Before Starting:

If you liked this kernel please don't forget to upvote the project, this will keep me motivated to other kernels in the future. I hope you enjoy our deep exploration into this dataset. Let's begin!

Credit Card Fraud Detection

Anonymized credit card transactions labeled as fraudulent or genuine

Table of Contents

__Introduction__

It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase. Eventually, it is also important for companies NOT to detect transactions which are genuine as fraudulent, otherwise, companies would keep blocking the credit card, and which may lead to customer dissatisfaction. So here are two important expects of this analysis:

- What would happen when the company will not able to detect the fraudulent transaction and would not confirm from a customer about this recent transaction whether it was made by him/her.
- In contract, what would happen when the company will detect a genuine transaction as fraudulent and keep calling customer for confirmation or might block the card.

The datasets contain transactions that have 492 frauds out of 284,807 transactions. So the dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. When we try to build the prediction model with this kind of unbalanced dataset, then the model will be more inclined towards to detect new unseen transaction as genuine as our dataset contains about 99% genuine data.

As our dataset is highly imbalanced, so we shouldn't use accuracy score as a metric because it will be usually high and misleading, instead use we should focus on f1-score, precision/recall score or confusion matrix.

Load Data

In [1]:
```
# Import Libraries
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# import cufflinks as cf
import plotly
import datetime
import math
import matplotlib
import sklearn
from IPython.display import HTML
from IPython.display import YouTubeVideo

import pickle
import os

import plotly.express as px
import plotly.graph_objects as go
import plotly.figure_factory as ff
from plotly.subplots import make_subplots

# Print versions of libraries
print(f"Numpy version : Numpy {np.__version__}")
print(f"Pandas version : Pandas {pd.__version__}")
print(f"Matplotlib version : Matplotlib {matplotlib.__version__}")
print(f"Seaborn version : Seaborn {sns.__version__}")
print(f"SkLearn version : SkLearn {sklearn.__version__}")
# print(f"Cufflinks version : cufflinks {cf.__version__}")
print(f"Plotly version : plotly {plotly.__version__}")

# Magic Functions for In-Notebook Display
# %matplotlib inline

%matplotlib inline

# Setting seabon style
sns.set(style='darkgrid', palette='colorblind')
```

```
Numpy version : Numpy 1.18.1
Pandas version : Pandas 1.0.3
Matplotlib version : Matplotlib 3.2.1
Seaborn version : Seaborn 0.10.0
SkLearn version : SkLearn 0.23.1
Plotly version : plotly 4.8.1
```

Import the Dataset

In [2]:
```
df = pd.read_csv('../input/creditcardfraud/creditcard.csv', encoding='
latin_1')
```

In [3]:
```
# Converting all column names to lower case
df.columns = df.columns.str.lower()
```

In [4]:
```
df.head()
```

Out[4]:

	time	v1	v2	v3	v4	v5	v6	v7	v8
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533

5 rows × 31 columns

In [5]:
```
df.tail()
```

Out[5]:

	time	v1	v2	v3	v4	v5	v6	v7
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006

5 rows × 31 columns

- **Due to confidentiality issue, original features V1, V2,... V28 have been transformed with PCA, however, we may guess that these features might be originally credit card number, expiry date, CVV, cardholder name, transaction location, transaction date-time, etc.**
- The only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning.
- Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

In [6]:
```
# Customising default values to view all columns
pd.options.display.max_rows = 100
pd.options.display.max_columns = 100

# pd.set_option('display.max_rows',1000)
```

In [7]:
```
df.head(10)
```

Out[7]:

	time	v1	v2	v3	v4	v5	v6	v7	v8
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	0.260314
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	0.081213
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	-3.807864
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	0.069539

__Exploratory Data Analysis__

Once the data is read into python, we need to explore/clean/filter it before processing it for machine learning It involves adding/deleting few columns or rows, joining some other data, and handling qualitative variables like dates.

Now that we have the data, I wanted to run a few initial comparisons between the three columns - Time, Amount, and Class.

Checking concise summary of dataset

It is also a good practice to know the features and their corresponding data types, along with finding whether they contain null values or not.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   time    284807 non-null  float64
 1   v1      284807 non-null  float64
 2   v2      284807 non-null  float64
 3   v3      284807 non-null  float64
 4   v4      284807 non-null  float64
 5   v5      284807 non-null  float64
 6   v6      284807 non-null  float64
 7   v7      284807 non-null  float64
 8   v8      284807 non-null  float64
 9   v9      284807 non-null  float64
 10  v10     284807 non-null  float64
 11  v11     284807 non-null  float64
 12  v12     284807 non-null  float64
 13  v13     284807 non-null  float64
 14  v14     284807 non-null  float64
 15  v15     284807 non-null  float64
 16  v16     284807 non-null  float64
 17  v17     284807 non-null  float64
 18  v18     284807 non-null  float64
 19  v19     284807 non-null  float64
 20  v20     284807 non-null  float64
 21  v21     284807 non-null  float64
 22  v22     284807 non-null  float64
 23  v23     284807 non-null  float64
 24  v24     284807 non-null  float64
 25  v25     284807 non-null  float64
 26  v26     284807 non-null  float64
 27  v27     284807 non-null  float64
 28  v28     284807 non-null  float64
 29  amount  284807 non-null  float64
 30  class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

