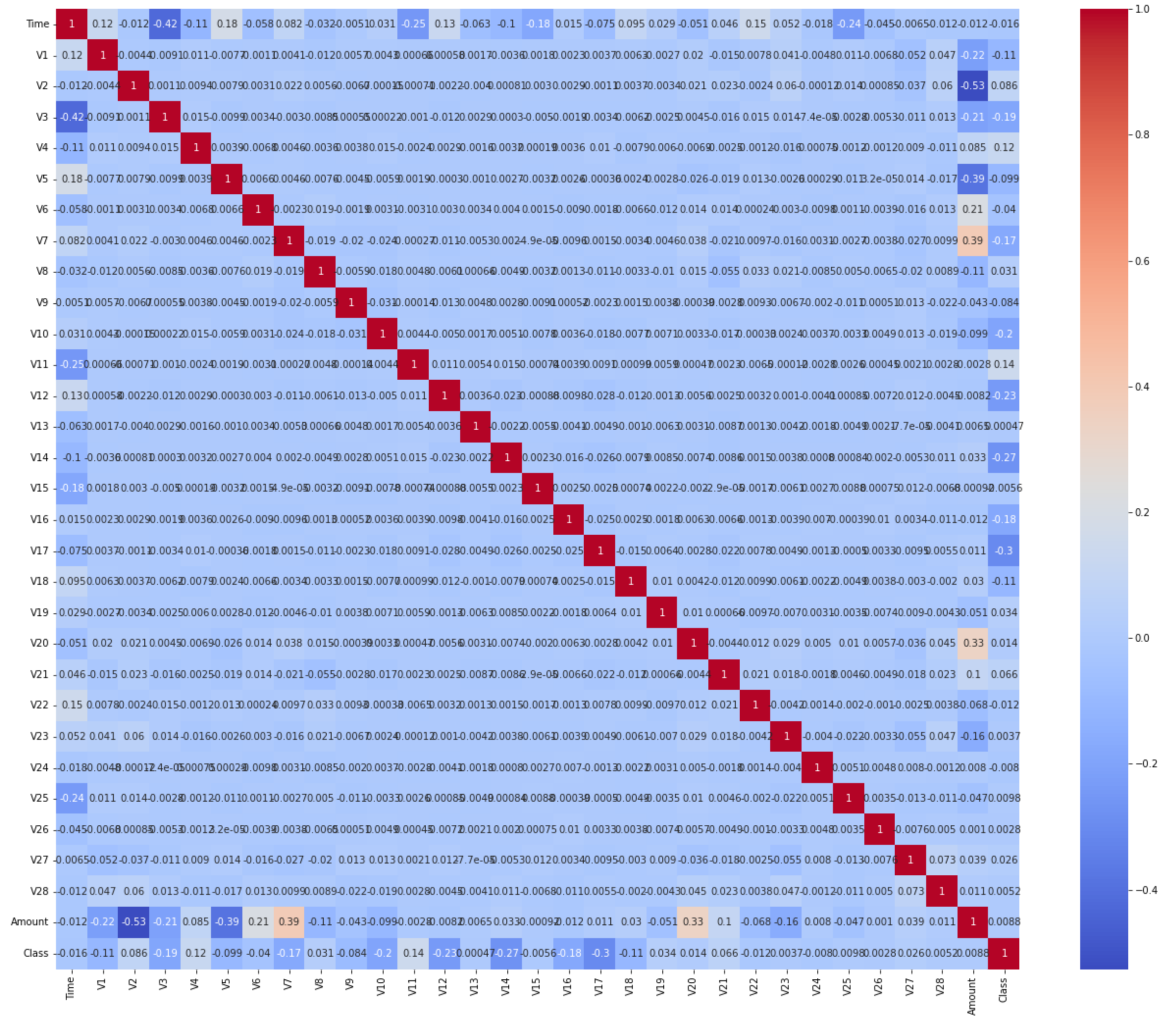
In [348]:
```python
# Checking the correlation
corr = df.corr()
corr
```

Out[348]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Cl
Time	1.000000	0.118292	-0.012198	-0.421144	-0.112760	0.177543	-0.058231	0.082175	-0.032188	-0.005067	...	0.046404	0.147353	0.051976	-0.017890	-0.239786	-0.044932	-0.006478	-0.011512	-0.012154	-0.016
V1	0.118292	1.000000	-0.004389	-0.009109	0.011063	-0.007697	-0.001057	0.004101	-0.011941	0.005733	...	-0.014567	0.007841	0.040755	-0.004787	0.010659	-0.006836	-0.052209	0.047343	-0.221128	-0.107
V2	-0.012198	-0.004389	1.000000	0.001119	0.009434	0.007904	0.003128	0.022083	0.005633	-0.006709	...	0.022919	-0.002429	0.059792	-0.000121	0.014421	0.000852	-0.037145	0.059617	-0.526473	0.085
V3	-0.421144	-0.009109	0.001119	1.000000	0.014623	-0.009876	0.003400	-0.003018	-0.008532	0.000546	...	-0.016443	0.015221	0.014239	-0.000074	-0.002764	-0.005301	-0.011037	0.013405	-0.206868	-0.187
V4	-0.112760	0.011063	0.009434	0.014623	1.000000	0.003917	-0.006781	0.004598	-0.003612	0.003839	...	-0.002481	-0.001170	-0.016054	0.000745	-0.001209	-0.001193	0.009019	-0.010950	0.085269	0.118
V5	0.177543	-0.007697	0.007904	-0.009876	0.003917	1.000000	0.006577	0.004568	-0.007636	-0.004547	...	-0.018828	0.013145	-0.002647	0.000292	-0.011059	0.000032	0.013777	-0.017210	-0.385417	-0.098
V6	-0.058231	-0.001057	0.003128	0.003400	-0.006781	0.006577	1.000000	-0.002341	0.019124	-0.001879	...	0.013933	0.000243	0.002962	-0.009796	0.001131	-0.003912	-0.015692	0.013130	0.210204	-0.040
V7	0.082175	0.004101	0.022083	-0.003018	0.004598	0.004568	-0.002341	1.000000	-0.018584	-0.020129	...	-0.020720	0.009716	-0.015913	0.003073	-0.002654	-0.003813	-0.026919	0.009925	0.390978	-0.173
V8	-0.032188	-0.011941	0.005633	-0.008532	-0.003612	-0.007636	0.019124	-0.018584	1.000000	-0.005950	...	-0.054987	0.033091	0.020526	-0.008456	0.004981	-0.006505	-0.020052	0.008887	-0.106148	0.031
V9	-0.005067	0.005733	-0.006709	0.000546	0.003839	-0.004547	-0.001879	-0.020129	-0.005950	1.000000	...	-0.002835	0.009277	-0.006674	-0.002013	-0.011032	0.000511	0.013312	-0.021642	-0.043101	-0.083
V10	0.030502	0.004307	-0.000151	0.000223	0.015494	-0.005913	0.003096	-0.024081	-0.017924	-0.030983	...	-0.017353	-0.000331	0.002377	0.003696	-0.003346	0.004854	0.012708	-0.019358	-0.099478	-0.197
V11	-0.250578	0.000665	-0.000712	-0.001009	-0.002416	0.001904	-0.003124	-0.000269	0.004796	-0.000142	...	0.002300	-0.006485	-0.000117	-0.002833	0.002569	0.000449	0.002089	0.002802	-0.002800	0.144
V12	0.125820	0.000583	-0.002155	-0.011736	0.002873	-0.000296	0.002972	-0.010803	-0.006058	-0.012543	...	0.002496	0.003197	0.001026	-0.004147	0.000852	-0.007224	0.012033	-0.004487	-0.008161	-0.234
V13	-0.062962	0.001689	-0.003999	0.002939	-0.001558	-0.001034	0.003397	-0.005260	0.000664	0.004843	...	-0.008727	0.001335	-0.004190	-0.001789	-0.004900	0.002078	-0.000077	-0.004109	0.006545	0.000
V14	-0.101219	-0.003645	0.000807	0.000297	0.003207	0.002666	0.004044	0.002015	-0.004924	0.002770	...	-0.008612	0.001467	0.003836	0.000799	0.000844	0.001969	-0.005305	0.010606	0.032959	-0.270
V15	-0.182811	0.001768	0.002957	-0.004989	0.000194	-0.003227	0.001456	-0.000049	-0.003204	-0.009113	...	-0.000029	-0.001682	-0.006076	0.002694	0.008768	0.000749	0.011544	-0.006817	-0.000919	-0.005
V16	0.015188	0.002290	0.002878	-0.001924	0.003615	0.002571	-0.008992	-0.009611	0.001285	0.000524	...	-0.006569	-0.001318	-0.003929	0.006953	-0.000389	0.010345	0.003444	-0.010813	-0.011884	-0.179
V17	-0.075270	0.003702	-0.001134	-0.003380	0.010248	-0.000359	-0.001768	0.001545	-0.011023	-0.002319	...	-0.021812	0.007780	0.004943	-0.001321	-0.000501	0.003339	-0.009543	0.005462	0.010560	-0.304
V18	0.095193	0.006327	0.003657	-0.006155	-0.007904	0.002368	-0.006642	-0.003419	-0.003290	0.001512	...	-0.011524	0.009922	-0.006112	-0.002160	-0.004865	0.003844	-0.002958	-0.001955	0.030029	-0.105
V19	0.028739	-0.002696	-0.003370	-0.002532	0.006034	0.002800	-0.011828	-0.004611	-0.010370	0.003776	...	0.000664	-0.009657	-0.007034	0.003117	-0.003531	-0.007377	0.009047	-0.004321	-0.051108	0.034
V20	-0.051428	0.019607	0.021118	0.004534	-0.006939	-0.025632	0.013653	0.037741	0.014759	-0.000393	...	-0.004366	0.012152	0.029424	0.005008	0.010408	0.005749	-0.036387	0.044919	0.332523	0.014
V21	0.046404	-0.014567	0.022919	-0.016443	-0.002481	-0.018828	0.013933	-0.020720	-0.054987	-0.002835	...	1.000000	0.020952	0.017877	-0.001753	0.004629	-0.004908	-0.017734	0.022524	0.104362	0.066
V22	0.147353	0.007841	-0.002429	0.015221	-0.001170	0.013145	0.000243	0.009716	0.033091	0.009277	...	0.020952	1.000000	-0.004213	0.001411	-0.001968	-0.001008	-0.002465	0.003759	-0.068103	-0.012
V23	0.051976	0.040755	0.059792	0.014239	-0.016054	-0.002647	0.002962	-0.015913	0.020526	-0.006674	...	0.017877	-0.004213	1.000000	-0.003951	-0.021767	-0.003337	-0.055086	0.046792	-0.159856	0.003
V24	-0.017890	-0.004787	-0.000121	-0.000074	0.000745	0.000292	-0.009796	0.003073	-0.008456	-0.002013	...	-0.001753	0.001411	-0.003951	1.000000	0.005079	0.004826	0.008047	-0.001192	0.007971	-0.008
V25	-0.239786	0.010659	0.014421	-0.002764	-0.001209	-0.011059	0.001131	-0.002654	0.004981	-0.011032	...	0.004629	-0.001968	-0.021767	0.005079	1.000000	0.003479	-0.013041	-0.010587	-0.047447	0.009
V26	-0.044932	-0.006836	0.000852	-0.005301	-0.001193	0.000032	-0.003912	-0.003813	-0.006505	0.000511	...	-0.004908	-0.001008	-0.003337	0.004826	0.003479	1.000000	-0.007550	0.004986	0.001026	0.002
V27	-0.006478	-0.052209	-0.037145	-0.011037	0.009019	0.013777	-0.015692	-0.026919	-0.020052	0.013312	...	-0.017734	-0.002465	-0.055086	0.008047	-0.013041	-0.007550	1.000000	0.073469	0.038908	0.026
V28	-0.011512	0.047343	0.059617	0.013405	-0.010950	-0.017210	0.013130	0.009925	0.008887	-0.021642	...	0.022524	0.003759	0.046792	-0.001192	-0.010587	0.004986	0.073469	1.000000	0.010640	0.005
Amount	-0.012154	-0.221128	-0.526473	-0.206868	0.085269	-0.385417	0.210204	0.390978	-0.106148	-0.043101	...	0.104362	-0.068103	-0.159856	0.007971	-0.047447	0.001026	0.038908	0.010640	1.000000	0.008
Class	-0.016079	-0.107550	0.085935	-0.187882	0.118884	-0.098504	-0.040388	-0.173761	0.031241	-0.083785	...	0.066410	-0.012237	0.003708	-0.008032	0.009835	0.002806	0.026460	0.005176	0.008788	1.000

31 rows × 31 columns

In [349]:
```python
# Checking the correlation in heatmap
plt.figure(figsize=(22,18))

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

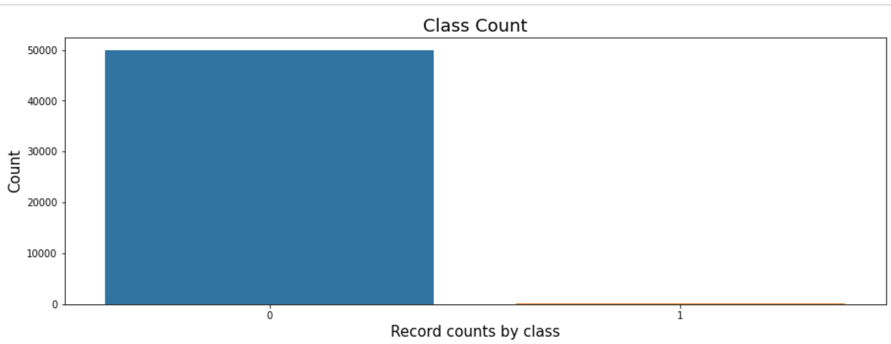


Here we will observe the distribution of our classes

In [350]:
```python
# Checking the % distribution of normal vs fraud
classes=df['Class'].value_counts()
normal_share=classes[0]/df['Class'].count()*100
fraud_share=classes[1]/df['Class'].count()*100

print(normal_share)
print(fraud_share)
```

```
99.832
0.168
```

In [351]:
```python
# Create a bar plot for the number and percentage of fraudulent vs non-fraudulent transcations
plt.figure(figsize=(15,5))
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 [352]:
```
# As time is given in relative fashion, we are using pandas.Timedelta which Represents a duration, the difference between two times or dates.
Delta_Time = pd.to_timedelta(df['Time'], unit='s')

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

In [353]:
```
# Drop unnecessary columns
# We will drop Time,as we have derived the Day/Hour/Minutes from the time column
df.drop('Time', axis = 1, inplace= True)
# We will keep only derived column hour, as day/minutes might not be very useful
df.drop(['Time_Day', 'Time_Min'], axis = 1, inplace= True)
```

Splitting the data into train & test data

In [354]:
```
# Splitting the dataset into X and y
y= df['Class']
X = df.drop(['Class'], axis=1)
```

In [355]:
```
# Checking some rows of X
X.head()
```

Out[355]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Time_Hou
192316	-0.541013	1.107931	-1.774693	-0.708025	-0.456628	-0.295740	2.360015	-0.293290	-0.597604	-0.513867	...	0.100783	0.308628	-0.177781	0.642998	0.037229	0.494459	-0.390304	-0.202842	300.00	1
91715	-0.627415	1.123128	1.560804	-0.076211	0.223046	-0.904104	0.938718	-0.155628	-0.392705	-0.595657	...	0.067629	0.094912	-0.176009	0.359548	-0.058737	-0.666741	-0.019712	0.135237	4.65	1
140052	-0.916656	0.996912	2.114384	1.082337	-1.104885	-0.049506	-0.411277	0.755500	0.369418	-0.500382	...	-0.006409	0.077001	-0.108486	0.345586	-0.031918	-0.303443	0.275835	0.123201	9.99	2
13800	-0.886470	-0.126264	3.551005	3.751230	-0.563369	1.141458	-0.889331	0.234025	2.022399	0.262257	...	-0.058542	0.915166	0.376580	0.375107	-0.396255	0.368435	0.028081	-0.129019	3.80	
55873	1.223111	-0.942174	0.196413	-0.420355	-1.078497	-0.451728	-0.572279	0.023307	-0.216677	0.492314	...	0.099471	0.171263	-0.112221	0.065022	0.518764	-0.116556	-0.004219	0.012913	80.44	1

5 rows × 30 columns

In [356]:
```
# Checking some rows of y
y.head()
```

Out[356]:
```
192316    0
91715     0
140052    0
13800     0
55873     0
Name: Class, dtype: int64
```

In [357]:
```
# Splitting the dataset using train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1, test_size=0.20)
```

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

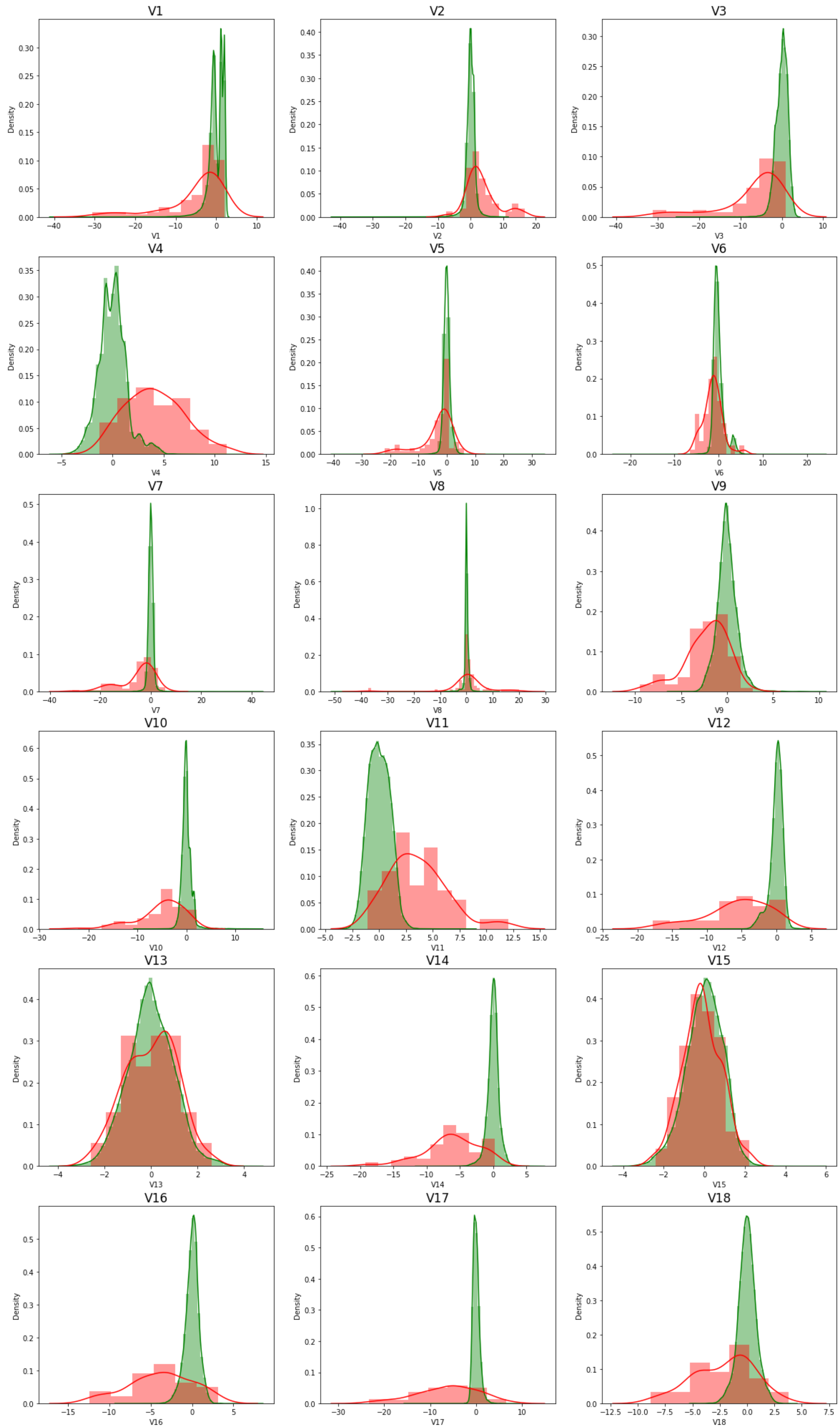
In [358]:
```
# Checking the spread of data post split
print(np.sum(y))
print(np.sum(y_train))
print(np.sum(y_test))
```