Highlights

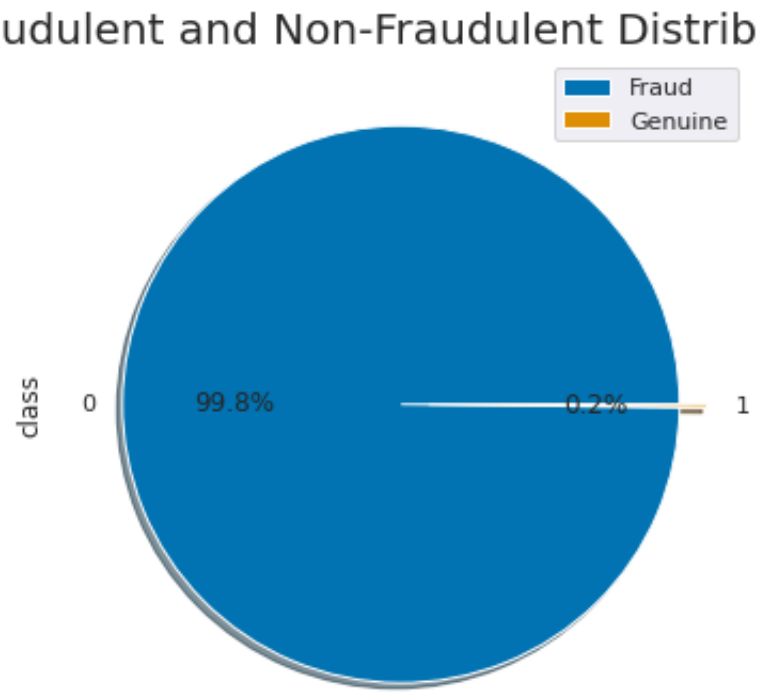
- Dataset contains details of 284807 transactions with 31 features.
- There is no missing data in our dataset, every column contain exactly 284807 rows.
- All data types are float64, except 1: Class
- All data types are float64, except 1: Class
- 28 columns have Sequential Names and values that don't make any logical sense - > V1, V2 ....V28
- 3 columns: TIME, AMOUNT and CLASS which can be analysed for various INSIGHTS!
- Memory Usage: 67.4 MB, not so Harsh !!

Count unique values of label

In [9]:
```
print(df['class'].value_counts())
print('\n')
print(df['class'].value_counts(normalize=True))
```

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


0    0.998273
1    0.001727
Name: class, dtype: float64
```

In [10]:
```
df["class"].value_counts().plot(kind = 'pie',explode=[0, 0.1],figsize=
(6, 6),autopct='%1.1f%%',shadow=True)
plt.title("Fraudulent and Non-Fraudulent Distribution",fontsize=20)
plt.legend(["Fraud", "Genuine"])
plt.show()
```



Highlights

This dataset has 492 frauds out of 284,315 transactions. The dataset is **highly unbalanced**, the positive class (frauds) account for 0.172% of all transactions. Most of the transactions are non-fraud. If we use this dataframe as the base for our predictive models and analysis, our algorithms will probably overfit since it will "assume" that most transactions are not a fraud. But we don't want our model to assume, we want our model to detect patterns that give signs of fraud!

Generate descriptive statistics

The describe() function generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Let's summarize the central tendency, dispersion and shape of a dataset's distribution. Out of all the columns, the only ones that made the most sense were Time, Amount, and Class (fraud or not fraud). The other 28 columns were transformed using what seems to be a PCA dimensionality reduction in order to protect user identities.

The data itself is short in terms of time (it's only 2 days long), and these transactions were made by European cardholders.

In [11]:
```
df[['time','amount']].describe()
```

Out[11]:

	time	amount
count	284807.000000	284807.000000
mean	94813.859575	88.349619
std	47488.145955	250.120109
min	0.000000	0.000000
25%	54201.500000	5.600000
50%	84692.000000	22.000000
75%	139320.500000	77.165000
max	172792.000000	25691.160000

Highlights

- On an average, credit card transaction is happening at every 94813.86 seconds.
- Average transaction amount is 88.35 with a standard deviation of 250, with a minimum amount of 0.0 and the maximum amount 25,691.16. By seeing the 75% and the maximum amount, it looks like the feature 'Amount' is highly **positively skewed**. We will check the distribution graph of the amount to get more clarity.

Finding null values

In [12]:
```
# Dealing with missing data
df.isnull().sum().max()
```

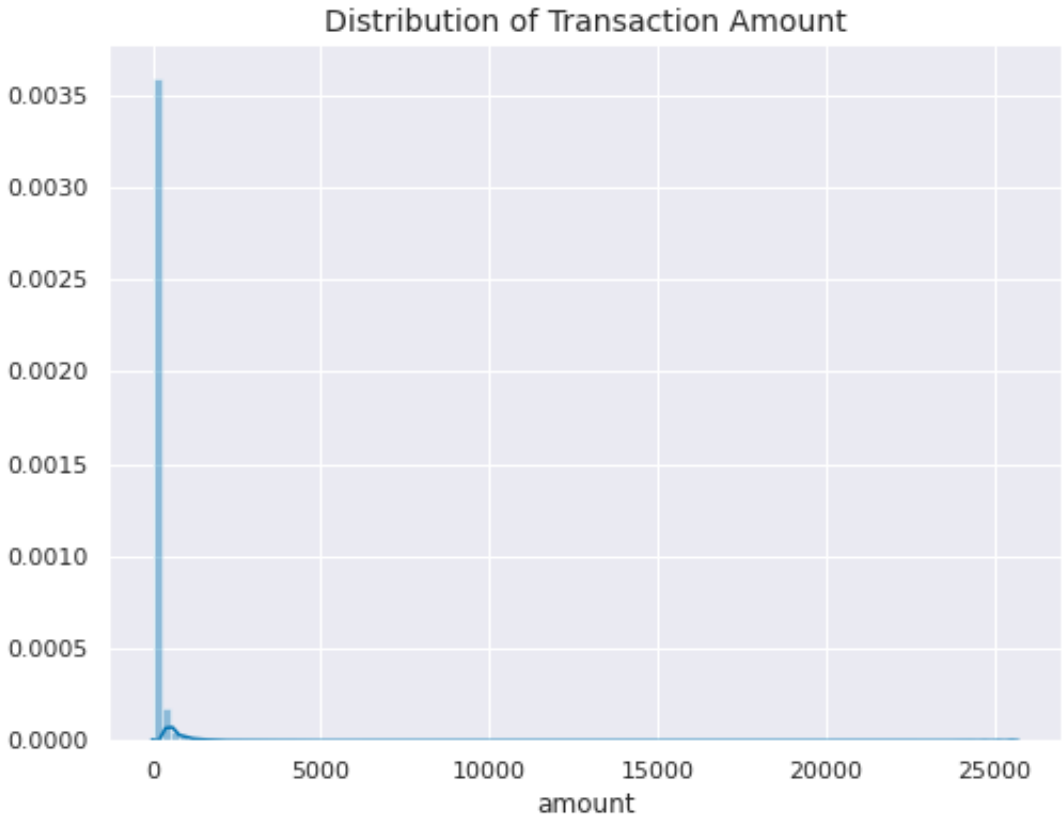
Out[12]: 0

Highlights

There are no missing values present in the dataset. It is not necessary that missing values are present in the dataset in the form of NA, NAN, Zeroes etc, it may be present by some other values also that can be explored by analysing each feature.

Distribution of Amount

In [13]:
```
plt.figure(figsize=(8,6))
plt.title('Distribution of Transaction Amount', fontsize=14)
sns.distplot(df['amount'], bins=100)
plt.show()
```



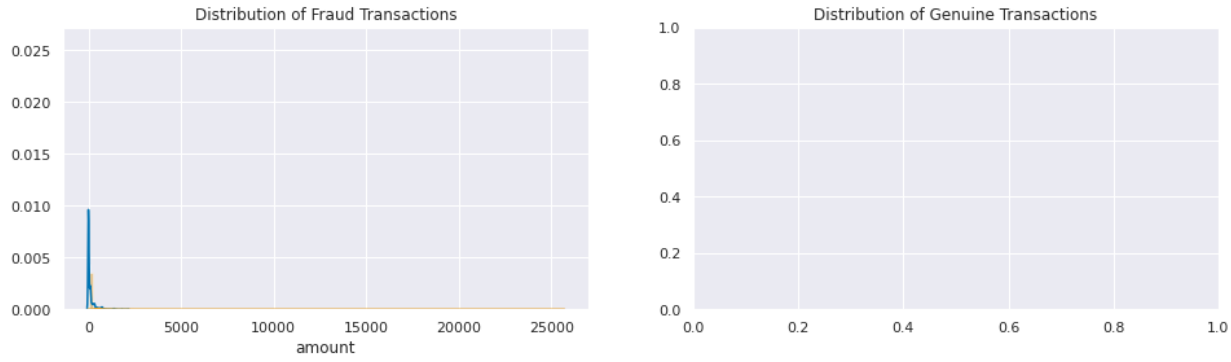
Highlights

Most the transaction amount falls between 0 and about 3000 and we have some outliers for really big amount transactions and it may actually make sense to drop those outliers in our analysis if they are just a few points that are very extreme.

Most daily transactions are not extremely expensive, but it's likely where most fraudulent transactions are occurring as well.

Distribution of Amount for Fradulent & Genuine transactions

In [14]:
```
fig, axs = plt.subplots(ncols=2,figsize=(16,4))
sns.distplot(df[df['class'] == 1]['amount'], bins=100, ax=axs[0])
axs[0].set_title("Distribution of Fraud Transactions")

sns.distplot(df[df['class'] == 0]['amount'], bins=100, ax=axs[0])
axs[1].set_title("Distribution of Genuine Transactions")

plt.show()
```



Highlights

This graph shows that most of the fraud transaction amount is less than 500 dollars. This also shows that the fraud transaction is very high for an amount near to 0, let's find that amount.

In [15]:
```
print("Fraud Transaction distribution : \n",df[(df['class'] == 1)]['amount'].value_counts().head())
print("\n")
print("Maximum amount of fraud transaction - ",df[(df['class'] == 1)]['amount'].max())
print("Minimum amount of fraud transaction - ",df[(df['class'] == 1)]['amount'].min())
```

```
Fraud Transaction distribution :
 1.00     113
0.00      27
99.99     27
0.76      17
0.77      10
Name: amount, dtype: int64