```
84
69
15
```

Plotting the distribution of a variable

In [359]:
```
# Accumulating all the column names under one variable
cols = list(X.columns.values)
```

In [360]:
```
# 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()
```

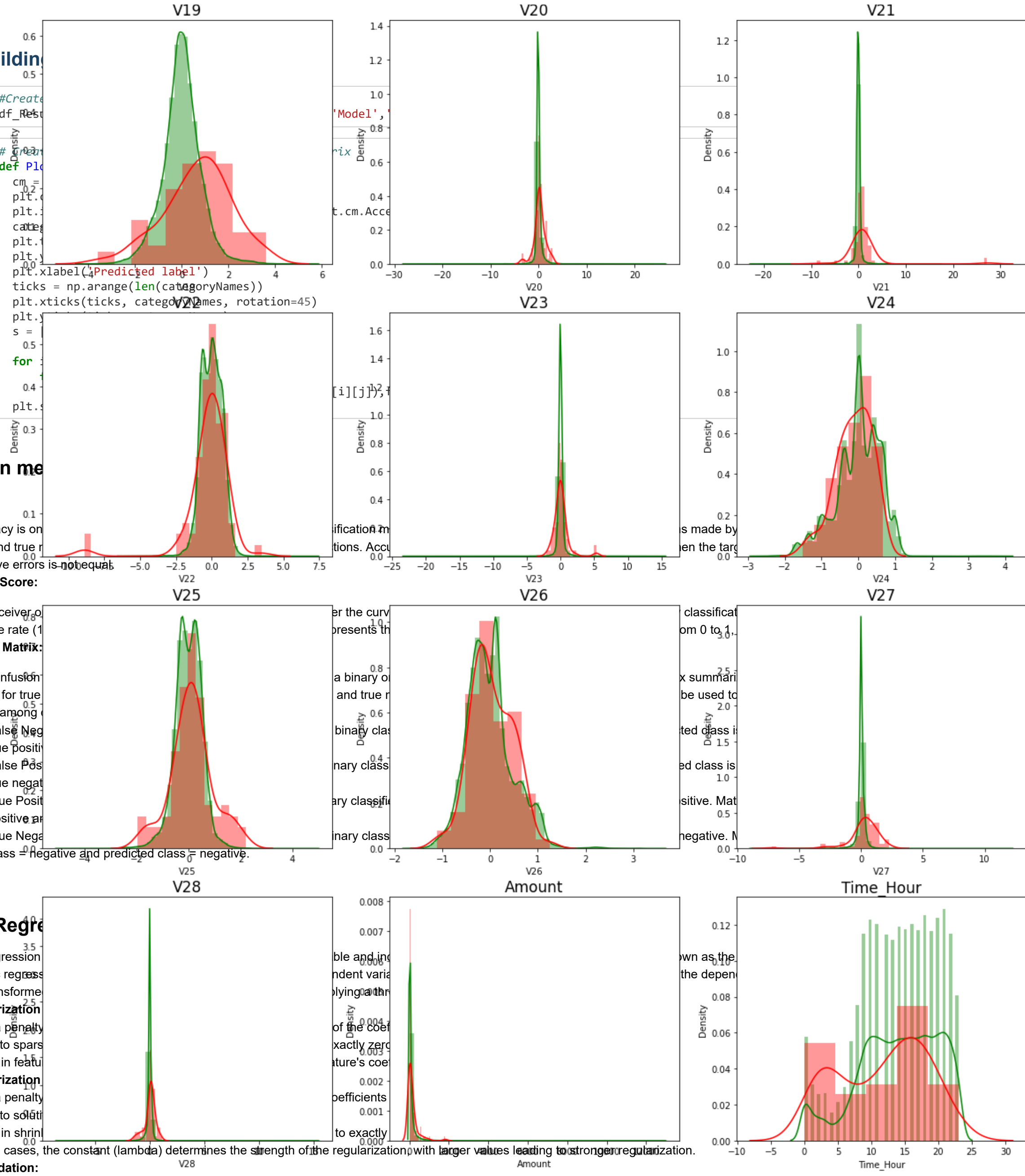


V1
V2
V3
V4
V5
V6
V7
V8
V9
V10
V11
V12
V13
V14
V15
V16
V17
V18
Density

Model Building

```
In [361]: #Create ...
          df_Res...                                   'Model','...
```

```
In [362]: # ...                                       ...rix
          def Pl...
              cm = ...
              plt.c...
              plt.i...                                ...t.cm.Acce...
              cate...
              plt.t...
              plt.y...
              plt.xlabel('Predicted label')
              ticks = np.arange(len(categoryNames))
              plt.xticks(ticks, categoryNames, rotation=45)
              plt.y...
              s = [...
              for ...
                                                      ...[i][j]),...
              plt.s...
```

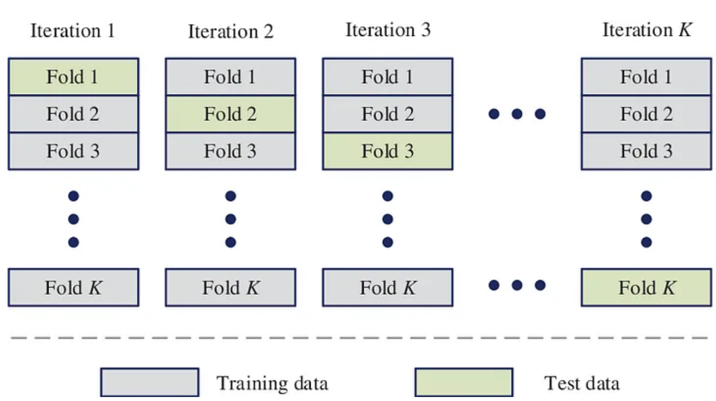


Evaluation me...

1. **Accuracy:**
   - Accuracy is on... ...ification m... ...s made by... ...ated as the ratio of true positive (TP) and true ... ...tions. Accu... ...en the targ... ...he cost of false positive and false negative errors is not equal.
2. **ROC-AUC Score:**
   - The receiver o... ...er the curv... ...classificat... ...e (sensitivity) against the false positive rate (... ...presents th... ...om 0 to 1.
3. **Confusion Matrix:**
   - The confusion ... ...a binary o... ...x summari... ...the model and provides the values for true ... ...and true n... ...be used to... ...cy, precision, recall, and F1-score, among ...
     - False Neg... ...binary clas... ...ted class i... ...s FN = actual positive instances - true positi...
     - False Pos... ...nary class... ...ed class is... ...FP = actual negative instances - true negat...
     - True Posit... ...ary classifi... ...sitive. Mat... ...of instances where actual class = positive a...
     - True Nega... ...inary class... ...negative. N... ...er of instances where actual class = negative and predicted class = negative.

Logistic Regre...

- Logistic regression ... ...ble and in... ...wn as the... ...etween 0 and 1.
- The logistic regress... ...ndent varia... ...the depen... ...e predicted probabilities can then be transforme... ...plying a thr...
- **L1 Regularization**
  - Adds a penalty ... ...of the coef...
  - Leads to spars... ...xactly zero...
  - Result in featu... ...ture's coef...
- **L2 Regularization**
  - Adds a penalty ... ...oefficients ...
  - Leads to soluti... ...to exactly ...
  - Result in shrin...
  - In both cases, the constant (lambda) determines the strength of the regularization, with larger values leading to stronger regularization.
- **Cross Validation:**
  - Cross Validation is a technique for evaluating the performance of machine learning models.
  - The idea is to split the available data into several parts (folds), use one part for testing and the remaining parts for training, repeat this process multiple times with different folds being used as the test set, and finally average the results to get a better estimate of the model's performance.
- **K-Fold Cross Validation:**
  - K-Fold Cross Validation is a specific implementation of cross validation where the data is divided into k equal parts, or folds.
  - The model is trained on k-1 folds and tested on the remaining one, the process is repeated k times with each fold being used as the test set once. The average performance across all k iterations is used as the performance measure for the model.



In [363]:
```
# # Created a common function to fit and predict on a Logistic Regression model for both L1 and L2
def buildAndRunLogisticModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):

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

  num_C = list(np.power(10.0, np.arange(-10, 10)))
  cv_num = KFold(n_splits=10, shuffle=True, random_state=42)
  # CV is used for cross validation in LogisticRegressionCV
  searchCV_l2 = linear_model.LogisticRegressionCV(
          Cs= num_C
          ,penalty='l2'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
          ,max_iter=10000
          ,fit_intercept=True
          ,solver='newton-cg'
          ,tol=10
      )

  searchCV_l1 = linear_model.LogisticRegressionCV(
          Cs=num_C
          ,penalty='l1'
          ,scoring='roc_auc'
          ,cv=cv_num
          ,random_state=42
          ,max_iter=10000
          ,fit_intercept=True
          ,solver='liblinear'
          ,tol=10
      )

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

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

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


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


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

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

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

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

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

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

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logistic Regression with L2 Regularisation','Accuracy': Accuracy_l2,'roc_value': l2_roc_valu
e,'threshold': threshold}, index=[0]),ignore_index= True)

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

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

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'Logistic Regression with L1 Regularisation','Accuracy': Accuracy_l1,'roc_value': l1_roc_valu
e,'threshold': threshold}, index=[0]),ignore_index= True)
  return df_Results
```

KNeighborsClassifier

- KNeighborsClassifier is a class in the scikit-learn library that implements the K-Nearest Neighbors (KNN) algorithm for classification.
- It is a simple and effective algorithm for solving classification problems by finding the closest training examples and assigning the majority class of these nearest neighbors to the test sample. The number of nearest neighbors (k) and the distance metric used can be specified as parameters during model training.

In [364]:
```python
# Created a common function to fit and predict on a KNN model
def buildAndRunKNNModels(df_Results,Methodology, X_train,y_train, X_test, y_test ):

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

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


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

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

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

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

  return df_Results
```

Decision Tree Classifier

- Predictive Model:
  - Decision Tree Classifier is a supervised learning algorithm used for classification problems. It creates a tree-like model of decisions and their possible consequences, which can be used to predict the class of a new data point. The algorithm starts at the root node and splits the data into different branches based on the values of the features. It continues this process recursively until it reaches the leaf nodes, which represent the final prediction.
- Tree Representation:
  - Decision trees can be easily visualized, which makes them a useful tool for interpreting the relationship between the features and the target variable. The tree representation also makes it possible to understand the logic behind the model's predictions. Each internal node in the tree represents a feature, and each branch represents a possible value of that feature. The leaves of the tree represent the class predictions for the instances that reach that leaf.

In [365]:
```python
# Created a common function to fit and predict on a Tree models for both gini and entropy criteria
def buildAndRunTreeModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  #Evaluate Decision Tree model with 'gini' & 'entropy'
  criteria = ['gini', 'entropy']
  scores = {}

  for c in criteria:
      dt = DecisionTreeClassifier(criterion = c, random_state=42)
      dt.fit(X_train, y_train)
      y_pred = dt.predict(X_test)
      test_score = dt.score(X_test, y_test)
      tree_preds = dt.predict_proba(X_test)[:, 1]
      tree_roc_value = roc_auc_score(y_test, tree_preds)
      scores = test_score
      print(c + " score: {0}" .format(test_score))
      print("Confusion Matrix")
      Plot_confusion_matrix(y_test, y_pred)
      print("classification Report")
      print(classification_report(y_test, y_pred))
      print(c + " tree_roc_value: {0}" .format(tree_roc_value))
      fpr, tpr, thresholds = metrics.roc_curve(y_test, tree_preds)
      threshold = thresholds[np.argmax(tpr-fpr)]
      print("Tree threshold: {0}".format(threshold))
      roc_auc = metrics.auc(fpr, tpr)
      print("ROC for the test dataset",'{:.1%}'.format(roc_auc))
      plt.plot(fpr,tpr,label="Test, auc="+str(roc_auc))
      plt.legend(loc=4)
      plt.show()

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

  return df_Results
```

Random Forest Classifier

- Ensemble Method:
  - Random Forest Classifier is an ensemble learning method for classification problems. It creates multiple decision trees and combines their predictions to produce a final output. This helps to reduce overfitting, which is a common issue with decision trees. The idea behind this method is to train multiple trees on random subsets of the data, and average their predictions to produce a more robust result.
- Bagging and Feature Selection:
  - Random Forest Classifier is based on two main concepts: Bagging (Bootstrapped Aggregating) and feature selection. Bagging is a resampling technique used to create multiple training sets from the original data. In Random Forest Classifier, each decision tree is trained on a different bootstrapped sample of the data. Feature selection is used to randomly select a subset of features for each split, which helps to reduce the correlation between trees and improve the overall accuracy of the model. The combination of bagging and feature selection results in a strong and reliable prediction model.

In [366]:
```python
# Created a common function to fit and predict on a Random Forest model
def buildAndRunRandomForestModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  #Evaluate Random Forest model

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

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


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

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

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

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

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

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

  return df_Results
```

XGBoost (eXtreme Gradient Boosting) Classifier

- Gradient Boosting:
  - XGBoost (eXtreme Gradient Boosting) Classifier is an implementation of gradient boosting for classification problems. It is an advanced version of gradient boosting that is designed to handle large datasets and high-dimensional features. XGBoost Classifier trains weak decision trees in a sequential manner, using the errors from the previous tree to improve the predictions of the next tree. This process continues until a set number of trees have been trained, or a specified stopping criterion is met.
- Scalability and Performance:
  - XGBoost Classifier is known for its scalability and performance. It uses parallel processing and efficient algorithms to train models quickly and accurately. It also includes a number of optimization techniques, such as regularization, to help prevent overfitting and improve the generalization performance of the model. XGBoost Classifier has won several machine learning competitions and is widely used in industry due to its ability to handle large datasets and produce accurate predictions.

In [367]:
```python
# Created a common function to fit and predict on a XGBoost model
def buildAndRunXGBoostModels(df_Results, Methodology,X_train,y_train, X_test, y_test ):
  #Evaluate XGboost model
  XGBmodel = XGBClassifier(random_state=42)
  XGBmodel.fit(X_train, y_train)
  y_pred = XGBmodel.predict(X_test)

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

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

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

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

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

  return df_Results
```

Support Vector Classifier (SVC)

- Support Vector Machines:
  - Support Vector Classifier (SVC) is a type of Support Vector Machine (SVM) algorithm used for classification problems. SVC is a linear classifier that finds the hyperplane that best separates the data into classes. The goal of the algorithm is to maximize the margin between the hyperplane and the closest data points, called support vectors, which are used to define the boundary between the classes.
- Kernel Trick:
  - SVC can also be used for non-linearly separable data by using a technique called the kernel trick. The kernel trick transforms the data into a high-dimensional feature space, where a linear boundary can be found. The transformation is performed implicitly, so the user does not need to explicitly perform the calculation. This makes SVC a versatile algorithm that can be used for a wide range of classification problems, including both linearly separable and non-linearly separable data.
- Note: SVC can also be used for regression problems, in which case it is called Support Vector Regression (SVR).

In [368]:
```
# Created a common function to fit and predict on a SVM model
def buildAndRunSVMModels(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  #Evaluate SVM model with sigmoid kernel  model
  from sklearn.svm import SVC
  from sklearn.metrics import accuracy_score
  from sklearn.metrics import roc_auc_score

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

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

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

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

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

  return df_Results
```

ANN

In [369]:
```
# Created a common function to fit and predict on a SVM model
def buildAndgetmetricsANN(df_Results, Methodology, X_train,y_train, X_test, y_test ):
  from sklearn.metrics import accuracy_score
  from sklearn.metrics import roc_auc_score

  ann = Sequential([Dense(input_dim = 30, units = 16, activation = 'relu'),
  Dense(units = 24, activation = 'relu'),
  Dropout(0.5),
  Dense(units = 20, activation = 'relu'),
  Dense(units = 24, activation = 'relu'),
  Dense(units =1, activation = 'sigmoid'),])

  ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
  ann.fit(X_train, y_train, batch_size = 15, epochs = 5)

  ann.fit(X_train,y_train)
  y_pred_ann_prob = ann.predict(X_test)
  y_pred_ann = np.where(y_pred_ann_prob>0.5,1,0)
  ANN_Score = accuracy_score(y_test,y_pred_ann)
  print("accuracy_score : {0}".format(ANN_Score))
  print("Confusion Matrix")
  Plot_confusion_matrix(y_test, y_pred_ann)
  print("classification Report")
  print(classification_report(y_test, y_pred_ann))