Maximum amount of fraud transaction -  2125.87
Minimum amount of fraud transaction -  0.0
```

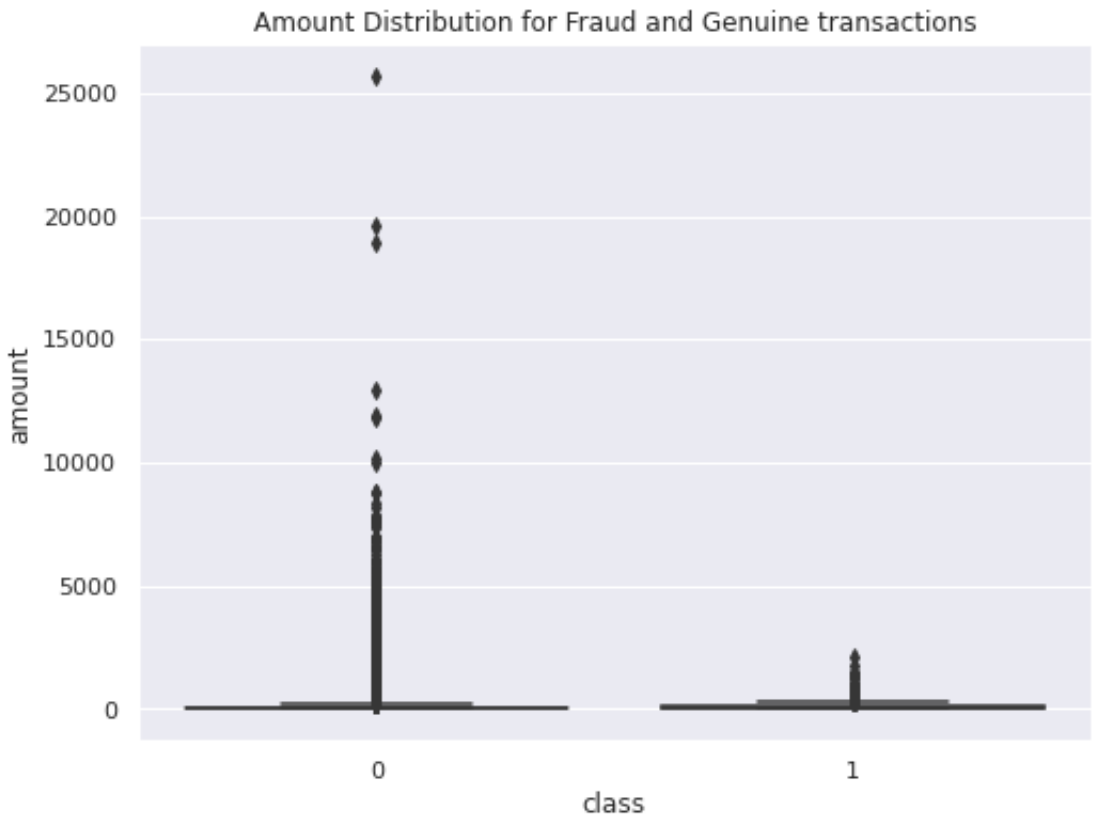
Highlights

- There are 113 fraud transactions for just one dollor and 27 fraud transaction for $99.99. And higest fraud transaction amount was 2125.87 and lowest was just 0.00.
- There are 27 fraud transaction for zero amount. Zero Authorization is an account verification method for credit cards that is used to verify a cardholders information without charging the consumer. Instead, an amount of zero is charged on the card to store the credit card information in the form of a token and to determine whether the card is legitimate or not. After creating the token, is then possible to charge the consumer with a new transaction with either Tokenization or Recurring Payments

Ref : https://docs.multisafepay.com/tools/zero-authorization/what-is-zero-authorization/ (https://docs.multisafepay.com/tools/zero-authorization/what-is-zero-authorization/)

Distribution of Amount w.r.t Class

In [16]:
```
plt.figure(figsize=(8,6))
sns.boxplot(x='class', y='amount',data = df)
plt.title('Amount Distribution for Fraud and Genuine transactions')
plt.show()
```



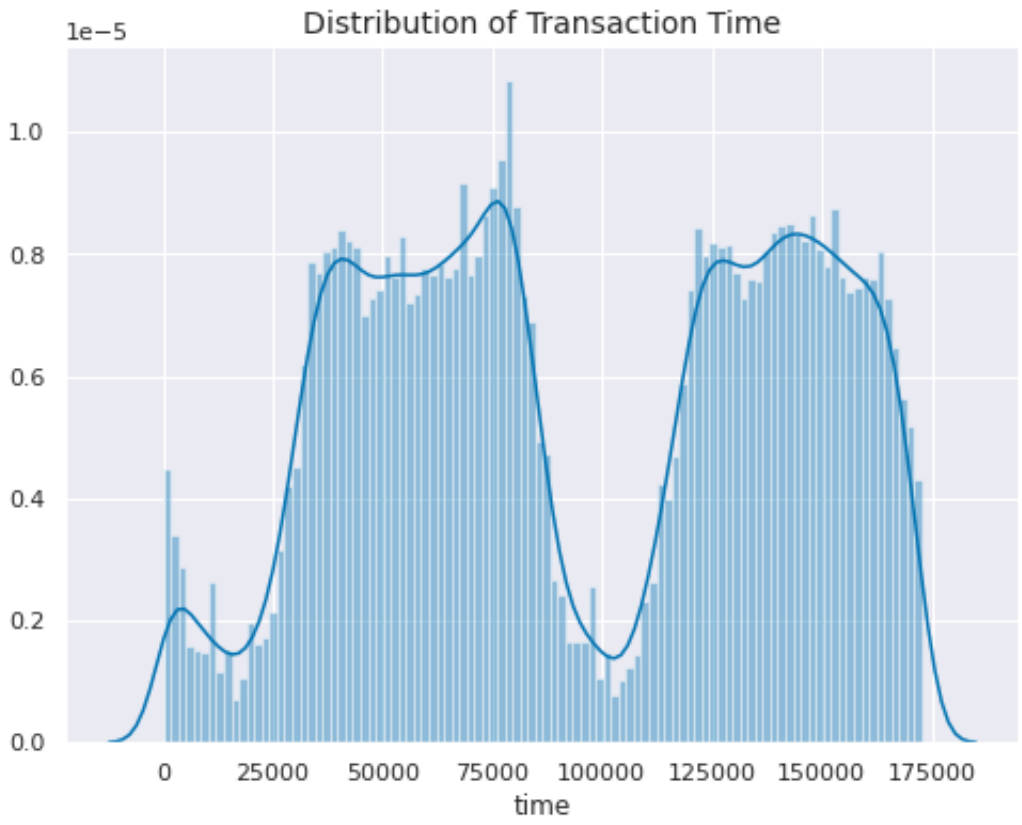
Highlights

Most the transaction amount falls between 0 and about 3000 and we have some outliers for really big amount transactions and it may actually make sense to drop those outliers in our analysis if they are just a few points that are very extreme. Also, we should be conscious about that these **outliers should not be the fraudulent transaction**. Generally, fraudulent transactions of the very big amount and removing them from the data can make the predicting model bais.

So we can essentially build a model that realistically predicts transaction as fraud without affected by outliers. It may not be really useful to actually have our model train on these extreme outliers.

Distribution of Time

In [17]:
```
plt.figure(figsize=(8,6))
plt.title('Distribution of Transaction Time', fontsize=14)
sns.distplot(df['time'], bins=100)
plt.show()
```

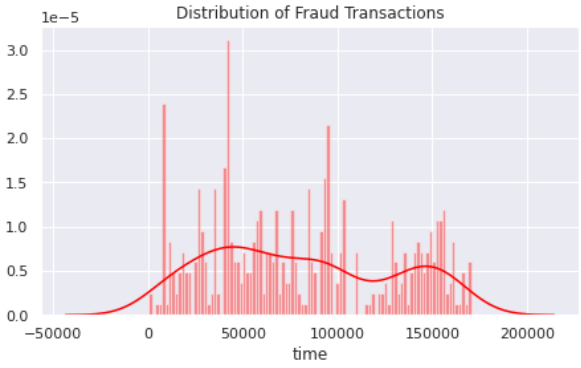


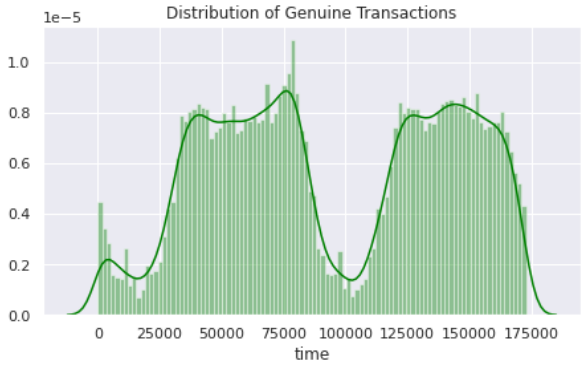
Highlights

By seeing the graph, we can see there are two peaks in the graph and even there are some local peaks. We can think of these as the time of the day like the peak is the day time when most people do the transactions and the depth is the night time when most people just sleeps. We already know that data contains a credit card transaction for only two days, so there are two peaks for day time and one depth for one night time.

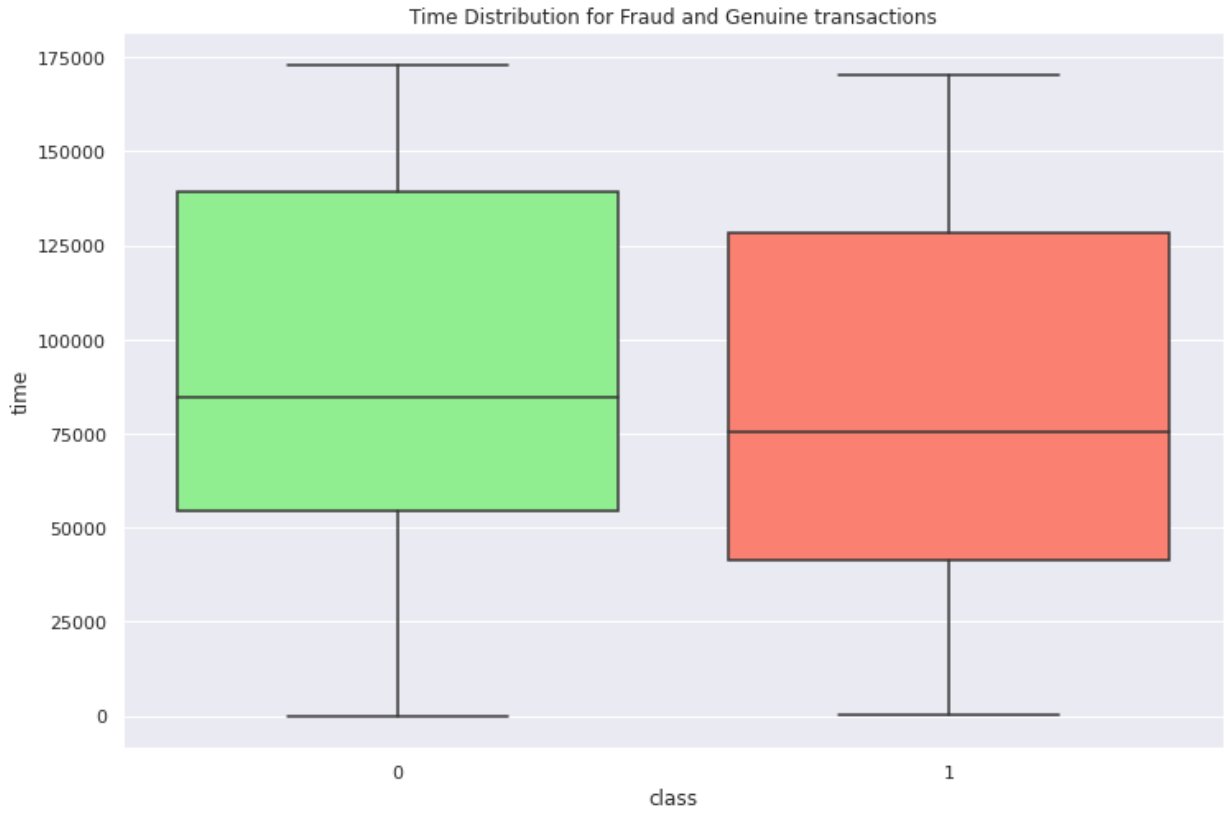
Distribution of time w.r.t. transactions types

In [18]:
```
fig, axs = plt.subplots(ncols=2, figsize=(16,4))