  # 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, y_pred_ann_prob)

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

  df_Results = df_Results.append(pd.DataFrame({'Methodology': Methodology,'Model': 'ANN','Accuracy': ANN_Score,'roc_value': roc_value,'threshold': threshold}, index=[0]),ignore_i
ndex= True)

  return df_Results
```

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

Perform cross validation with RepeatedKFold

RepeatedKFold

- Cross-Validation:
  - RepeatedKFold is a type of cross-validation technique used for model selection and evaluation. Cross-validation is a method for evaluating the performance of a machine learning model by dividing the data into training and testing sets, and training the model on the training set and evaluating it on the testing set. RepeatedKFold is a variation of k-fold cross-validation, where the same k-folds are used multiple times to get a better estimate of the model's performance.
- Estimating Model Performance:
  - The purpose of RepeatedKFold is to get a more reliable estimate of the model's performance by repeating the k-fold process multiple times and averaging the results. By repeating the process, RepeatedKFold helps to reduce the variance of the estimate, making it a more robust evaluation technique. Additionally, by using the same folds multiple times, the user can ensure that all parts of the data are used for both training and testing, giving a more comprehensive evaluation of the model's performance.

In [370]:

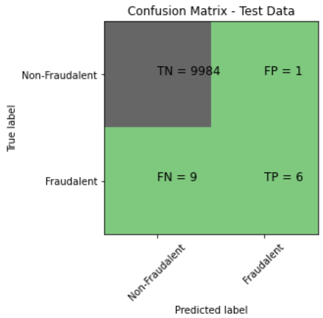
```python
#Lets perfrom RepeatedKFold and check the results
from sklearn.model_selection import RepeatedKFold
rkf = RepeatedKFold(n_splits=5, n_repeats=10, random_state=None)
# X is the feature set and y is the target
for train_index, test_index in rkf.split(X,y):
    print("TRAIN:", train_index, "TEST:", 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     2     3 ... 49994 49996 49999] TEST: [    1     8    22 ... 49995 49997 49998]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [   13    16    18 ... 49989 49994 49996]
TRAIN: [    1     2     3 ... 49996 49997 49998] TEST: [    0     4     7 ... 49992 49993 49999]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    5     9    14 ... 49963 49977 49978]
TRAIN: [    0     1     4 ... 49997 49998 49999] TEST: [    2     3     6 ... 49975 49987 49990]
TRAIN: [    0     1     3 ... 49996 49997 49999] TEST: [    2    11    55 ... 49983 49991 49998]
TRAIN: [    0     1     2 ... 49994 49995 49998] TEST: [    4     5    10 ... 49996 49997 49999]
TRAIN: [    0     2     4 ... 49997 49998 49999] TEST: [    1     3    14 ... 49984 49993 49995]
TRAIN: [    1     2     3 ... 49997 49998 49999] TEST: [    0     7     9 ... 49988 49990 49994]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    6     8    16 ... 49978 49987 49989]
TRAIN: [    0     1     2 ... 49994 49996 49997] TEST: [    6     9    10 ... 49995 49998 49999]
TRAIN: [    0     4     5 ... 49996 49998 49999] TEST: [    1     2     3 ... 49984 49987 49997]
TRAIN: [    1     2     3 ... 49997 49998 49999] TEST: [    0     5    11 ... 49990 49991 49996]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    7     8    13 ... 49975 49977 49993]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    4    19    25 ... 49982 49989 49994]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    8    21    24 ... 49984 49993 49994]
TRAIN: [    0     2     5 ... 49997 49998 49999] TEST: [    1     3     4 ... 49975 49982 49987]
TRAIN: [    1     3     4 ... 49996 49997 49998] TEST: [    0     2     5 ... 49989 49995 49999]
TRAIN: [    0     1     2 ... 49996 49998 49999] TEST: [    9    15    18 ... 49991 49992 49997]
TRAIN: [    0     1     2 ... 49995 49997 49999] TEST: [    7    29    45 ... 49986 49996 49998]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [   11    14    19 ... 49987 49992 49996]
TRAIN: [    0     1     2 ... 49996 49997 49998] TEST: [    4     6    12 ... 49991 49993 49999]
TRAIN: [    1     2     4 ... 49997 49998 49999] TEST: [    0     3     7 ... 49981 49989 49994]
TRAIN: [    0     3     4 ... 49997 49998 49999] TEST: [    1     2     5 ... 49961 49966 49990]
TRAIN: [    0     1     2 ... 49994 49996 49999] TEST: [    8    10    22 ... 49995 49997 49998]
TRAIN: [    0     1     2 ... 49996 49997 49999] TEST: [    4     9    14 ... 49989 49992 49998]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [   10    11    12 ... 49966 49974 49983]
TRAIN: [    3     4     5 ... 49996 49998 49999] TEST: [    0     1     2 ... 49972 49994 49997]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    3     6     8 ... 49991 49993 49996]
TRAIN: [    0     1     2 ... 49996 49997 49998] TEST: [    5     7    15 ... 49987 49995 49999]
TRAIN: [    0     2     4 ... 49997 49998 49999] TEST: [    1     3     5 ... 49978 49990 49992]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    4     8    26 ... 49984 49991 49993]
TRAIN: [    0     1     2 ... 49993 49995 49997] TEST: [    6     7    12 ... 49996 49998 49999]
TRAIN: [    1     2     3 ... 49997 49998 49999] TEST: [    0     9    22 ... 49987 49989 49995]
TRAIN: [    0     1     3 ... 49996 49998 49999] TEST: [    2    11    13 ... 49983 49985 49997]
TRAIN: [    0     1     2 ... 49996 49997 49998] TEST: [    6     8    11 ... 49970 49978 49999]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    5    12    17 ... 49992 49994 49995]
TRAIN: [    0     3     5 ... 49997 49998 49999] TEST: [    1     2     4 ... 49990 49993 49996]
TRAIN: [    1     2     4 ... 49996 49998 49999] TEST: [    0     3     7 ... 49985 49991 49997]
TRAIN: [    0     1     2 ... 49996 49997 49999] TEST: [    9    13    14 ... 49988 49989 49998]
TRAIN: [    0     1     3 ... 49996 49998 49999] TEST: [    2     6    17 ... 49988 49991 49997]
TRAIN: [    0     1     2 ... 49996 49997 49999] TEST: [    5    13    14 ... 49987 49992 49998]
TRAIN: [    0     2     3 ... 49997 49998 49999] TEST: [    1    11    27 ... 49989 49993 49994]
TRAIN: [    1     2     5 ... 49997 49998 49999] TEST: [    0     3     4 ... 49982 49995 49996]
TRAIN: [    0     1     2 ... 49996 49997 49998] TEST: [    9    15    20 ... 49979 49990 49999]
TRAIN: [    0     1     2 ... 49994 49996 49999] TEST: [    5     8     9 ... 49995 49997 49998]
TRAIN: [    1     2     3 ... 49996 49997 49998] TEST: [    0    13    16 ... 49981 49988 49999]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [    6    14    15 ... 49992 49994 49996]
TRAIN: [    0     2     3 ... 49997 49998 49999] TEST: [    1     4     7 ... 49985 49987 49991]
TRAIN: [    0     1     4 ... 49997 49998 49999] TEST: [    2     3    11 ... 49984 49989 49993]
```

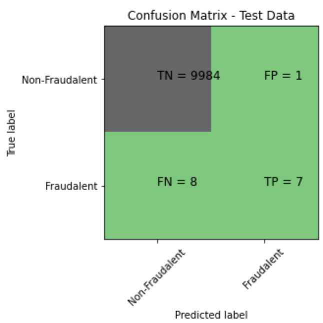
In [371]:

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

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

#Run Decision Tree Models with  'gini' & 'entropy' criteria
print("Decision Tree Models with  'gini' & 'entropy' criteria")
start_time = time.time()
df_Results = buildAndRunTreeModels(df_Results, Data_Validation_Method ,X_train_cv,y_train_cv, X_test_cv, y_test_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

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

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

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

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

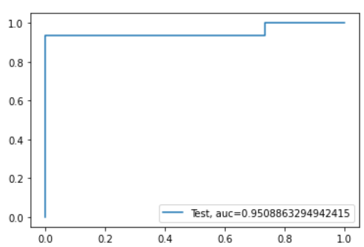
```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.9632348876283263
Max auc_roc for l2: 0.9584189367560885
Parameters for l1 regularisations
[[-0.05916746 -0.14318117 -0.18150557  0.05851872 -0.27074988  0.09183386
   0.14170428 -0.03610091 -0.15757988 -0.13344612 -0.03987543  0.01458198
  -0.135869   -0.21607118  0.01094898 -0.10418107 -0.21198404  0.03788204
   0.01613639  0.30092196  0.08855907 -0.019655    0.0457783  -0.01608273
   0.02177818  0.00577888 -0.10557782  0.01302801 -0.00660377 -0.11590942]]
[-2.08620727]
{1: array([[0.5       , 0.5       , 0.5       , 0.5       , 0.41701076,
        0.48761261, 0.61524024, 0.55868368, 0.99931181, 1.        ,
        1.        , 1.        , 1.        , 1.        , 1.        ,
        1.        , 0.99993744, 1.        , 1.        , 1.        ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.59346029,
        0.59134441, 0.57185612, 0.58018041, 0.87407779, 0.87482948,
        0.84208915, 0.86185584, 0.86670008, 0.86152176, 0.85394916,
        0.84520727, 0.84078065, 0.87352098, 0.86344275, 0.90253069],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.45969925,
        0.43964912, 0.49819549, 0.5385213 , 0.87102757, 0.93265664,
        0.82075188, 0.88501253, 0.93275689, 0.91591479, 0.83488722,
        0.93857143, 0.8712782 , 0.9141604 , 0.86055138, 0.99308271],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.58969925,
        0.56963659, 0.60986216, 0.59397243, 0.77929825, 0.86140351,
        0.83300752, 0.83729323, 0.88779449, 0.85674185, 0.89077694,
        0.89263158, 0.82255639, 0.86701754, 0.91468672, 0.96704261],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.66948198,
        0.62831582, 0.5640015 , 0.58427177, 0.75337838, 0.81637888,
        0.68512262, 0.71458959, 0.81206206, 0.74399399, 0.81781782,
        0.86286286, 0.69901151, 0.77752753, 0.86273774, 0.85635636],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.39401095,
        0.43071804, 0.44542235, 0.42191693, 0.62448571, 0.88089872,
        0.85052413, 0.86633752, 0.89817896, 0.8750313 , 0.905549  ,
        0.94157633, 0.87746413, 0.89495904, 0.90744517, 0.98615434],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.29233301,
        0.3078423 , 0.36955744, 0.3192909 , 0.84998748, 0.98415084,
        0.88859075, 0.92290079, 0.97098494, 0.96357912, 0.90966334,
        0.96060964, 0.9301635 , 0.9561733 , 0.69832922, 0.99062645],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.6000167 ,
        0.55620981, 0.53906011, 0.53279601, 0.97962081, 0.94320555,
        0.91962471, 0.9282274 , 0.93515966, 0.92396782, 0.91235836,
        0.91227484, 0.92831092, 0.93362844, 0.91728612, 0.93758178],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.5650025 ,
        0.51786161, 0.47394269, 0.50203328, 0.90077578, 0.92842843,
        0.80580581, 0.85003754, 0.97522523, 0.9227978 , 0.9771021 ,
        0.98936436, 0.8262012 , 0.9369995 , 0.99543293, 0.99912412],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.51349186,
        0.50876095, 0.4571965 , 0.4852816 , 0.99939925, 0.99964956,
        0.98493116, 0.98763454, 0.99964956, 0.99564456, 0.99964956,
        0.99984981, 0.98683354, 0.99964956, 0.99984981, 0.99984981]])}
Parameters for l2 regularisations
[[ 5.99668738e-03 -3.01823363e-03 -1.12397706e-01  2.47784799e-01
   1.05734896e-01 -3.77486716e-02  2.10189897e-02 -1.28127106e-01
  -1.10876988e-01 -1.68084395e-01  1.60997492e-01 -1.69798369e-01
  -5.27692239e-02 -3.95079590e-01 -5.02671609e-02 -6.92493777e-02
  -8.86719602e-02 -3.14514377e-03  1.67428310e-02  7.75524119e-03
   2.16229124e-02  2.53131116e-02  2.04644575e-02 -2.78403521e-02
   4.25101475e-03  6.58098416e-03 -2.25813405e-03 -2.10530722e-02
   2.82424821e-04 -4.49599848e-02]]
[-6.57797975]
{1: array([[0.6788038 , 0.67892893, 0.68218218, 0.7078954 , 0.83383383,
        0.97378629, 0.99937437, 0.99993744, 0.99981231, 0.99968719,
        0.99968719, 0.99968719, 0.99968719, 0.99968719, 0.99968719,
        0.99968719, 0.99968719, 0.99968719, 0.99968719, 0.99968719],
       [0.53590022, 0.5360951 , 0.53381219, 0.57281662, 0.74812773,
        0.8629973 , 0.88268048, 0.88281968, 0.88680086, 0.91695203,
        0.91695203, 0.91695203, 0.91695203, 0.91695203, 0.91695203,
        0.91695203, 0.91695203, 0.91695203, 0.91695203, 0.91695203],
       [0.6664411 , 0.66641604, 0.66934837, 0.6935589 , 0.7981203 ,
        0.88799499, 0.92042607, 0.94433584, 0.97922306, 0.97385965,
        0.97385965, 0.97385965, 0.97385965, 0.97385965, 0.97385965,
        0.97385965, 0.97385965, 0.97385965, 0.97385965, 0.97385965],
       [0.4497995 , 0.4497995 , 0.45072682, 0.46401003, 0.57325815,
        0.72536341, 0.84661654, 0.91932331, 0.94325815, 0.94285714,
        0.94285714, 0.94285714, 0.94285714, 0.94285714, 0.94285714,
        0.94285714, 0.94285714, 0.94285714, 0.94285714, 0.94285714],
       [0.32995495, 0.32995495, 0.33051802, 0.33508509, 0.37875375,
        0.52195946, 0.6971972 , 0.82388639, 0.84922422, 0.80061311,
        0.80061311, 0.80061311, 0.80061311, 0.80061311, 0.80061311,
        0.80061311, 0.80061311, 0.80061311, 0.80061311, 0.80061311],
       [0.67936746, 0.67936746, 0.68158563, 0.70086938, 0.80823584,
        0.92919752, 0.94633466, 0.95985832, 0.97928518, 0.96865944,
        0.96865944, 0.96865944, 0.96865944, 0.96865944, 0.96865944,
        0.96865944, 0.96865944, 0.96865944, 0.96865944, 0.96865944],
       [0.84347608, 0.84347608, 0.84540804, 0.86519266, 0.94232764,
        0.98103825, 0.98958892, 0.99556366, 0.99445458, 0.99484813,
        0.99484813, 0.99484813, 0.99484813, 0.99484813, 0.99484813,
        0.99484813, 0.99484813, 0.99484813, 0.99484813, 0.99484813],
       [0.50034801, 0.50034801, 0.50232467, 0.5316128 , 0.70564325,
        0.89362176, 0.9471032 , 0.95448092, 0.95314458, 0.92931318,
        0.92931318, 0.92931318, 0.92931318, 0.92931318, 0.92931318,
        0.92931318, 0.92931318, 0.92931318, 0.92931318, 0.92931318],
       [0.45883383, 0.45883383, 0.4608984 , 0.47472472, 0.57157157,
        0.76332583, 0.95245245, 0.99843594, 0.99918669, 0.99931181,
        0.99931181, 0.99931181, 0.99931181, 0.99931181, 0.99931181,
        0.99931181, 0.99931181, 0.99931181, 0.99931181, 0.99931181],
       [0.59389237, 0.59389237, 0.59549437, 0.62187735, 0.78478098,
        0.96290363, 0.999199  , 0.99974969, 0.99979975, 0.99984981,
        0.99984981, 0.99984981, 0.99984981, 0.99984981, 0.99984981,
        0.99984981, 0.99984981, 0.99984981, 0.99984981, 0.99984981]])}
Accuarcy of Logistic model with l2 regularisation : 0.999
Confusion Matrix
```



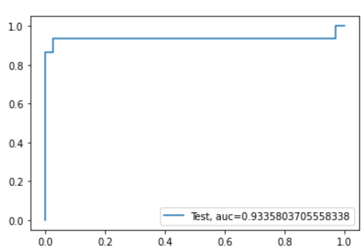

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.86      0.40      0.55        15

    accuracy                           1.00     10000
   macro avg       0.93      0.70      0.77     10000
weighted avg       1.00      1.00      1.00     10000

Accuarcy of Logistic model with l1 regularisation : 0.9991
Confusion Matrix
```



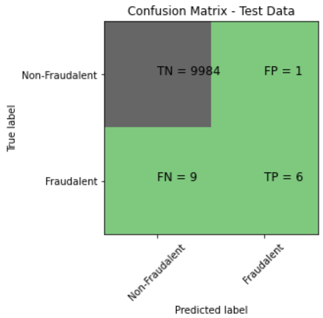

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.88      0.47      0.61        15

    accuracy                           1.00     10000
   macro avg       0.94      0.73      0.80     10000
weighted avg       1.00      1.00      1.00     10000

l2 roc_value: 0.9508863294942415
l2 threshold: 0.030689645354979082
ROC for the test dataset 95.1%
```



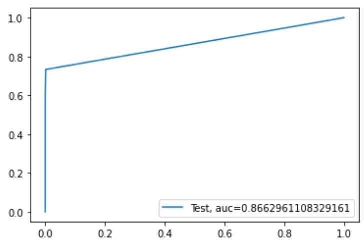

```
l1 roc_value: 0.9335803705558338
l1 threshold: 0.05783283302624465
ROC for the test dataset 93.4%
```



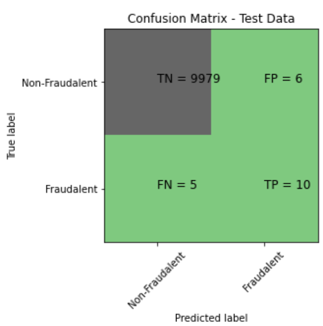

```
Time Taken by Model: --- 51.19481015205383 seconds ---
-----------------------------------------------------------
KNN Model
model score
0.999
Confusion Matrix
```



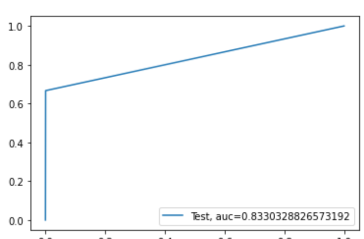

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.86      0.40      0.55        15

    accuracy                           1.00     10000
   macro avg       0.93      0.70      0.77     10000
weighted avg       1.00      1.00      1.00     10000

KNN roc_value: 0.8662961108329161
KNN threshold: 0.2
ROC for the test dataset 86.6%
```



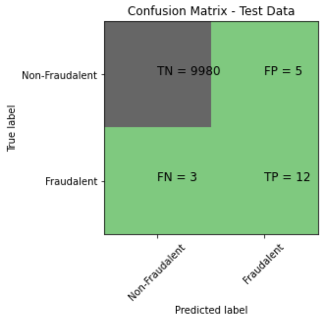

```
Time Taken by Model: --- 45.10881972312927 seconds ---
------------------------------------------------------------
Decision Tree Models with  'gini' & 'entropy' criteria
gini score: 0.9989
Confusion Matrix
```



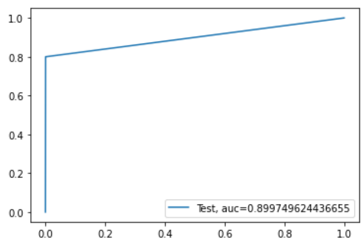

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.62      0.67      0.65        15

    accuracy                           1.00     10000
   macro avg       0.81      0.83      0.82     10000
weighted avg       1.00      1.00      1.00     10000

gini tree_roc_value: 0.8330328826573192
Tree threshold: 1.0
ROC for the test dataset 83.3%
```



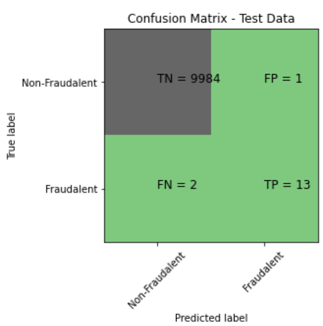

```
entropy score: 0.9992
Confusion Matrix
```



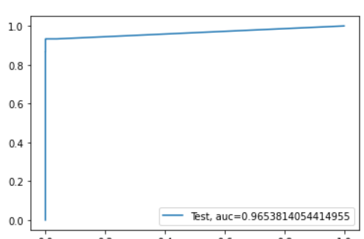

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.71      0.80      0.75        15

    accuracy                           1.00     10000
   macro avg       0.85      0.90      0.87     10000
weighted avg       1.00      1.00      1.00     10000

entropy tree_roc_value: 0.899749624436655
Tree threshold: 1.0
ROC for the test dataset 90.0%
```



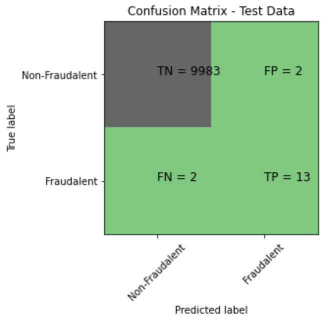

```
Time Taken by Model: --- 3.956048011779785 seconds ---
------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9997
Confusion Matrix
```



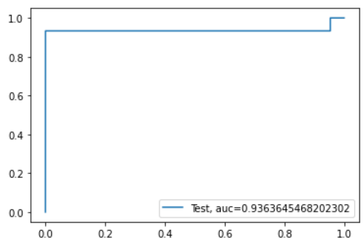

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.93      0.87      0.90        15

    accuracy                           1.00     10000
   macro avg       0.96      0.93      0.95     10000
weighted avg       1.00      1.00      1.00     10000

Random Forest roc_value: 0.9653814054414955
Random Forest threshold: 0.3
ROC for the test dataset 96.5%
```



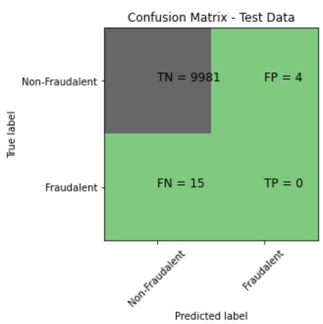

```
Time Taken by Model: --- 22.89270567893982 seconds ---
------------------------------------------------------------
XGBoost Model
Model Accuracy: 0.9996
Confusion Matrix
```



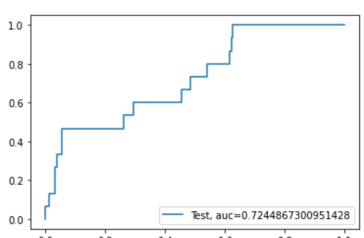

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.87      0.87      0.87        15

    accuracy                           1.00     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       1.00      1.00      1.00     10000

XGboost roc_value: 0.9363645468202302
XGBoost threshold: 0.28312182426452637
ROC for the test dataset 93.6%
```



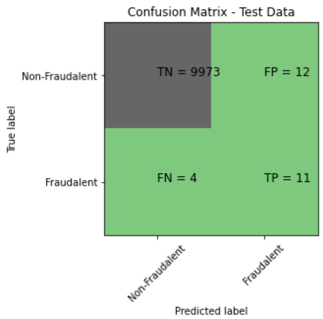

```
Time Taken by Model: --- 9.490098476409912 seconds ---
------------------------------------------------------------
SVM Model with Sigmoid Kernel
accuracy_score : 0.9981
Confusion Matrix
```



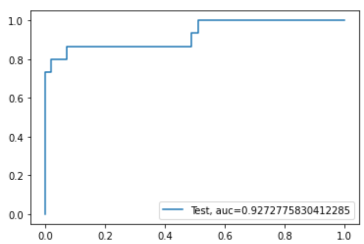

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.00      0.00      0.00        15

    accuracy                           1.00     10000
   macro avg       0.50      0.50      0.50     10000
weighted avg       1.00      1.00      1.00     10000

SVM roc_value: 0.7244867300951428
SVM threshold: 0.0034816380928626425
ROC for the test dataset 72.4%
```



```
Time Taken by Model: --- 6.716370582580566 seconds ---
ANN Model
Epoch 1/5
2130/2130 [==============================] - 8s 3ms/step - loss: 0.1777 - accuracy: 0.9484
Epoch 2/5
2130/2130 [==============================] - 5s 3ms/step - loss: 0.0412 - accuracy: 0.9887
Epoch 3/5
2130/2130 [==============================] - 7s 3ms/step - loss: 0.0343 - accuracy: 0.9907
Epoch 4/5
2130/2130 [==============================] - 6s 3ms/step - loss: 0.0242 - accuracy: 0.9944
Epoch 5/5
2130/2130 [==============================] - 6s 3ms/step - loss: 0.0181 - accuracy: 0.9956
999/999 [==============================] - 3s 3ms/step - loss: 0.0138 - accuracy: 0.9967
313/313 [==============================] - 1s 2ms/step
accuracy_score : 0.9984
Confusion Matrix
```




```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.48      0.73      0.58        15

    accuracy                           1.00     10000
   macro avg       0.74      0.87      0.79     10000
weighted avg       1.00      1.00      1.00     10000

ANN roc_value: 0.9272775830412285
ANN threshold: 0.0005211673560552299
ROC for the test dataset 92.7%
```




```
Time Taken by Model: --- 38.665369749069214 seconds ---
--------------------------------------------------------------------------------
```

In [372]:
```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Out[372]:

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9990	0.950886	0.030690
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.933580	0.057833
2	RepeatedKFold Cross Validation	KNN	0.9990	0.866296	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.9989	0.833033	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.9992	0.899750	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.9997	0.965381	0.300000
6	RepeatedKFold Cross Validation	XGBoost	0.9996	0.936365	0.283122
7	RepeatedKFold Cross Validation	SVM	0.9981	0.724487	0.003482
8	RepeatedKFold Cross Validation	ANN	0.9984	0.927278	0.000521

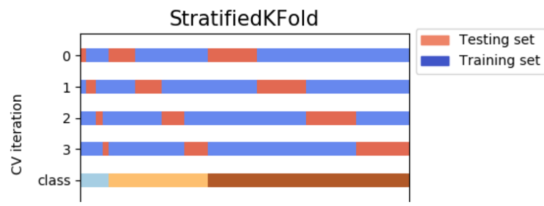
Results for cross validation with RepeatedKFold:

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

Perform cross validation with StratifiedKFold

StratifiedKFold

- StratifiedKFold is a cross-validation technique that splits a dataset into k folds while preserving the class distribution of the samples.
- This method is useful for handling imbalanced datasets, as it ensures that each fold contains a balanced representation of all class labels.



In [373]:
```
#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: [ 9053  9725 10002 ... 49997 49998 49999] TEST: [    0     1     2 ...  9999 10000 10001]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [ 9053  9725 10002 ... 20003 20004 20005]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [17659 18269 19032 ... 30001 30002 30003]
TRAIN: [    0     1     2 ... 49997 49998 49999] TEST: [29558 29634 29870 ... 40000 40001 40002]
TRAIN: [    0     1     2 ... 40000 40001 40002] TEST: [38516 39164 39215 ... 49997 49998 49999]
```

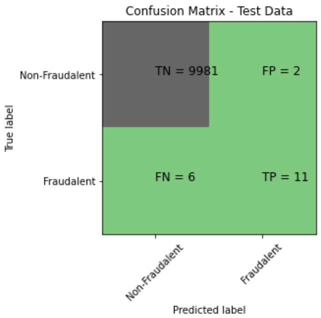
In [374]:
```
y_train_SKF_cv.value_counts()
```

Out[374]:
```
0    39933
1       67
Name: Class, dtype: int64
```

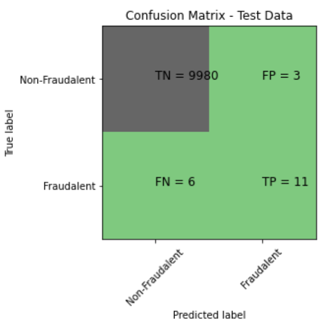
In [375]:

```python
Data_Validation_Method = "StratifiedKFold Cross Validation"
#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_Validation_Method, X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

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

#Run 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_Validation_Method,X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*60 )

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

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

#Run SVM Model with Sigmoid Kernel
print("SVM Model with Sigmoid Kernel")
start_time = time.time()
df_Results = buildAndRunSVMModels(df_Results,Data_Validation_Method,X_train_SKF_cv,y_train_SKF_cv, X_test_SKF_cv, y_test_SKF_cv)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))

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

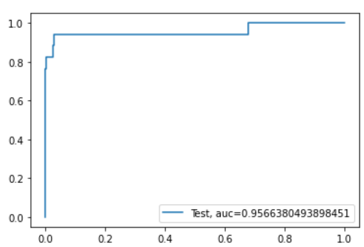
```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.955610831614236
Max auc_roc for l2: 0.9567308642474954
Parameters for l1 regularisations
[[-0.07151745 -0.15407323 -0.19254832  0.06567634 -0.28698825  0.09795102
   0.14259043 -0.03449754 -0.15403987 -0.14443599 -0.03359353  0.00725298
  -0.12649247 -0.23510622  0.00940845 -0.11324575 -0.24276231  0.03651103
   0.0190662   0.27380617  0.08747869 -0.02202149  0.02427419 -0.00966349
  -0.00172041  0.01093092 -0.08807507  0.03092445 -0.00671194 -0.11568803]]
[-2.07376685]
{1: array([[0.5       , 0.5       , 0.5       , 0.5       , 0.39711779,
        0.43189223, 0.50948622, 0.50324561, 0.77631579, 0.86656642,
        0.8173183 , 0.82373434, 0.870401  , 0.8416792 , 0.8585213 ,
        0.86591479, 0.82308271, 0.85328321, 0.89037594, 0.95275689],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.52134452,
        0.52622684, 0.56920798, 0.5447129 , 0.94207979, 0.9265565 ,
        0.92376064, 0.92568019, 0.91758471, 0.91387081, 0.91967117,
        0.93561175, 0.92810048, 0.92029711, 0.93093807, 0.91420464],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.55931809,
        0.54017745, 0.54447068, 0.52983793, 0.96375801, 0.98400773,
        0.97255912, 0.97474151, 0.98762119, 0.98858717, 0.98926693,
        0.97499195, 0.97009052, 0.98304175, 0.99116311, 0.9745984 ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.49652964,
        0.45758649, 0.46560052, 0.49200386, 0.8555329 , 0.87907409,
        0.83846732, 0.84680333, 0.87166828, 0.87986119, 0.82086509,
        0.87073808, 0.86487067, 0.86741083, 0.75253122, 0.88590748],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.11809953,
        0.11201746, 0.11995993, 0.10768845, 0.62305463, 0.87928876,
        0.58473758, 0.88537083, 0.91563808, 0.90100533, 0.79120604,
        0.89395728, 0.83735823, 0.91596007, 0.63568388, 0.98772852],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.55930611,
        0.54069076, 0.57469627, 0.53245554, 0.83263402, 0.86253758,
        0.82518161, 0.84030561, 0.89237851, 0.86761022, 0.89153307,
        0.91223071, 0.82411698, 0.87230711, 0.90866107, 0.99120115],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.50397272,
        0.46468343, 0.50416041, 0.5175488 , 0.86161161, 0.85172673,
        0.76682933, 0.82695195, 0.85441692, 0.82100851, 0.81594094,
        0.82088338, 0.79416917, 0.85372873, 0.84334334, 0.89239239],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.65168293,
        0.57929805, 0.43233859, 0.48082457, 0.88613614, 0.88951451,
        0.87230981, 0.88807558, 0.92210961, 0.90102603, 0.91798048,
        0.90496747, 0.87731481, 0.9115991 , 0.91535285, 0.9634009 ],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.48923115,
        0.43170191, 0.48686988, 0.44959035, 0.97323888, 0.96848056,
        0.95209474, 0.95506422, 0.98229044, 0.97030518, 0.98153912,
        0.98264821, 0.94941147, 0.97864119, 0.98619012, 0.99398948],
       [0.5       , 0.5       , 0.5       , 0.5       , 0.83521162,
        0.74782655, 0.66539659, 0.67723874, 0.99774606, 0.99678008,
        0.98479482, 0.99599299, 0.99992845, 0.99685163, 0.99996422,
        0.99992845, 0.99570677, 0.99978534, 0.99982112, 0.99992845]])}
Parameters for l2 regularisations
[[-1.31491217e-02  3.07463695e-02 -1.23758286e-01  2.90976907e-01
   1.00763702e-01 -5.25175254e-02  4.50978185e-02 -1.31263201e-01
  -1.08854534e-01 -1.56738327e-01  1.83509896e-01 -2.25403516e-01
  -5.45158708e-02 -4.16368675e-01 -5.45595030e-02 -7.19437239e-02
  -1.10316217e-01 -2.06331993e-02  2.04222421e-02 -2.72144122e-02
   4.33597690e-02  6.29205995e-02  3.69527095e-02 -1.48824898e-02
  -1.39901365e-02  2.51950981e-03 -4.00624298e-03 -7.95897195e-02
   3.42612078e-04  1.20704152e-02]]
[-7.64242687]
{1: array([[0.65944862, 0.65944862, 0.66107769, 0.67919799, 0.77882206,
        0.84032581, 0.88531328, 0.92263158, 0.9356391 , 0.93924812,
        0.93924812, 0.93924812, 0.93924812, 0.93924812, 0.93924812,
        0.93924812, 0.93924812, 0.93924812, 0.93924812, 0.93924812],
       [0.55307962, 0.55307962, 0.55474879, 0.58771491, 0.73147221,
        0.84848105, 0.87819229, 0.90110165, 0.91103322, 0.89542647,
        0.89542647, 0.89542647, 0.89542647, 0.89542647, 0.89542647,
        0.89542647, 0.89542647, 0.89542647, 0.89542647, 0.89542647],
       [0.49772817, 0.49772817, 0.49898036, 0.52584881, 0.67553934,
        0.87138206, 0.94204143, 0.97273801, 0.98007227, 0.98443705,
        0.98443705, 0.98443705, 0.98443705, 0.98443705, 0.98443705,
        0.98443705, 0.98443705, 0.98443705, 0.98443705, 0.98443705],
       [0.58863726, 0.58860148, 0.59024722, 0.61833208, 0.79202891,
        0.9631498 , 0.9797145 , 0.92626382, 0.90862581, 0.91062932,
        0.91062932, 0.91062932, 0.91062932, 0.91062932, 0.91062932,
        0.91062932, 0.91062932, 0.91062932, 0.91062932, 0.91062932],
       [0.89706987, 0.89692676, 0.89774963, 0.90143465, 0.92533362,
        0.9622196 , 0.93864262, 0.95302494, 0.98339952, 0.98164645,
        0.98164645, 0.98164645, 0.98164645, 0.98164645, 0.98164645,
        0.98164645, 0.98164645, 0.98164645, 0.98164645, 0.98164645],
       [0.50973823, 0.50973823, 0.51111598, 0.53710546, 0.66846192,
        0.79950526, 0.90812876, 0.97263277, 0.96940757, 0.96157941,
        0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 ,
        0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 ],
       [0.51582833, 0.51582833, 0.51607858, 0.52152152, 0.57645145,
        0.76295045, 0.90546797, 0.90934685, 0.9288038 , 0.87825325,
        0.87825325, 0.87825325, 0.87825325, 0.87825325, 0.87825325,
        0.87825325, 0.87825325, 0.87825325, 0.87825325, 0.87825325],
       [0.41166166, 0.41166166, 0.41278779, 0.44575826, 0.6265015 ,
        0.80974725, 0.9004004 , 0.94513263, 0.95952202, 0.95658158,
        0.95658158, 0.94331832, 0.94331832, 0.94331832, 0.94331832,
        0.94331832, 0.94331832, 0.94331832, 0.94331832, 0.94331832],
       [0.554041  , 0.554041  , 0.55550785, 0.57096347, 0.69085185,
        0.88064828, 0.9584988 , 0.98533147, 0.99080534, 0.98966048,
        0.98966048, 0.98966048, 0.98966048, 0.98966048, 0.98966048,
        0.98966048, 0.98966048, 0.98966048, 0.98966048, 0.98966048],
       [0.33032807, 0.33022074, 0.33458552, 0.38728489, 0.68140675,
        0.94451004, 0.99745984, 0.99992845, 1.        , 1.        ,
        1.        , 1.        , 1.        , 1.        , 1.        ,
        1.        , 1.        , 1.        , 1.        , 1.        ]])}
Accuarcy of Logistic model with l2 regularisation : 0.9992
Confusion Matrix
```



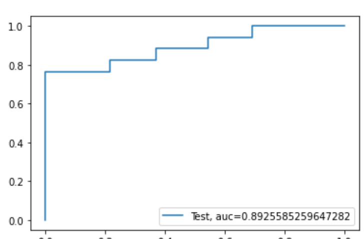

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.85      0.65      0.73        17

    accuracy                           1.00     10000
   macro avg       0.92      0.82      0.87     10000
weighted avg       1.00      1.00      1.00     10000

Accuarcy of Logistic model with l1 regularisation : 0.9991
Confusion Matrix
```



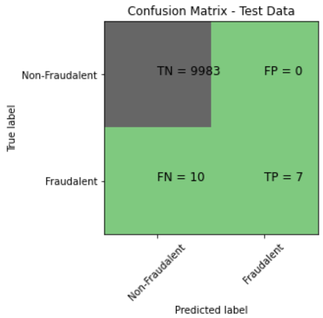

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.79      0.65      0.71        17

    accuracy                           1.00     10000
   macro avg       0.89      0.82      0.85     10000
weighted avg       1.00      1.00      1.00     10000

l2 roc_value: 0.9566380493898451
l2 threshold: 0.002112752055838854
ROC for the test dataset 95.7%
```



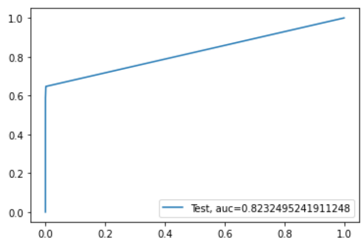

```
l1 roc_value: 0.8925585259647282
l1 threshold: 0.3014627218940176
ROC for the test dataset 89.3%
```



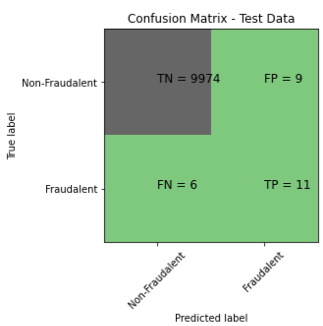

```
Time Taken by Model: --- 54.348634481430054 seconds ---
----------------------------------------------------------
KNN Model
model score
0.999
Confusion Matrix
```



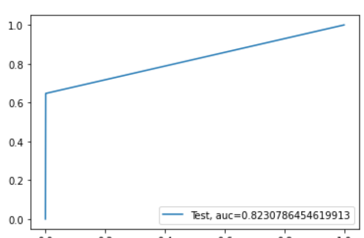

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       1.00      0.41      0.58        17

    accuracy                           1.00     10000
   macro avg       1.00      0.71      0.79     10000
weighted avg       1.00      1.00      1.00     10000

KNN roc_value: 0.8232495241911248
KNN threshold: 0.2
ROC for the test dataset 82.3%
```



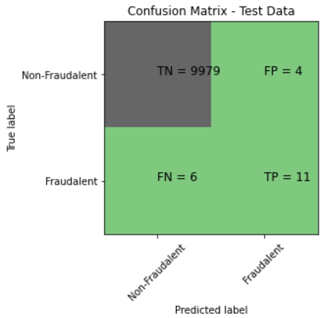

```
Time Taken by Model: --- 44.82395100593567 seconds ---
------------------------------------------------------------
Decision Tree Models with  'gini' & 'entropy' criteria
gini score: 0.9985
Confusion Matrix
```