sns.distplot(df[(df['class'] == 1)]['time'], bins=100, color='red', ax
=axs[0])
axs[0].set_title("Distribution of Fraud Transactions")

sns.distplot(df[(df['class'] == 0)]['time'], bins=100, color='green',
ax=axs[1])
axs[1].set_title("Distribution of Genuine Transactions")

plt.show()
```



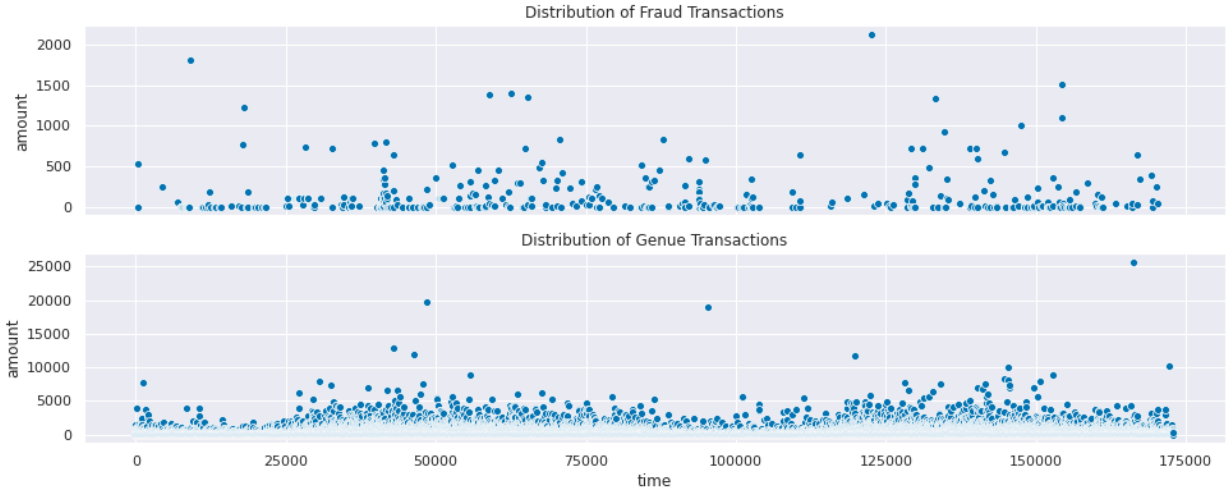


In [19]:
```
plt.figure(figsize=(12,8))
ax = sns.boxplot(x='class', y='time',data = df)

# Change the appearance of that box
ax.artists[0].set_facecolor('#90EE90')
ax.artists[1].set_facecolor('#FA8072')

plt.title('Time Distribution for Fraud and Genuine transactions')
plt.show()
```



Distribution of transaction type w.r.t amount

In [20]:
```
fig, axs = plt.subplots(nrows=2,sharex=True,figsize=(16,6))

sns.scatterplot(x='time',y='amount', data=df[df['class']==1], ax=axs[0
])
axs[0].set_title("Distribution of Fraud Transactions")

sns.scatterplot(x='time',y='amount', data=df[df['class']==0], ax=axs[1
])
axs[1].set_title("Distribution of Genue Transactions")

plt.show()
```



Removal of Outliers

The main objective is to remove "extreme outliers" from Amount Feature, rather than just outliers as we have seen that the distribution of amount is highly skewed after about $3000 and another reason to remove the extreme outliers is that dataset is highly imbalanced and we do not want to take risk of removing the class which is already less in number.

We have to be careful as to how far do we want the threshold for removing outliers. We determine the threshold by multiplying a number (ex: 1.5) by the (Interquartile Range). The higher this threshold is, the fewer outliers will detect (multiplying by a higher number ex: 3), and the lower this threshold is the more outliers it will detect.

In [21]:
```
Q3 = np.percentile(df['amount'], 75)
Q1 = np.percentile(df['amount'], 25)

cutoff = 5.0

# calculate interquartrile range - IQR = thirdQuartile - firstQuartile
IQR = (Q3 - Q1)

# Usually we take scale value of 1.5 times IQR to calculate. But this scale depends on the distribution
# followed by the data.

# Say if my data seem to follow exponential distribution then this scale would change.
# So I am taking sacle from 1.5 to 5.

# Lower outlier boundry (LOB) / Lower Whisker
LOB = Q1 - (IQR * cutoff)
print(f"Lower Whisker : {LOB}")

# Upper outlier boundry (UOB) / Upper Whisker
UOB = Q3 + (IQR * cutoff)
print(f"Upper Whisker : {UOB}")

amtAllOutliers = df[(df['amount'] < LOB) | (df['amount'] > UOB)]['amount']
amtFrdOutliers = df[(df['class'] == 1) & ((df['amount'] < LOB) | (df['amount'] > UOB))]['amount']
amtGenuOutliers = df[(df['class'] == 0) & ((df['amount'] < LOB) | (df['amount'] > UOB))]['amount']

print('\n')
print("No of amount outliers : ", amtAllOutliers.count())
print("No of fraud amount outliers : ", amtFrdOutliers.count())
print("No of genuine amount outliers : ", amtGenuOutliers.count())
# print("Percentage of outliers : ", round((amtGenuOutliers.count()/len(df))*100,2))
print("Percentage of Fraud amount outliers : ", round((amtFrdOutliers.count()/amtAllOutliers.count())*100,2))
```

```
Lower Whisker : -352.22499999999997
Upper Whisker : 434.99


No of amount outliers :  11366
No of fraud amount outliers :  41
No of genuine amount outliers :  11325
Percentage of Fraud amount outliers :  0.36
```

Highlights

There are a total number of 11366 outliers in amount columns, out of which 0.36% (41 in count) are fraud transactions.

Checking proportion of data for fraud vs genuine

In [22]:
```
# Check the balace of data including outliers
print("Balace of data including outliers")
print(df['class'].value_counts(normalize=True))
print("\nRatio of fraud vs genuine : ",df['class'].value_counts()[1]/df['class'].value_counts()[0])

print('\n')
# Check the balance of data excluding outliers
print("Balace of data excluding outliers")
print(df[(df['amount'] < LOB) | (df['amount'] > UOB)]['class'].value_counts(normalize=True))
print("\nRatio of fraud vs genuine excluding outliers: ",df[(df['amount'] < LOB) | (df['amount'] > UOB)]['class'].value_counts()[1]/df[(df['amount'] < LOB) | (df['amount'] > UOB)]['class'].value_counts()[0])
```

```
Balace of data including outliers
0    0.998273
1    0.001727
Name: class, dtype: float64

Ratio of fraud vs genuine :  0.0017304750013189597


Balace of data excluding outliers
0    0.996393
1    0.003607
Name: class, dtype: float64

Ratio of fraud vs genuine excluding outliers:  0.003620309050772627
```

Highlights

Now we have checked the total number of outliers in Amount feature and how many of them are fraudulent. We found that balance of fraud vs genuine is not impacted much, even the ratio of fraud vs genuine has increased from 0.0017 to 0.0036. So we can remove the outlier.

Delete Outliers

In [23]:
```
# check shape before deleting outliers
df.shape
```

Out[23]: (284807, 31)

In [24]:
```
# Removing outliers
df = df.drop(amtAllOutliers.index)
```

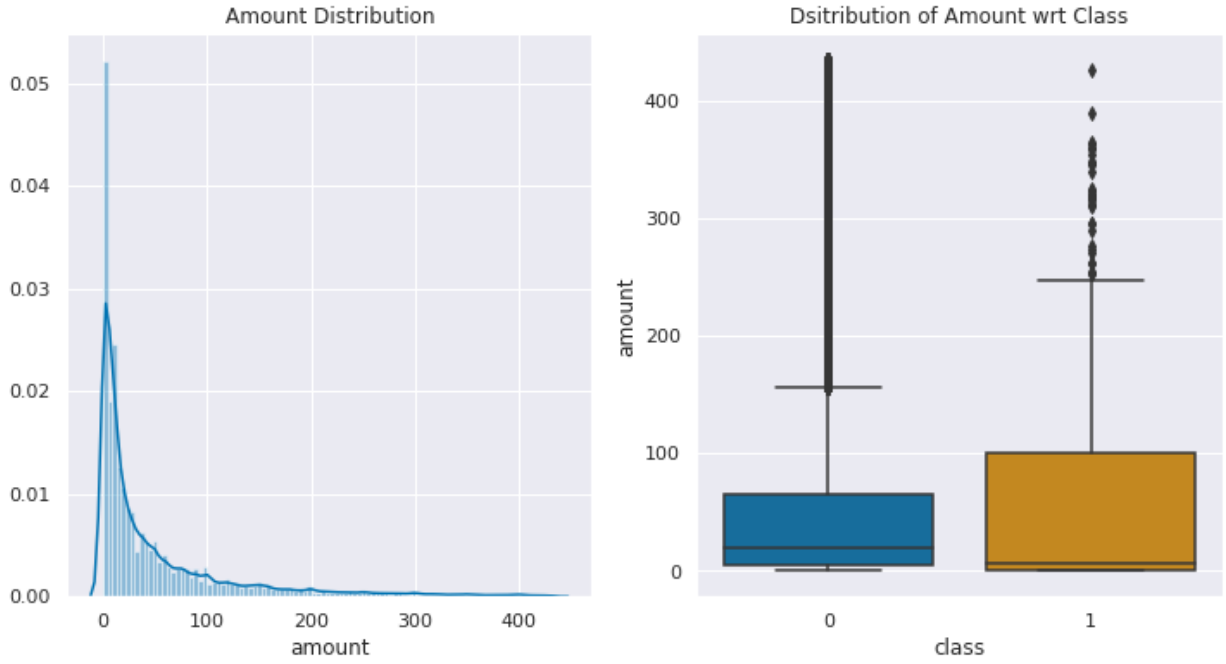
In [25]:
```
# check shape after deleting outliers
df.shape
```

Out[25]: (273441, 31)

In [26]:
```
df.reset_index(inplace = True , drop = True)
```

Check Amount Distribution after deleting outliers

In [27]:
```
fig, axs = plt.subplots(ncols=2,figsize=(12,6))