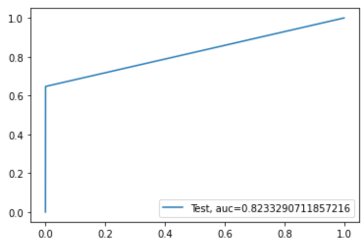

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.55      0.65      0.59        17

    accuracy                           1.00     10000
   macro avg       0.77      0.82      0.80     10000
weighted avg       1.00      1.00      1.00     10000

gini tree_roc_value: 0.8230786454619913
Tree threshold: 1.0
ROC for the test dataset 82.3%
```



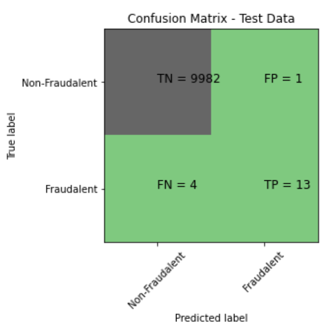

```
entropy score: 0.999
Confusion Matrix
```



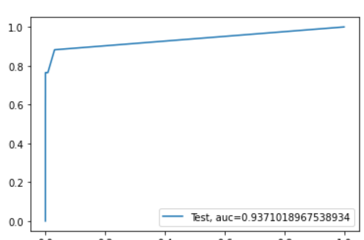

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.73      0.65      0.69        17

    accuracy                           1.00     10000
   macro avg       0.87      0.82      0.84     10000
weighted avg       1.00      1.00      1.00     10000

entropy tree_roc_value: 0.8233290711857216
Tree threshold: 1.0
ROC for the test dataset 82.3%
```



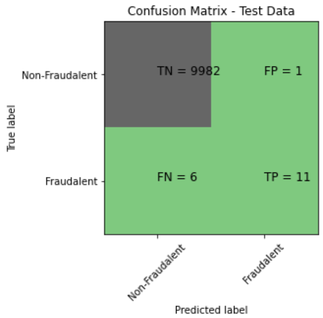

```
Time Taken by Model: --- 5.102020263671875 seconds ---
------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9995
Confusion Matrix
```



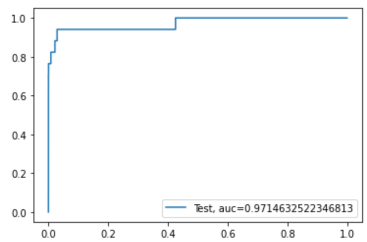

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.93      0.76      0.84        17

    accuracy                           1.00     10000
   macro avg       0.96      0.88      0.92     10000
weighted avg       1.00      1.00      1.00     10000

Random Forest roc_value: 0.9371018967538934
Random Forest threshold: 0.01
ROC for the test dataset 93.7%
```



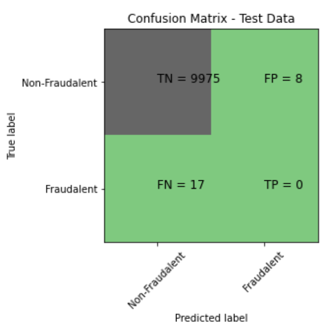

```
Time Taken by Model: --- 20.428218603134155 seconds ---
------------------------------------------------------------
XGBoost Model
Model Accuracy: 0.9993
Confusion Matrix
```



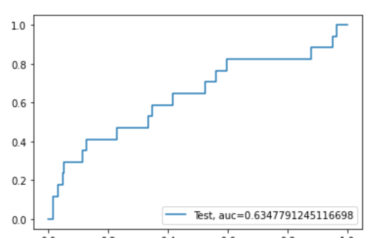

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.92      0.65      0.76        17

    accuracy                           1.00     10000
   macro avg       0.96      0.82      0.88     10000
weighted avg       1.00      1.00      1.00     10000

XGboost roc_value: 0.9714632522346813
XGBoost threshold: 0.0013952451990917325
ROC for the test dataset 97.1%
```



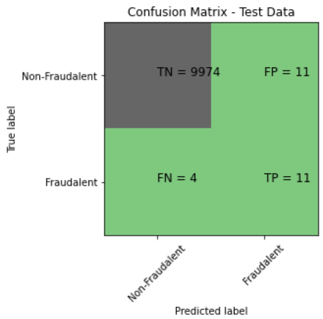

```
Time Taken by Model: --- 9.466046333312988 seconds ---
------------------------------------------------------------
SVM Model with Sigmoid Kernel
accuracy_score : 0.9975
Confusion Matrix
```



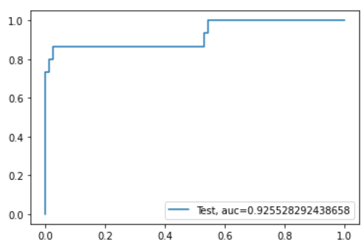

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.00      0.00      0.00        17

    accuracy                           1.00     10000
   macro avg       0.50      0.50      0.50     10000
weighted avg       1.00      1.00      1.00     10000

SVM roc_value: 0.6347791245116698
SVM threshold: 0.003424350421318629
ROC for the test dataset 63.5%
```



```
Time Taken by Model: --- 6.3848254680633545 seconds ---
ANN Model
Epoch 1/5
2130/2130 [==============================] - 7s 3ms/step - loss: 0.1839 - accuracy: 0.9428
Epoch 2/5
2130/2130 [==============================] - 5s 2ms/step - loss: 0.0361 - accuracy: 0.9894
Epoch 3/5
2130/2130 [==============================] - 7s 3ms/step - loss: 0.0295 - accuracy: 0.9924
Epoch 4/5
2130/2130 [==============================] - 5s 2ms/step - loss: 0.0243 - accuracy: 0.9939
Epoch 5/5
2130/2130 [==============================] - 5s 2ms/step - loss: 0.0201 - accuracy: 0.9951
999/999 [==============================] - 4s 4ms/step - loss: 0.0120 - accuracy: 0.9970
313/313 [==============================] - 1s 2ms/step
accuracy_score : 0.9985
Confusion Matrix
```




```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9985
           1       0.50      0.73      0.59        15

    accuracy                           1.00     10000
   macro avg       0.75      0.87      0.80     10000
weighted avg       1.00      1.00      1.00     10000

ANN roc_value: 0.925528292438658
ANN threshold: 7.655112858628854e-05
ROC for the test dataset 92.6%
```




```
Time Taken by Model: --- 48.55423426628113 seconds ---
--------------------------------------------------------------------------------
```

In [376]:
```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Out[376]:

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9990	0.950886	0.030690
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.933580	0.057833
2	RepeatedKFold Cross Validation	KNN	0.9990	0.866296	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.9989	0.833033	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.9992	0.899750	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.9997	0.965381	0.300000
6	RepeatedKFold Cross Validation	XGBoost	0.9996	0.936365	0.283122
7	RepeatedKFold Cross Validation	SVM	0.9981	0.724487	0.003482
8	RepeatedKFold Cross Validation	ANN	0.9984	0.927278	0.000521
9	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9992	0.956638	0.002113
10	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.892559	0.301463
11	StratifiedKFold Cross Validation	KNN	0.9990	0.823250	0.200000
12	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.9985	0.823079	1.000000
13	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.9990	0.823329	1.000000
14	StratifiedKFold Cross Validation	Random Forest	0.9995	0.937102	0.010000
15	StratifiedKFold Cross Validation	XGBoost	0.9993	0.971463	0.001395
16	StratifiedKFold Cross Validation	SVM	0.9975	0.634779	0.003424
17	StratifiedKFold Cross Validation	ANN	0.9985	0.925528	0.000077

Results for cross validation with StratifiedKFold:

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

Conclusion:

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

Proceed with the model which shows the best result

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

In [378]:
```
# Logistic Regression
from sklearn import linear_model #import the package
from sklearn.model_selection import KFold

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

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

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


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


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


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


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


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


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

```
Max auc_roc for l2: 0.9567308642474954
Parameters for l2 regularisations
[[-1.31491217e-02  3.07463695e-02 -1.23758286e-01  2.90976907e-01
   1.00763702e-01 -5.25175254e-02  4.50978185e-02 -1.31263201e-01
  -1.08854534e-01 -1.56738327e-01  1.83509896e-01 -2.25403516e-01
  -5.45158708e-02 -4.16368675e-01 -5.45595030e-02 -7.19437239e-02
  -1.10316217e-01 -2.06331993e-02  2.04222421e-02 -2.72144122e-02
   4.33597690e-02  6.29205995e-02  3.69527095e-02 -1.48824898e-02
  -1.39901365e-02  2.51950981e-03 -4.00624298e-03 -7.95897195e-02
   3.42612078e-04  1.20704152e-02]]
[-7.64242687]
{1: array([[0.65944862, 0.65944862, 0.66107769, 0.67919799, 0.77882206,
        0.84032581, 0.88531328, 0.92263158, 0.9356391 , 0.93924812,
        0.93924812, 0.93924812, 0.93924812, 0.93924812, 0.93924812,
        0.93924812, 0.93924812, 0.93924812, 0.93924812, 0.93924812],
       [0.55307962, 0.55307962, 0.55474879, 0.58771491, 0.73147221,
        0.84848105, 0.87819229, 0.90110165, 0.91103322, 0.89542647,
        0.89542647, 0.89542647, 0.89542647, 0.89542647, 0.89542647,
        0.89542647, 0.89542647, 0.89542647, 0.89542647, 0.89542647],
       [0.49772817, 0.49772817, 0.49898036, 0.52584881, 0.67553934,
        0.87138206, 0.94204143, 0.97273801, 0.98007227, 0.98443705,
        0.98443705, 0.98443705, 0.98443705, 0.98443705, 0.98443705,
        0.98443705, 0.98443705, 0.98443705, 0.98443705, 0.98443705],
       [0.58863726, 0.58860148, 0.59024722, 0.61833208, 0.79202891,
        0.9631498 , 0.9797145 , 0.92626382, 0.90862581, 0.91062932,
        0.91062932, 0.91062932, 0.91062932, 0.91062932, 0.91062932,
        0.91062932, 0.91062932, 0.91062932, 0.91062932, 0.91062932],
       [0.89706987, 0.89692676, 0.89774963, 0.90143465, 0.92533362,
        0.9622196 , 0.93864262, 0.95302494, 0.98339952, 0.98164645,
        0.98164645, 0.98164645, 0.98164645, 0.98164645, 0.98164645,
        0.98164645, 0.98164645, 0.98164645, 0.98164645, 0.98164645],
       [0.50973823, 0.50973823, 0.51111598, 0.53710546, 0.66846192,
        0.79950526, 0.90812876, 0.97263277, 0.96940757, 0.96157941,
        0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 ,
        0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 , 0.9332728 ],
       [0.51582833, 0.51582833, 0.51607858, 0.52152152, 0.57645145,
        0.76295045, 0.90546797, 0.90934685, 0.9288038 , 0.87825325,
        0.87825325, 0.87825325, 0.87825325, 0.87825325, 0.87825325,
        0.87825325, 0.87825325, 0.87825325, 0.87825325, 0.87825325],
       [0.41166166, 0.41166166, 0.41278779, 0.44575826, 0.6265015 ,
        0.80974725, 0.9004004 , 0.94513263, 0.95952202, 0.95658158,
        0.95658158, 0.94331832, 0.94331832, 0.94331832, 0.94331832,
        0.94331832, 0.94331832, 0.94331832, 0.94331832, 0.94331832],
       [0.554041  , 0.554041  , 0.55550785, 0.57096347, 0.69085185,
        0.88064828, 0.9584988 , 0.98533147, 0.99080534, 0.98966048,
        0.98966048, 0.98966048, 0.98966048, 0.98966048, 0.98966048,
        0.98966048, 0.98966048, 0.98966048, 0.98966048, 0.98966048],
       [0.33032807, 0.33022074, 0.33458552, 0.38728489, 0.68140675,
        0.94451004, 0.99745984, 0.99992845, 1.        , 1.        ,
        1.        , 1.        , 1.        , 1.        , 1.        ,
        1.        , 1.        , 1.        , 1.        , 1.        ]])}
Accuarcy of Logistic model with l2 regularisation : 0.9995
l2 roc_value: 0.9785411450509097
l2 threshold: 0.008174170065796222
```

In [379]:
```
# Checking for the coefficient values
clf.coef_
```

Out[379]:
```
array([[-1.31491217e-02,  3.07463695e-02, -1.23758286e-01,
         2.90976907e-01,  1.00763702e-01, -5.25175254e-02,
         4.50978185e-02, -1.31263201e-01, -1.08854534e-01,
        -1.56738327e-01,  1.83509896e-01, -2.25403516e-01,
        -5.45158708e-02, -4.16368675e-01, -5.45595030e-02,
        -7.19437239e-02, -1.10316217e-01, -2.06331993e-02,
         2.04222421e-02, -2.72144122e-02,  4.33597690e-02,
         6.29205995e-02,  3.69527095e-02, -1.48824898e-02,
        -1.39901365e-02,  2.51950981e-03, -4.00624298e-03,
        -7.95897195e-02,  3.42612078e-04,  1.20704152e-02]])
```

In [380]:
```
# Creating a dataframe with the coefficient values
coefficients = pd.concat([pd.DataFrame(X.columns),pd.DataFrame(np.transpose(clf.coef_))], axis = 1)
coefficients.columns = ['Feature','Importance Coefficient']
```

In [381]:
```
coefficients
```

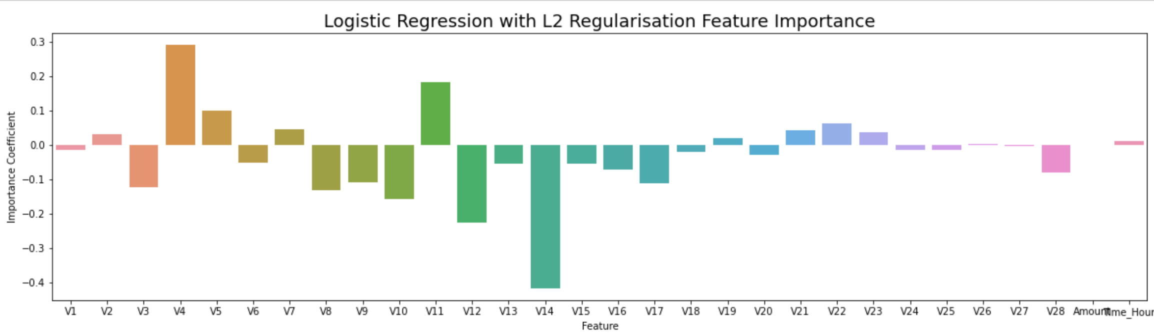
Out[381]:

	Feature	Importance Coefficient
0	V1	-0.013149
1	V2	0.030746
2	V3	-0.123758
3	V4	0.290977
4	V5	0.100764
5	V6	-0.052518
6	V7	0.045098
7	V8	-0.131263
8	V9	-0.108855
9	V10	-0.156738
10	V11	0.183510
11	V12	-0.225404
12	V13	-0.054516
13	V14	-0.416369
14	V15	-0.054560
15	V16	-0.071944
16	V17	-0.110316
17	V18	-0.020633
18	V19	0.020422
19	V20	-0.027214
20	V21	0.043360
21	V22	0.062921
22	V23	0.036953
23	V24	-0.014882
24	V25	-0.013990
25	V26	0.002520
26	V27	-0.004006
27	V28	-0.079590
28	Amount	0.000343
29	Time_Hour	0.012070

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

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

In [382]:
```
# Plotting the coefficient values
plt.figure(figsize=(20,5))
sns.barplot(x='Feature', y='Importance Coefficient', data=coefficients)
plt.title("Logistic Regression with L2 Regularisation Feature Importance", fontsize=18)

plt.show()
```



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

Model building with balancing Classes

Perform class balancing with:

- Random Oversampling
- SMOTE
- ADASYN

Oversampling with RandomOverSampler with StratifiedKFold Cross Validation

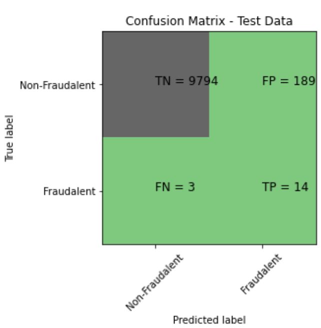
- **Random Oversampling:**
  - Random Oversampling is a technique for handling imbalanced datasets by duplicating instances from the minority class to balance the class distribution.
  - This method increases the number of samples in the minority class to compensate for the class imbalance, but it can also lead to overfitting, as it increases the likelihood of duplicating instances in the training set.

In [383]:
```
# Creating the dataset with RandomOverSampler and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn.over_sampling import RandomOverSampler

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

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

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

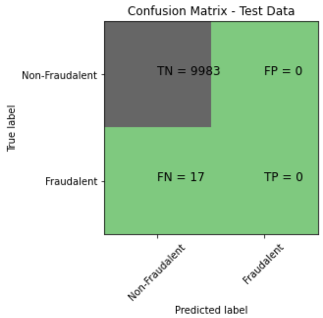
In [384]:

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

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

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

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

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

#Run ANN Model
print("ANN Model")
start_time = time.time()
df_Results = buildAndgetmetricsANN(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 )
```

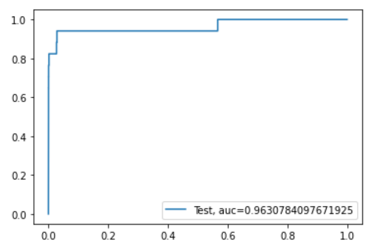
```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.9902811135417251
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]])}
Parameters for l2 regularisations
[[ 1.00287017  1.75499432  0.65939176  0.96551128  1.33001016 -0.68910098
  -0.41794128  0.10929295  0.32371588 -1.23855061  1.0777743  -0.84159353
  -0.27422317 -1.78960219  0.09829525  0.81281043 -1.39714535 -1.7745295
   1.70662517 -1.0846819  -1.21053154  1.11054779  1.62081414 -0.01230903
   1.13104885 -1.04185668  1.09879398  1.8527709   0.01426082  0.04171269]]
[-6.89355115]
{1: array([[0.56755895, 0.59439814, 0.70231474, 0.85220335, 0.93222631,
        0.9652848 , 0.9768219 , 0.98400971, 0.98808398, 0.98903956,
        0.98921246, 0.98921246, 0.98921246, 0.98921246, 0.98921246,
        0.98921246, 0.98921246, 0.98921246, 0.98921246, 0.98921246],
       [0.58162639, 0.60675551, 0.71086415, 0.85593627, 0.93524667,
        0.96812634, 0.97844339, 0.98484704, 0.98810019, 0.9892657 ,
        0.98961873, 0.98961873, 0.98961873, 0.98961873, 0.98961873,
        0.98961873, 0.98961873, 0.98961873, 0.98961873, 0.98961873],
       [0.5719322 , 0.59882317, 0.70644911, 0.85580391, 0.93290425,
        0.96544273, 0.97584737, 0.98319154, 0.98783122, 0.98921468,
        0.98960152, 0.98960152, 0.98960152, 0.98960152, 0.98960152,
        0.98960152, 0.98960152, 0.98960152, 0.98960152, 0.98960152],
       [0.57853633, 0.6049119 , 0.71489173, 0.86736679, 0.94294924,
        0.97152042, 0.97939392, 0.98591272, 0.98991296, 0.9908403 ,
        0.99106778, 0.99106778, 0.99106778, 0.99106778, 0.99106778,
        0.99106778, 0.99106778, 0.99106778, 0.99106778, 0.99106778],
       [0.56777214, 0.59413411, 0.70595942, 0.86185744, 0.93957544,
        0.97116118, 0.9807    , 0.98671759, 0.99016801, 0.99113981,
        0.99142011, 0.99142011, 0.99142011, 0.99142011, 0.99142011,
        0.99142011, 0.99142011, 0.99142011, 0.99142011, 0.99142011],
       [0.56345782, 0.58730994, 0.69175087, 0.84649804, 0.92702125,
        0.96190758, 0.97405815, 0.98204158, 0.98657427, 0.98799529,
        0.98847302, 0.98850781, 0.98850781, 0.98850781, 0.98850781,
        0.98850781, 0.98850781, 0.98850781, 0.98850781, 0.98850781],
       [0.5724701 , 0.59923691, 0.70951807, 0.86245509, 0.93679318,
        0.96840294, 0.97791177, 0.98501817, 0.98946752, 0.99076735,
        0.99100381, 0.99100381, 0.99100381, 0.99100381, 0.99100381,
        0.99100381, 0.99100381, 0.99100381, 0.99100381, 0.99100381],
       [0.56412245, 0.59072964, 0.70228613, 0.85572542, 0.93379845,
        0.96823649, 0.97935214, 0.98627608, 0.98955185, 0.99062136,
        0.99091057, 0.99091057, 0.99091057, 0.99091057, 0.99091057,
        0.99091057, 0.99091057, 0.99091057, 0.99091057, 0.99091057],
       [0.55289919, 0.58075621, 0.69678641, 0.85645818, 0.93641018,
        0.96878613, 0.97901316, 0.98563259, 0.98928921, 0.99040971,
        0.9906853 , 0.9906853 , 0.9906853 , 0.9906853 , 0.9906853 ,
        0.9906853 , 0.9906853 , 0.9906853 , 0.9906853 , 0.9906853 ],
       [0.56355412, 0.59193147, 0.70231962, 0.85884523, 0.93792208,
        0.97001416, 0.97828892, 0.98478536, 0.98906349, 0.99048169,
        0.99078304, 0.99078304, 0.99078304, 0.99078304, 0.99078304,
        0.99078304, 0.99078304, 0.99078304, 0.99078304, 0.99078304]])}
Accuarcy of Logistic model with l2 regularisation : 0.9808
Confusion Matrix
```



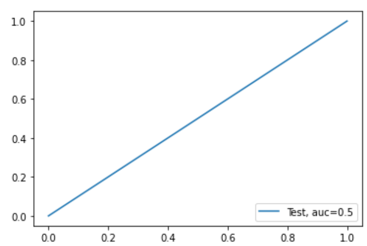

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.98      0.99      9983
           1       0.07      0.82      0.13        17

    accuracy                           0.98     10000
   macro avg       0.53      0.90      0.56     10000
weighted avg       1.00      0.98      0.99     10000

Accuarcy of Logistic model with l1 regularisation : 0.9983
Confusion Matrix
```



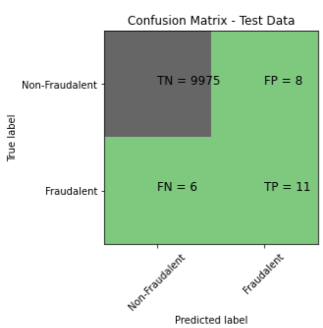

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.00      0.00      0.00        17

    accuracy                           1.00     10000
   macro avg       0.50      0.50      0.50     10000
weighted avg       1.00      1.00      1.00     10000

l2 roc_value: 0.9630784097671925
l2 threshold: 0.3837371155321852
ROC for the test dataset 96.3%
```



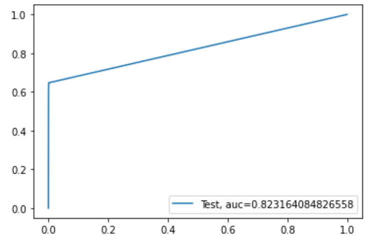

```
l1 roc_value: 0.5
l1 threshold: 1.5
ROC for the test dataset 50.0%
```




```
Time Taken by Model: --- 99.70262217521667 seconds ---
------------------------------------------------------------
KNN Model
model score
0.9986
Confusion Matrix
```

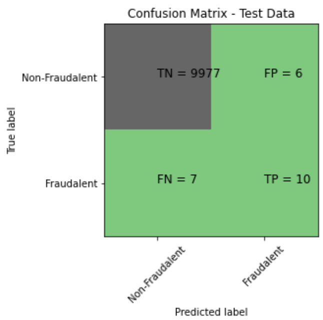



```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.58      0.65      0.61        17

    accuracy                           1.00     10000
   macro avg       0.79      0.82      0.81     10000
weighted avg       1.00      1.00      1.00     10000

KNN roc_value: 0.823164084826558
KNN threshold: 0.6
ROC for the test dataset 82.3%
```



Time Taken by Model: --- 76.98786473274231 seconds ---

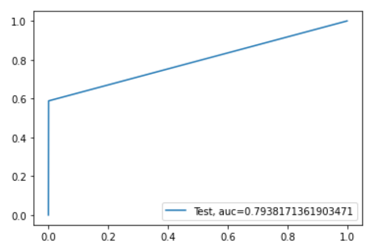
Decision Tree Models with 'gini' & 'entropy' criteria
gini score: 0.9987
Confusion Matrix



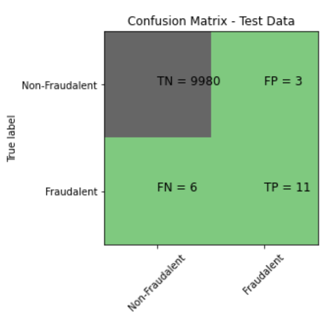
classification Report

	precision	recall	f1-score	support
0	1.00	1.00	1.00	9983
1	0.62	0.59	0.61	17
accuracy			1.00	10000
macro avg	0.81	0.79	0.80	10000
weighted avg	1.00	1.00	1.00	10000

gini tree_roc_value: 0.7938171361903471
Tree threshold: 1.0
ROC for the test dataset 79.4%



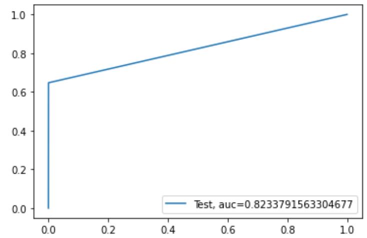
entropy score: 0.9991
Confusion Matrix



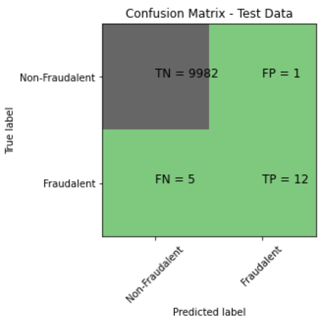
classification Report

	precision	recall	f1-score	support
0	1.00	1.00	1.00	9983
1	0.79	0.65	0.71	17
accuracy			1.00	10000
macro avg	0.89	0.82	0.85	10000
weighted avg	1.00	1.00	1.00	10000

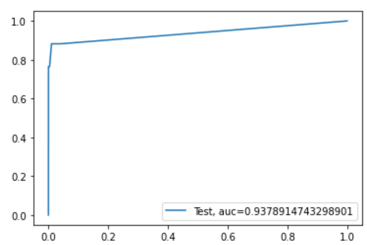
entropy tree_roc_value: 0.8233791563304677
Tree threshold: 1.0
ROC for the test dataset 82.3%



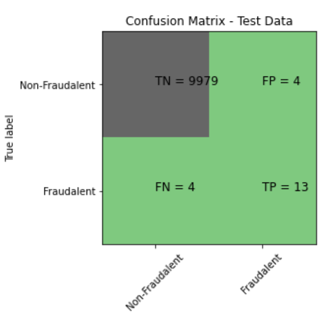

```
Time Taken by Model: --- 2.96124267578125 seconds ---
------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9994
Confusion Matrix
```



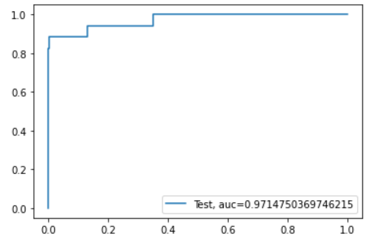

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.92      0.71      0.80        17

    accuracy                           1.00     10000
   macro avg       0.96      0.85      0.90     10000
weighted avg       1.00      1.00      1.00     10000

Random Forest roc_value: 0.9378914743298901
Random Forest threshold: 0.02
ROC for the test dataset 93.8%
```



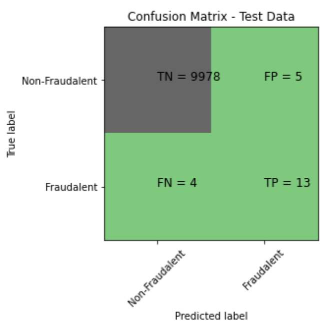

```
Time Taken by Model: --- 17.280887126922607 seconds ---
------------------------------------------------------------
XGBoost Model
Model Accuracy: 0.9992
Confusion Matrix
```



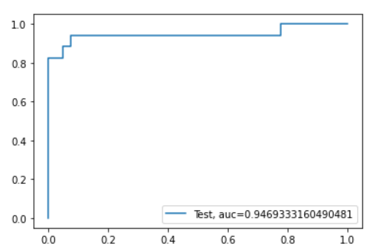

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.76      0.76      0.76        17

    accuracy                           1.00     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       1.00      1.00      1.00     10000

XGboost roc_value: 0.9714750369746215
XGBoost threshold: 0.13544505834579468
ROC for the test dataset 97.1%
```




```
Time Taken by Model: --- 11.824404954910278 seconds ---
----------------------------------------------------------
ANN Model
Epoch 1/5
2130/2130 [==============================] - 7s 3ms/step - loss: 0.1501 - accuracy: 0.9519
Epoch 2/5
2130/2130 [==============================] - 6s 3ms/step - loss: 0.0373 - accuracy: 0.9907
Epoch 3/5
2130/2130 [==============================] - 5s 3ms/step - loss: 0.0187 - accuracy: 0.9945
Epoch 4/5
2130/2130 [==============================] - 9s 4ms/step - loss: 0.0177 - accuracy: 0.9959
Epoch 5/5
2130/2130 [==============================] - 5s 3ms/step - loss: 0.0199 - accuracy: 0.9953
999/999 [==============================] - 4s 4ms/step - loss: 0.0079 - accuracy: 0.9978
313/313 [==============================] - 1s 2ms/step
accuracy_score : 0.9991
Confusion Matrix
```




```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.72      0.76      0.74        17

    accuracy                           1.00     10000
   macro avg       0.86      0.88      0.87     10000
weighted avg       1.00      1.00      1.00     10000

ANN roc_value: 0.9469333160490481
ANN threshold: 7.360094514297089e-06
ROC for the test dataset 94.7%
```




```
Time Taken by Model: --- 38.464799880981445 seconds ---
--------------------------------------------------------------------------------
```

In [385]:
```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Out[385]:

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9990	0.950886	0.030690
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.933580	0.057833
2	RepeatedKFold Cross Validation	KNN	0.9990	0.866296	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.9989	0.833033	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.9992	0.899750	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.9997	0.965381	0.300000
6	RepeatedKFold Cross Validation	XGBoost	0.9996	0.936365	0.283122
7	RepeatedKFold Cross Validation	SVM	0.9981	0.724487	0.003482
8	RepeatedKFold Cross Validation	ANN	0.9984	0.927278	0.000521
9	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9992	0.956638	0.002113
10	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.892559	0.301463
11	StratifiedKFold Cross Validation	KNN	0.9990	0.823250	0.200000
12	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.9985	0.823079	1.000000
13	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.9990	0.823329	1.000000
14	StratifiedKFold Cross Validation	Random Forest	0.9995	0.937102	0.010000
15	StratifiedKFold Cross Validation	XGBoost	0.9993	0.971463	0.001395
16	StratifiedKFold Cross Validation	SVM	0.9975	0.634779	0.003424
17	StratifiedKFold Cross Validation	ANN	0.9985	0.925528	0.000077
18	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9808	0.963078	0.383737
19	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
20	Random Oversampling with StratifiedKFold CV	KNN	0.9986	0.823164	0.600000
21	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9987	0.793817	1.000000
22	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9991	0.823379	1.000000
23	Random Oversampling with StratifiedKFold CV	Random Forest	0.9994	0.937891	0.020000
24	Random Oversampling with StratifiedKFold CV	XGBoost	0.9992	0.971475	0.135445
25	Random Oversampling with StratifiedKFold CV	ANN	0.9991	0.946933	0.000007

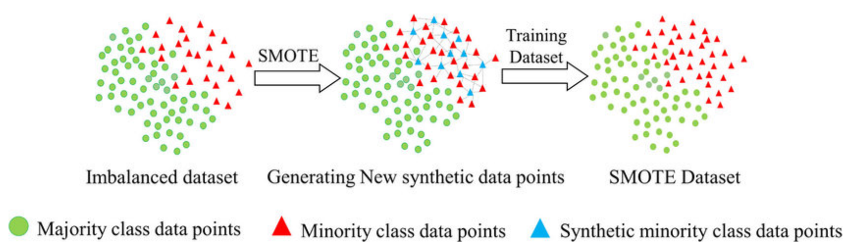
Results for Random Oversampling with StratifiedKFold technique:

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

Oversampling with SMOTE Oversampling

SMOTE (Synthetic Minority Over-sampling Technique):

- SMOTE (Synthetic Minority Over-sampling Technique) is a data augmentation technique for handling imbalanced datasets by creating synthetic samples for the minority class.
- SMOTE generates new instances for the minority class by interpolating between existing instances and their nearest neighbors, creating a more diverse representation of the minority class. This can help balance the class distribution and prevent overfitting due to random oversampling.

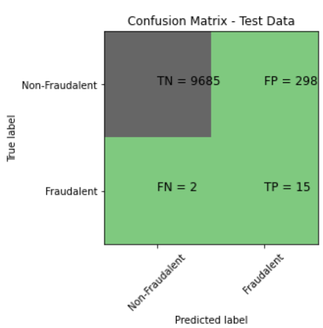


In [386]:
```
# Creating dataframe with Smote and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

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

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

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

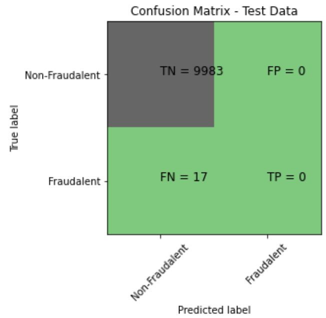
In [387]:

```python
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_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

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


#Run ANN Model
print("ANN Model")
start_time = time.time()
df_Results = buildAndgetmetricsANN(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 )
```

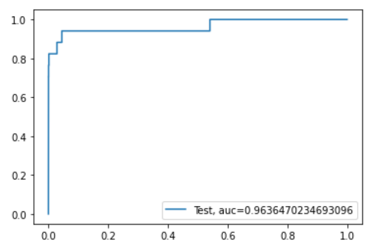
```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.9937543557426121
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]])}
Parameters for l2 regularisations
[[ 1.42022339  2.31293461  0.87060762  0.96375704  1.66493635 -0.96429402
  -1.05744597  0.34854345 -0.05949724 -1.21892002  1.52125225 -1.13556078
  -0.31224447 -2.56147413  0.22618059  0.34005457 -2.21259453 -1.77331137
   1.82057226 -1.51642205 -1.72622855  1.0738567   2.27934189 -0.01970136
   1.45285904 -0.89588821  1.67509565  2.2167778   0.01891426 -0.01176203]]
[-6.60697538]
{1: array([[0.56781703, 0.59989521, 0.71421849, 0.86160592, 0.94446916,
        0.97688395, 0.98577365, 0.99029306, 0.99234684, 0.99303961,
        0.99331294, 0.99331294, 0.99331294, 0.99331294, 0.99331294,
        0.99331294, 0.99331294, 0.99331294, 0.99331294, 0.99331294],
       [0.56179403, 0.59138159, 0.7020747 , 0.85222565, 0.94080907,
        0.97643313, 0.98573697, 0.99018033, 0.99218814, 0.9927833 ,
        0.99300691, 0.99300691, 0.99300691, 0.99300691, 0.99300691,
        0.99300691, 0.99300691, 0.99300691, 0.99300691, 0.99300691],
       [0.5615875 , 0.59208966, 0.70789042, 0.85740609, 0.94212019,
        0.9760304 , 0.98546705, 0.99014098, 0.99236637, 0.99329045,
        0.99363513, 0.99363513, 0.99363513, 0.99363513, 0.99363513,
        0.99363513, 0.99363513, 0.99363513, 0.99363513, 0.99363513],
       [0.57040099, 0.60196825, 0.71770209, 0.86470113, 0.94664147,
        0.97856684, 0.9872015 , 0.99169284, 0.99378062, 0.99431836,
        0.99449255, 0.99449255, 0.99449255, 0.99449255, 0.99449255,
        0.99449255, 0.99449255, 0.99449255, 0.99449255, 0.99449255],
       [0.56157549, 0.59339155, 0.70828357, 0.86131921, 0.94592051,
        0.97897883, 0.98737894, 0.9912704 , 0.99322964, 0.99389037,
        0.9940955 , 0.9940955 , 0.9940955 , 0.9940955 , 0.9940955 ,
        0.9940955 , 0.9940955 , 0.9940955 , 0.9940955 , 0.9940955 ],
       [0.56459132, 0.59619432, 0.71266198, 0.86294691, 0.94710641,
        0.97924038, 0.98767956, 0.99148251, 0.99322142, 0.99381655,
        0.99397406, 0.99397406, 0.99397406, 0.99397406, 0.99397406,
        0.99397406, 0.99397406, 0.99397406, 0.99397406, 0.99397406],
       [0.56548199, 0.59541412, 0.70664802, 0.85788411, 0.94246045,
        0.976799  , 0.98550238, 0.98972733, 0.99182258, 0.99249777,
        0.99286472, 0.99286472, 0.99286472, 0.99286472, 0.99286472,
        0.99286472, 0.99286472, 0.99286472, 0.99286472, 0.99286472],
       [0.57365586, 0.60570742, 0.7224519 , 0.87046444, 0.94909652,
        0.97921938, 0.98736336, 0.99143554, 0.99339039, 0.99390258,
        0.99412327, 0.99412327, 0.99412327, 0.99412327, 0.99412327,
        0.99412327, 0.99412327, 0.99412327, 0.99412327, 0.99412327],
       [0.55882703, 0.59033484, 0.70627597, 0.85769342, 0.94177696,
        0.97678983, 0.98576234, 0.9903384 , 0.99247387, 0.99323723,
        0.99350328, 0.99350328, 0.99350328, 0.99350328, 0.99350328,
        0.99350328, 0.99350328, 0.99350328, 0.99350328, 0.99350328],
       [0.5675161 , 0.5992262 , 0.71100345, 0.86077611, 0.94418097,
        0.977928  , 0.98688479, 0.99147679, 0.99367588, 0.99428946,
        0.99453519, 0.99453519, 0.99453519, 0.99453519, 0.99453519,
        0.99453519, 0.99453519, 0.99453519, 0.99453519, 0.99453519]])}
Accuarcy of Logistic model with l2 regularisation : 0.97
Confusion Matrix
```



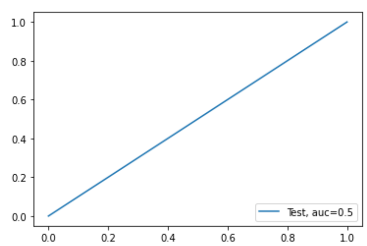

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.97      0.98      9983
           1       0.05      0.88      0.09        17

    accuracy                           0.97     10000
   macro avg       0.52      0.93      0.54     10000
weighted avg       1.00      0.97      0.98     10000

Accuarcy of Logistic model with l1 regularisation : 0.9983
Confusion Matrix
```



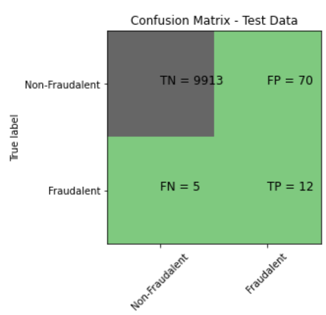

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.00      0.00      0.00        17

    accuracy                           1.00     10000
   macro avg       0.50      0.50      0.50     10000
weighted avg       1.00      1.00      1.00     10000

l2 roc_value: 0.9636470234693096
l2 threshold: 0.3707247623650453
ROC for the test dataset 96.4%
```



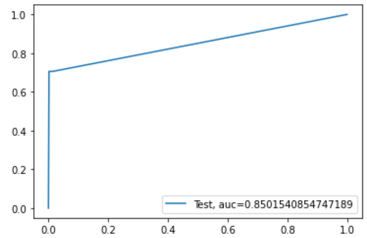

```
l1 roc_value: 0.5
l1 threshold: 1.5
ROC for the test dataset 50.0%
```




```
Time Taken by Model: --- 118.28125309944153 seconds ---
--------------------------------------------------------------------------------
KNN Model
model score
0.9925
Confusion Matrix
```

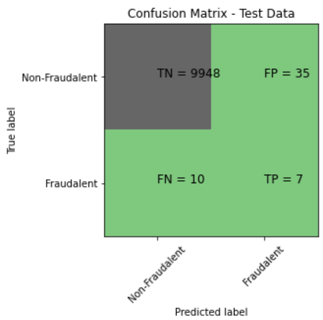



```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9983
           1       0.15      0.71      0.24        17

    accuracy                           0.99     10000
   macro avg       0.57      0.85      0.62     10000
weighted avg       1.00      0.99      0.99     10000

KNN roc_value: 0.8501540854747189
KNN threshold: 1.0
ROC for the test dataset 85.0%
```



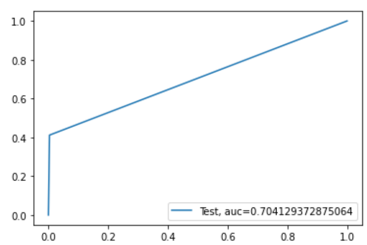
Time Taken by Model: --- 106.98998665809631 seconds ---

Decision Tree Models with 'gini' & 'entropy' criteria
gini score: 0.9955
Confusion Matrix

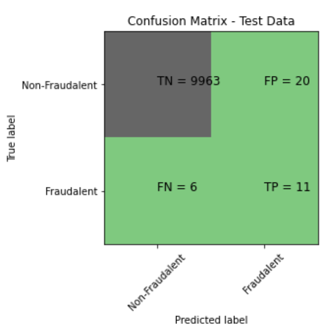



```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.17      0.41      0.24        17

    accuracy                           1.00     10000
   macro avg       0.58      0.70      0.62     10000
weighted avg       1.00      1.00      1.00     10000
```

gini tree_roc_value: 0.704129372875064
Tree threshold: 1.0
ROC for the test dataset 70.4%

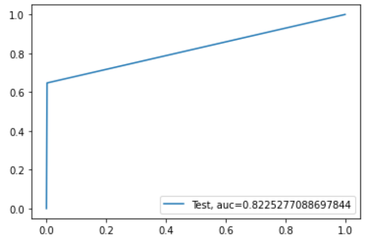


entropy score: 0.9974
Confusion Matrix

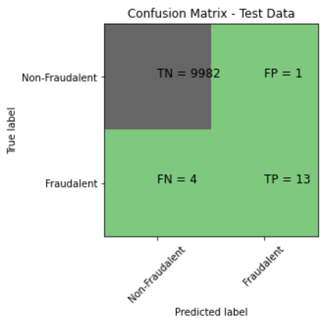



```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.35      0.65      0.46        17

    accuracy                           1.00     10000
   macro avg       0.68      0.82      0.73     10000
weighted avg       1.00      1.00      1.00     10000
```

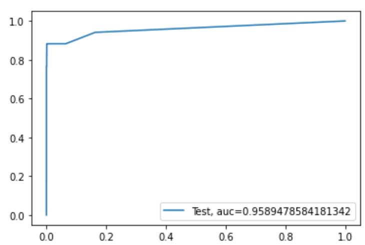
entropy tree_roc_value: 0.8225277088697844
Tree threshold: 1.0
ROC for the test dataset 82.3%



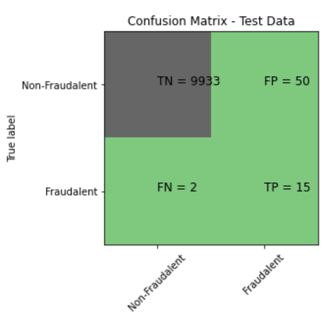

```
Time Taken by Model: --- 9.980096578598022 seconds ---
------------------------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9995
Confusion Matrix
```



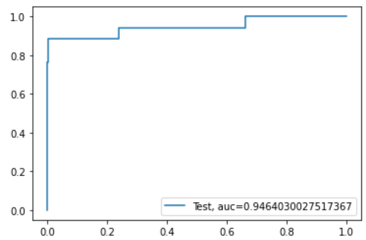

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.93      0.76      0.84        17

    accuracy                           1.00     10000
   macro avg       0.96      0.88      0.92     10000
weighted avg       1.00      1.00      1.00     10000

Random Forest roc_value: 0.9589478584181342
Random Forest threshold: 0.18
ROC for the test dataset 95.9%
```



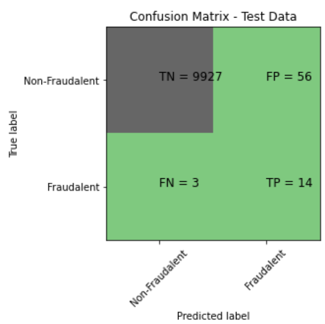

```
Time Taken by Model: --- 43.73682975769043 seconds ---
------------------------------------------------------------------------------
XGBoost Model
Model Accuracy: 0.9948
Confusion Matrix
```



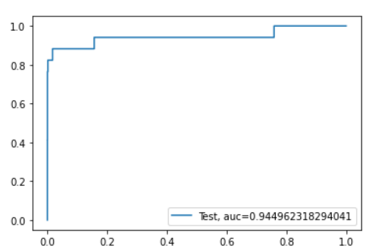

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9983
           1       0.23      0.88      0.37        17

    accuracy                           0.99     10000
   macro avg       0.62      0.94      0.68     10000
weighted avg       1.00      0.99      1.00     10000

XGboost roc_value: 0.9464030027517367
XGBoost threshold: 0.5603106617927551
ROC for the test dataset 94.6%
```




```
Time Taken by Model: --- 21.104939460754395 seconds ---
--------------------------------------------------------------------------------
ANN Model
Epoch 1/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.2444 - accuracy: 0.9168
Epoch 2/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.0891 - accuracy: 0.9701
Epoch 3/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.0657 - accuracy: 0.9789
Epoch 4/5
5325/5325 [==============================] - 17s 3ms/step - loss: 0.0516 - accuracy: 0.9838
Epoch 5/5
5325/5325 [==============================] - 14s 3ms/step - loss: 0.0393 - accuracy: 0.9877
2496/2496 [==============================] - 8s 3ms/step - loss: 0.0293 - accuracy: 0.9910
313/313 [==============================] - 1s 1ms/step
accuracy_score : 0.9941
Confusion Matrix
```




```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9983
           1       0.20      0.82      0.32        17

    accuracy                           0.99     10000
   macro avg       0.60      0.91      0.66     10000
weighted avg       1.00      0.99      1.00     10000

ANN roc_value: 0.944962318294041
ANN threshold: 0.06009089574217796
ROC for the test dataset 94.5%
```




```
Time Taken by Model: --- 87.03782868385315 seconds ---
--------------------------------------------------------------------------------
```

In [388]:
```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Out[388]:

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9990	0.950886	0.030690
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.933580	0.057833
2	RepeatedKFold Cross Validation	KNN	0.9990	0.866296	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.9989	0.833033	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.9992	0.899750	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.9997	0.965381	0.300000
6	RepeatedKFold Cross Validation	XGBoost	0.9996	0.936365	0.283122
7	RepeatedKFold Cross Validation	SVM	0.9981	0.724487	0.003482
8	RepeatedKFold Cross Validation	ANN	0.9984	0.927278	0.000521
9	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9992	0.956638	0.002113
10	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.892559	0.301463
11	StratifiedKFold Cross Validation	KNN	0.9990	0.823250	0.200000
12	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.9985	0.823079	1.000000
13	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.9990	0.823329	1.000000
14	StratifiedKFold Cross Validation	Random Forest	0.9995	0.937102	0.010000
15	StratifiedKFold Cross Validation	XGBoost	0.9993	0.971463	0.001395
16	StratifiedKFold Cross Validation	SVM	0.9975	0.634779	0.003424
17	StratifiedKFold Cross Validation	ANN	0.9985	0.925528	0.000077
18	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9808	0.963078	0.383737
19	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
20	Random Oversampling with StratifiedKFold CV	KNN	0.9986	0.823164	0.600000
21	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9987	0.793817	1.000000
22	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9991	0.823379	1.000000
23	Random Oversampling with StratifiedKFold CV	Random Forest	0.9994	0.937891	0.020000
24	Random Oversampling with StratifiedKFold CV	XGBoost	0.9992	0.971475	0.135445
25	Random Oversampling with StratifiedKFold CV	ANN	0.9991	0.946933	0.000007
26	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9700	0.963647	0.370725
27	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
28	SMOTE Oversampling with StratifiedKFold CV	KNN	0.9925	0.850154	1.000000
29	SMOTE Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9955	0.704129	1.000000
30	SMOTE Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9974	0.822528	1.000000
31	SMOTE Oversampling with StratifiedKFold CV	Random Forest	0.9995	0.958948	0.180000
32	SMOTE Oversampling with StratifiedKFold CV	XGBoost	0.9948	0.946403	0.560311
33	SMOTE Oversampling with StratifiedKFold CV	ANN	0.9941	0.944962	0.060091

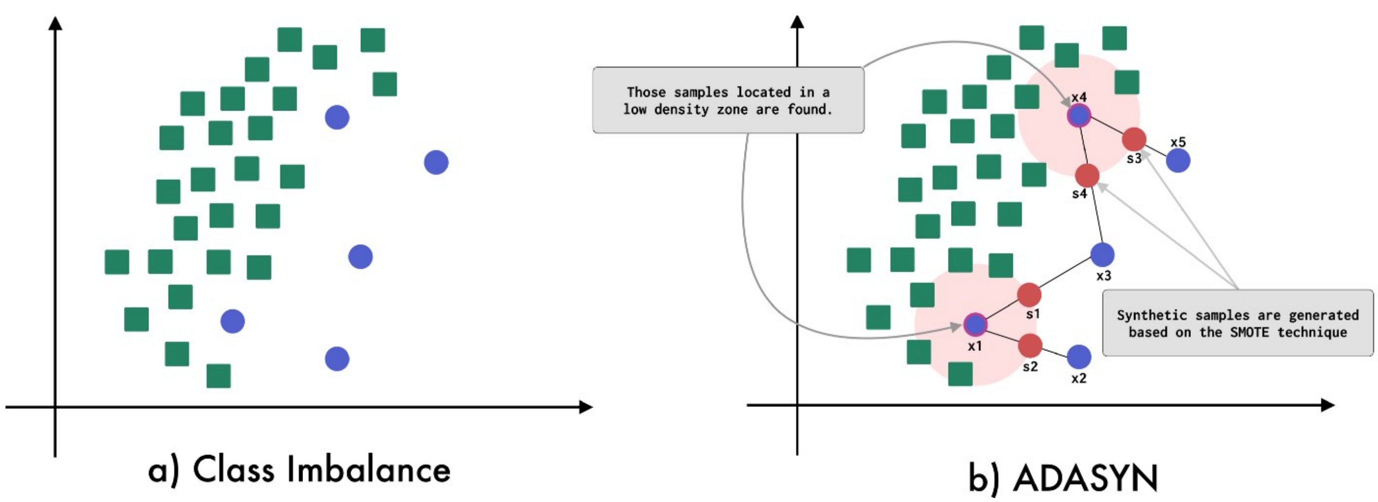
Results for SMOTE Oversampling with StratifiedKFold:

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

Oversampling with ADASYN Oversampling

ADASYN (Adaptive Synthetic Sampling):

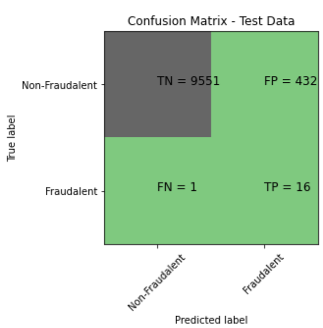
- ADASYN (Adaptive Synthetic Sampling) is a data augmentation technique for handling imbalanced datasets by creating synthetic samples for the minority class with adaptive weighting.
- Unlike SMOTE, which generates synthetic samples for all minority class instances with equal weight, ADASYN assigns higher weights to minority class instances that are harder to classify, leading to a more balanced distribution of synthetic samples. This can help improve the performance of classification models on imbalanced datasets.



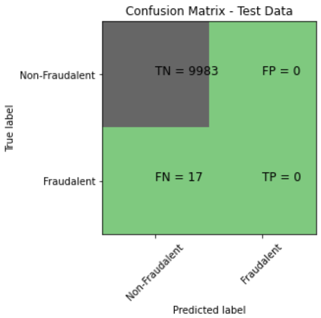
In [389]:
```
# Creating dataframe with ADASYN and StratifiedKFold
from sklearn.model_selection import StratifiedKFold
from imblearn import over_sampling

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

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

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

In [390]:
```
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_test)
print("Time Taken by Model: --- %s seconds ---" % (time.time() - start_time))
print('-'*80 )

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

#Run ANN Model
print("ANN Model")
start_time = time.time()
df_Results = buildAndgetmetricsANN(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 )
```

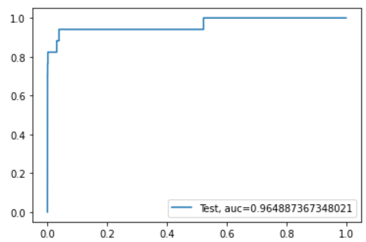
```
Logistic Regression with L1 And L2 Regularisation
Max auc_roc for l1: 0.5
Max auc_roc for l2: 0.990614868151021
Parameters for l1 regularisations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0.]]
[0.]
{1: array([[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
       [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
        0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]])}
Parameters for l2 regularisations
[[ 1.83122538  3.05070972  1.12889673  1.03609642  2.19537452 -1.24143156
  -1.47721851  0.54078427  0.0192194  -1.43836518  1.73165582 -1.26261903
  -0.34025159 -2.96330909  0.2452743   0.39676769 -2.59767792 -2.16994575
   2.23465888 -2.04439939 -2.15269522  1.43232107  2.91884907 -0.03885319
   1.76321095 -1.04053388  1.69370689  2.81180162  0.02545477 -0.01301738]]
[-7.4042831]
{1: array([[0.60230187, 0.61016433, 0.65488978, 0.76861658, 0.89753741,
        0.9626292 , 0.9786859 , 0.98531415, 0.98823005, 0.98940061,
        0.99001333, 0.99001333, 0.99001333, 0.99001333, 0.99001333,
        0.99001333, 0.99001333, 0.99001333, 0.99001333, 0.99001333],
       [0.59198551, 0.59948425, 0.64389145, 0.7605135 , 0.89724268,
        0.9650588 , 0.98016343, 0.9861063 , 0.98882313, 0.98965897,
        0.99003914, 0.99003914, 0.99003914, 0.99003914, 0.99003914,
        0.99003914, 0.99003914, 0.99003914, 0.99003914, 0.99003914],
       [0.59646958, 0.60372513, 0.64795962, 0.76485066, 0.89752787,
        0.96500261, 0.98093438, 0.98693348, 0.98917879, 0.99021551,
        0.99059208, 0.99059208, 0.99059208, 0.99059208, 0.99059208,
        0.99059208, 0.99059208, 0.99059208, 0.99059208, 0.99059208],
       [0.60275353, 0.61019224, 0.6568437 , 0.77671405, 0.905517  ,
        0.96783029, 0.98182116, 0.98775584, 0.99030012, 0.99095372,
        0.99131166, 0.99131166, 0.99131166, 0.99131166, 0.99131166,
        0.99131166, 0.99131166, 0.99131166, 0.99131166, 0.99131166],
       [0.59016023, 0.59817008, 0.64484347, 0.76614248, 0.90034203,
        0.96628354, 0.98081964, 0.98669327, 0.98954741, 0.9905303 ,
        0.99089345, 0.99089345, 0.99089345, 0.99089345, 0.99089345,
        0.99089345, 0.99089345, 0.99089345, 0.99089345, 0.99089345],
       [0.6029776 , 0.61078779, 0.65647905, 0.77359231, 0.90302125,
        0.96631566, 0.98100551, 0.98667924, 0.98924792, 0.99015907,
        0.99055022, 0.99055022, 0.99055022, 0.99055022, 0.99055022,
        0.99055022, 0.99055022, 0.99055022, 0.99055022, 0.99055022],
       [0.59739262, 0.60530873, 0.65086531, 0.77048969, 0.90196038,
        0.96541083, 0.979455  , 0.9853595 , 0.98834314, 0.98941996,
        0.98998167, 0.98998167, 0.99035117, 0.99035117, 0.99035117,
        0.99035117, 0.99035117, 0.99035117, 0.99035117, 0.99035117],
       [0.59637601, 0.60460371, 0.65204751, 0.77517531, 0.90412903,
        0.96663064, 0.98115004, 0.98686483, 0.98953402, 0.99038293,
        0.99069029, 0.99069029, 0.99069029, 0.99069029, 0.99069029,
        0.99069029, 0.99069029, 0.99069029, 0.99069029, 0.99069029],
       [0.59538801, 0.60313171, 0.64776172, 0.76594601, 0.89834426,
        0.96513194, 0.97999818, 0.98602632, 0.98872745, 0.9897023 ,
        0.99006168, 0.99025806, 0.99025806, 0.99025806, 0.99025806,
        0.99025806, 0.99025806, 0.99025806, 0.99025806, 0.99025806],
       [0.59666165, 0.60404397, 0.64863165, 0.76404433, 0.89643651,
        0.96453479, 0.98061611, 0.98729169, 0.99019847, 0.99112301,
        0.99144929, 0.99144929, 0.99144929, 0.99144929, 0.99144929,
        0.99144929, 0.99144929, 0.99144929, 0.99144929, 0.99144929]])}
Accuarcy of Logistic model with l2 regularisation : 0.9567
Confusion Matrix
```



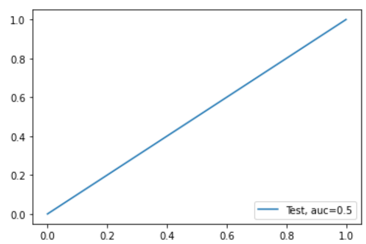

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.96      0.98      9983
           1       0.04      0.94      0.07        17

    accuracy                           0.96     10000
   macro avg       0.52      0.95      0.52     10000
weighted avg       1.00      0.96      0.98     10000

Accuarcy of Logistic model with l1 regularisation : 0.9983
Confusion Matrix
```



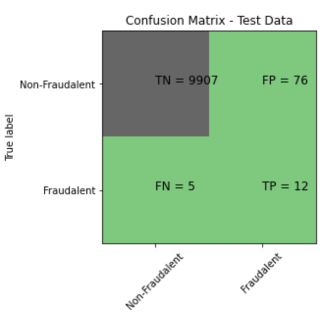

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.00      0.00      0.00        17

    accuracy                           1.00     10000
   macro avg       0.50      0.50      0.50     10000
weighted avg       1.00      1.00      1.00     10000

l2 roc_value: 0.964887367348021
l2 threshold: 0.5350685855528712
ROC for the test dataset 96.5%
```



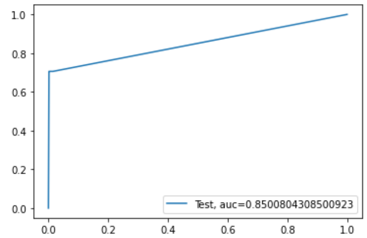

```
l1 roc_value: 0.5
l1 threshold: 1.5
ROC for the test dataset 50.0%
```



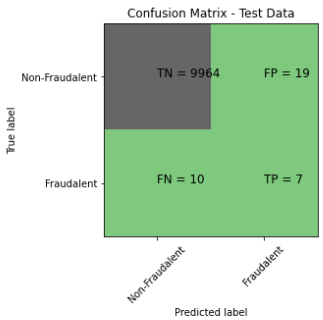

```
Time Taken by Model: --- 116.42573523521423 seconds ---
--------------------------------------------------------------------------------
KNN Model
model score
0.9919
Confusion Matrix
```



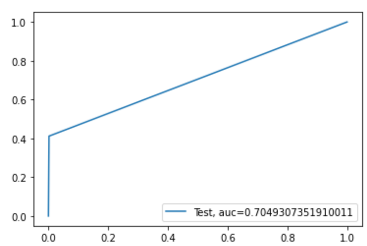

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9983
           1       0.14      0.71      0.23        17

    accuracy                           0.99     10000
   macro avg       0.57      0.85      0.61     10000
weighted avg       1.00      0.99      0.99     10000

KNN roc_value: 0.8500804308500923
KNN threshold: 1.0
ROC for the test dataset 85.0%
```



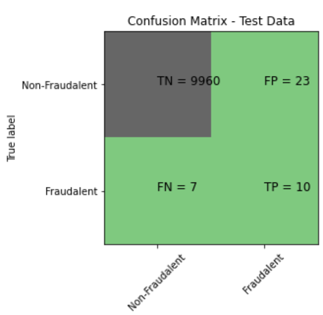

```
Time Taken by Model: --- 90.96645188331604 seconds ---
--------------------------------------------------------------------------------
Decision Tree Models with  'gini' & 'entropy' criteria
gini score: 0.9971
Confusion Matrix
```



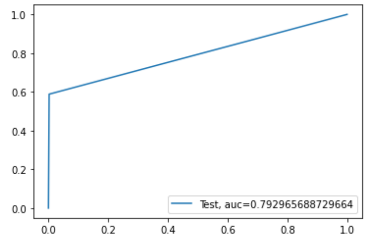

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.27      0.41      0.33        17

    accuracy                           1.00     10000
   macro avg       0.63      0.70      0.66     10000
weighted avg       1.00      1.00      1.00     10000

gini tree_roc_value: 0.7049307351910011
Tree threshold: 1.0
ROC for the test dataset 70.5%
```



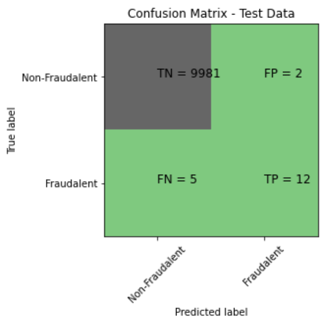

```
entropy score: 0.997
Confusion Matrix
```



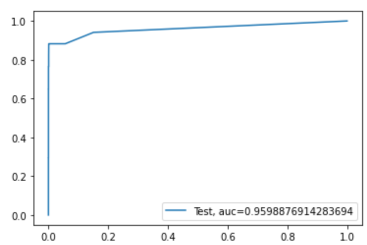

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.30      0.59      0.40        17

    accuracy                           1.00     10000
   macro avg       0.65      0.79      0.70     10000
weighted avg       1.00      1.00      1.00     10000

entropy tree_roc_value: 0.792965688729664
Tree threshold: 1.0
ROC for the test dataset 79.3%
```



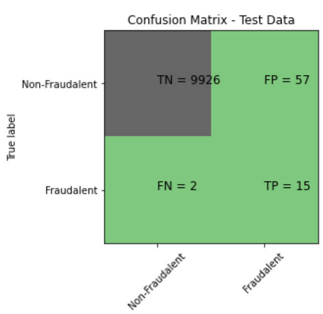

```
Time Taken by Model: --- 8.870583772659302 seconds ---
------------------------------------------------------------------------------
Random Forest Model
Model Accuracy: 0.9993
Confusion Matrix
```



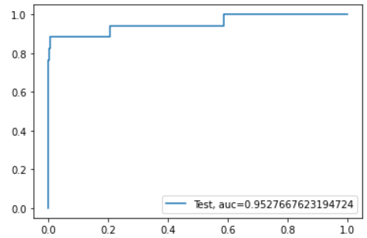

```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.86      0.71      0.77        17

    accuracy                           1.00     10000
   macro avg       0.93      0.85      0.89     10000
weighted avg       1.00      1.00      1.00     10000

Random Forest roc_value: 0.9598876914283694
Random Forest threshold: 0.15
ROC for the test dataset 96.0%
```



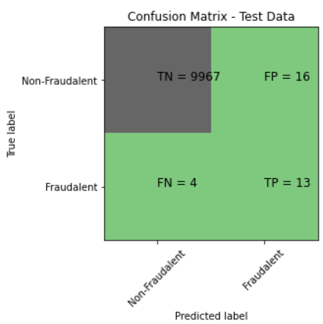

```
Time Taken by Model: --- 44.52493453025818 seconds ---
------------------------------------------------------------------------------
XGBoost Model
Model Accuracy: 0.9941
Confusion Matrix
```



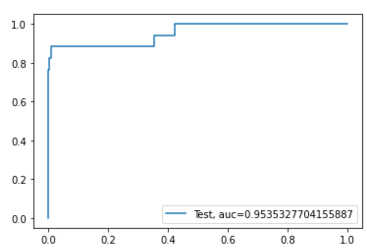

```
classification Report
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9983
           1       0.21      0.88      0.34        17

    accuracy                           0.99     10000
   macro avg       0.60      0.94      0.67     10000
weighted avg       1.00      0.99      1.00     10000

XGboost roc_value: 0.9527667623194724
XGBoost threshold: 0.575517475605011
ROC for the test dataset 95.3%
```




```
Time Taken by Model: --- 20.68782329559326 seconds ---
--------------------------------------------------------------------------------
ANN Model
Epoch 1/5
5325/5325 [==============================] - 16s 3ms/step - loss: 0.1769 - accuracy: 0.9382
Epoch 2/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.0594 - accuracy: 0.9793
Epoch 3/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.0404 - accuracy: 0.9861
Epoch 4/5
5325/5325 [==============================] - 16s 3ms/step - loss: 0.0291 - accuracy: 0.9909
Epoch 5/5
5325/5325 [==============================] - 15s 3ms/step - loss: 0.0249 - accuracy: 0.9928
2496/2496 [==============================] - 8s 3ms/step - loss: 0.0161 - accuracy: 0.9956
313/313 [==============================] - 1s 1ms/step
accuracy_score : 0.998
Confusion Matrix
```




```
classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9983
           1       0.45      0.76      0.57        17

    accuracy                           1.00     10000
   macro avg       0.72      0.88      0.78     10000
weighted avg       1.00      1.00      1.00     10000

ANN roc_value: 0.9535327704155887
ANN threshold: 0.0017699062591418624
ROC for the test dataset 95.4%
```




```
Time Taken by Model: --- 84.96875286102295 seconds ---
--------------------------------------------------------------------------------
```

In [391]:
```
# Checking the df_result dataframe which contains consolidated results of all the runs
df_Results
```

Out[391]:

	Methodology	Model	Accuracy	roc_value	threshold
0	RepeatedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9990	0.950886	0.030690
1	RepeatedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.933580	0.057833
2	RepeatedKFold Cross Validation	KNN	0.9990	0.866296	0.200000
3	RepeatedKFold Cross Validation	Tree Model with gini criteria	0.9989	0.833033	1.000000
4	RepeatedKFold Cross Validation	Tree Model with entropy criteria	0.9992	0.899750	1.000000
5	RepeatedKFold Cross Validation	Random Forest	0.9997	0.965381	0.300000
6	RepeatedKFold Cross Validation	XGBoost	0.9996	0.936365	0.283122
7	RepeatedKFold Cross Validation	SVM	0.9981	0.724487	0.003482
8	RepeatedKFold Cross Validation	ANN	0.9984	0.927278	0.000521
9	StratifiedKFold Cross Validation	Logistic Regression with L2 Regularisation	0.9992	0.956638	0.002113
10	StratifiedKFold Cross Validation	Logistic Regression with L1 Regularisation	0.9991	0.892559	0.301463
11	StratifiedKFold Cross Validation	KNN	0.9990	0.823250	0.200000
12	StratifiedKFold Cross Validation	Tree Model with gini criteria	0.9985	0.823079	1.000000
13	StratifiedKFold Cross Validation	Tree Model with entropy criteria	0.9990	0.823329	1.000000
14	StratifiedKFold Cross Validation	Random Forest	0.9995	0.937102	0.010000
15	StratifiedKFold Cross Validation	XGBoost	0.9993	0.971463	0.001395
16	StratifiedKFold Cross Validation	SVM	0.9975	0.634779	0.003424
17	StratifiedKFold Cross Validation	ANN	0.9985	0.925528	0.000077
18	Random Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9808	0.963078	0.383737
19	Random Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
20	Random Oversampling with StratifiedKFold CV	KNN	0.9986	0.823164	0.600000
21	Random Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9987	0.793817	1.000000
22	Random Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9991	0.823379	1.000000
23	Random Oversampling with StratifiedKFold CV	Random Forest	0.9994	0.937891	0.020000
24	Random Oversampling with StratifiedKFold CV	XGBoost	0.9992	0.971475	0.135445
25	Random Oversampling with StratifiedKFold CV	ANN	0.9991	0.946933	0.000007
26	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9700	0.963647	0.370725
27	SMOTE Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
28	SMOTE Oversampling with StratifiedKFold CV	KNN	0.9925	0.850154	1.000000
29	SMOTE Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9955	0.704129	1.000000
30	SMOTE Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9974	0.822528	1.000000
31	SMOTE Oversampling with StratifiedKFold CV	Random Forest	0.9995	0.958948	0.180000
32	SMOTE Oversampling with StratifiedKFold CV	XGBoost	0.9948	0.946403	0.560311
33	SMOTE Oversampling with StratifiedKFold CV	ANN	0.9941	0.944962	0.060091
34	ADASYN Oversampling with StratifiedKFold CV	Logistic Regression with L2 Regularisation	0.9567	0.964887	0.535069
35	ADASYN Oversampling with StratifiedKFold CV	Logistic Regression with L1 Regularisation	0.9983	0.500000	1.500000
36	ADASYN Oversampling with StratifiedKFold CV	KNN	0.9919	0.850080	1.000000
37	ADASYN Oversampling with StratifiedKFold CV	Tree Model with gini criteria	0.9971	0.704931	1.000000
38	ADASYN Oversampling with StratifiedKFold CV	Tree Model with entropy criteria	0.9970	0.792966	1.000000
39	ADASYN Oversampling with StratifiedKFold CV	Random Forest	0.9993	0.959888	0.150000
40	ADASYN Oversampling with StratifiedKFold CV	XGBoost	0.9941	0.952767	0.575517
41	ADASYN Oversampling with StratifiedKFold CV	ANN	0.9980	0.953533	0.001770

Results for ADASYN Oversampling with StratifiedKFold:

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

Overall conclusion after running the models on Oversampled data:

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

Hyperparameter Tuning

HPT - Xgboost Regression

- Hyperparameter tuning in XGBoost regression involves adjusting the values of certain parameters to optimize the performance of the model on a given dataset.
- Commonly tuned hyperparameters in XGBoost regression include the learning rate, the number of trees in the model, the maximum depth of each tree, the minimum child weight, the subsample ratio, and the regularization coefficients. The optimal values for these hyperparameters can be found using techniques such as grid search, random search, or bayesian optimization.

In [392]:
```python
# Performing Hyperparameter tuning
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
param_test = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2),
 'n_estimators':range(60,130,150),
 'learning_rate':[0.05,0.1,0.125,0.15,0.2],
 'gamma':[i/10.0 for i in range(0,5)],
 'subsample':[i/10.0 for i in range(7,10)],
 'colsample_bytree':[i/10.0 for i in range(7,10)]
}

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

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

Out[392]:
```
({'mean_fit_time': array([ 7.95628734, 17.85095525,  7.41580715, 13.04083924, 17.3790236 ]),
  'std_fit_time': array([0.84154201, 0.32982394, 0.63760971, 0.41301149, 1.98682468]),
  'mean_score_time': array([0.05614347, 0.06945863, 0.05744319, 0.06470861, 0.05429459]),
  'std_score_time': array([0.01722913, 0.01862621, 0.02510229, 0.01841114, 0.00632305]),
  'param_subsample': masked_array(data=[0.9, 0.9, 0.7, 0.7, 0.8],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_n_estimators': masked_array(data=[60, 60, 60, 60, 60],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_min_child_weight': masked_array(data=[1, 3, 1, 1, 3],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_max_depth': masked_array(data=[3, 9, 3, 5, 9],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_learning_rate': masked_array(data=[0.15, 0.1, 0.15, 0.1, 0.1],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_gamma': masked_array(data=[0.4, 0.4, 0.3, 0.4, 0.3],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_colsample_bytree': masked_array(data=[0.7, 0.8, 0.7, 0.9, 0.8],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'subsample': 0.9,
    'n_estimators': 60,
    'min_child_weight': 1,
    'max_depth': 3,
    'learning_rate': 0.15,
    'gamma': 0.4,
    'colsample_bytree': 0.7},
   {'subsample': 0.9,
    'n_estimators': 60,
    'min_child_weight': 3,
    'max_depth': 9,
    'learning_rate': 0.1,
    'gamma': 0.4,
    'colsample_bytree': 0.8},
   {'subsample': 0.7,
    'n_estimators': 60,
    'min_child_weight': 1,
    'max_depth': 3,
    'learning_rate': 0.15,
    'gamma': 0.3,
    'colsample_bytree': 0.7},
   {'subsample': 0.7,
    'n_estimators': 60,
    'min_child_weight': 1,
    'max_depth': 5,
    'learning_rate': 0.1,
    'gamma': 0.4,
    'colsample_bytree': 0.9},
   {'subsample': 0.8,
    'n_estimators': 60,
    'min_child_weight': 3,
    'max_depth': 9,
    'learning_rate': 0.1,
    'gamma': 0.3,
    'colsample_bytree': 0.8}],
  'split0_test_score': array([0.99983833, 0.99996529, 0.99982739, 0.99993807, 0.99997238]),
  'split1_test_score': array([0.99991597, 1.        , 0.99992885, 0.99997357, 1.        ]),
  'split2_test_score': array([0.99972677, 0.99999417, 0.99979866, 0.99990744, 0.99997796]),
  'split3_test_score': array([0.99989202, 0.99992623, 0.99987478, 0.99988858, 0.99992124]),
  'split4_test_score': array([0.99993421, 1.        , 0.99992461, 0.99998018, 0.99998206]),
  'mean_test_score': array([0.99986146, 0.99997714, 0.99987086, 0.99993757, 0.99997073]),
  'std_test_score': array([7.46656402e-05, 2.85217757e-05, 5.17106231e-05, 3.58332424e-05,
         2.64156355e-05]),
  'rank_test_score': array([5, 1, 4, 3, 2], dtype=int32)},
 {'subsample': 0.9,
  'n_estimators': 60,
  'min_child_weight': 3,
  'max_depth': 9,
  'learning_rate': 0.1,
  'gamma': 0.4,
  'colsample_bytree': 0.8},
 0.9999771372442581)
```

- Please note that the hyperparameters found above using RandomizedSearchCV and the hyperparameters used below in creating the final model might be different, the reason being, I have executed the RandomizedSearchCV multiple times to find which set of hyperparameters gives the optimum result and finally used the one below which gave me the best performance.

In [393]:
```python
# Creating XGBoost model with selected hyperparameters
from xgboost import XGBClassifier

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

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

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

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

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

```
Model Accuracy: 0.9991
XGboost roc_value: 0.9680280005420979
XGBoost threshold: 0.0020904664415866137
```

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

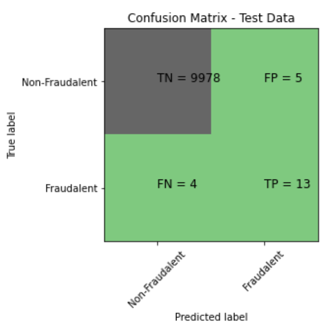
In [394]:
```python
imp_var = []
for i in clf.feature_importances_:
    imp_var.append(i)
print('Top var =', imp_var.index(np.sort(clf.feature_importances_)[-1])+1)
print('2nd Top var =', imp_var.index(np.sort(clf.feature_importances_)[-2])+1)
print('3rd Top var =', imp_var.index(np.sort(clf.feature_importances_)[-3])+1)
```

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

In [395]:
```python
# 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))
```

```
XGboost roc_value: 0.9680280005420979
XGBoost threshold: 0.0020904664415866137
```

In [396]:
```python
# Confusion matrix
Plot_confusion_matrix(y_test,clf.predict(X_test))
```



Conclusion

- In the oversample cases, of all the models we build found that the XGBOOST model with Random Oversampling with StratifiedKFold CV gave us the best accuracy and ROC on oversampled data. Post that we performed hyperparameter tuning and got the below metrices :
  - XGboost roc_value: 0.9680280005420979
  - XGBoost threshold: 0.0020904664415866137
- However, of all the models we created we found Logistic Regression with L2 Regularisation for StratifiedKFold cross validation (without any oversampling or undersampling) gave us the best result.
- Also, all the models we created gave good results