sns.distplot(df['amount'], bins=100, ax=axs[0])
axs[0].set_title("Amount Distribution")

sns.boxplot(x='class', y='amount',data = df, ax=axs[1])
axs[1].set_title("Dsitribution of Amount wrt Class")
plt.show()
```



Categorical vs Continuous Features

Finding unique values for each column to understand which column is categorical and which one is Continuous

In [28]:
```
# Finging unique values for each column
df[['time','amount','class']].nunique()
```

Out[28]:
```
time      122816
amount     24761
class          2
dtype: int64
```

In [29]:
```
fig = px.scatter(df, x="time", y="amount", color="class",
                 marginal_y="violin",marginal_x="box", trendline="ols"
, template="simple_white")
fig.show()
```

Correlation Among Explanatory Variables

Having **too many features** in a model is not always a good thing because it might cause overfitting and worse results when we want to predict values for a new dataset. Thus, **if a feature does not improve your model a lot, not adding it may be a better choice.**

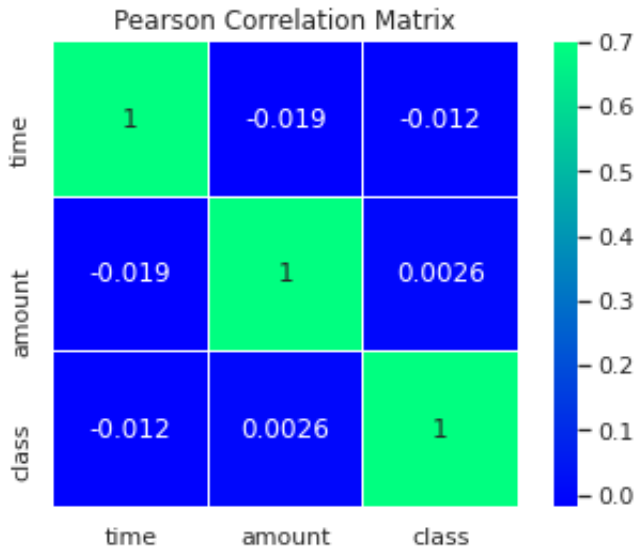
Another important thing is **correlation. If there is a very high correlation between two features, keeping both of them is not a good idea most of the time not to cause overfitting.** However, this does not mean that you must remove one of the highly correlated features.

Let's find out top 10 features which are highly correlated with a price.

In [30]:
```
df[['time','amount','class']].corr()['class'].sort_values(ascending=False).head(10)
```

Out[30]:
```
class     1.000000
amount    0.002600
time     -0.012475
Name: class, dtype: float64
```

In [31]:
```
plt.title('Pearson Correlation Matrix')
sns.heatmap(df[['time', 'amount','class']].corr(),linewidths=0.25,vmax=0.7,square=True,cmap="winter",
            linecolor='w',annot=True);
```



Highlights

It looks like that no features are highly correlated with any other features.

Lets check the data again after cleaning

In [32]:
```
df.shape
```

Out[32]:
```
(273441, 31)
```

In [33]:
```
df['class'].value_counts(normalize=True)
```

Out[33]:
```
0    0.998351
1    0.001649
Name: class, dtype: float64
```

Feature Engineering

Feature engineering on Time

Converting time from second to hour

In [34]:
```
# Converting time from second to hour
df['time'] = df['time'].apply(lambda sec : (sec/3600))
```

Calculating hour of the day

In [35]:
```
# Calculating hour of the day
df['hour'] = df['time']%24   # 2 days of data
df['hour'] = df['hour'].apply(lambda x : math.floor(x))
```

Calculating First and Second Day

In [36]:
```
# Calculating First and Second day
df['day'] = df['time']/24    # 2 days of data
df['day'] = df['day'].apply(lambda x : 1 if(x==0) else math.ceil(x))
```

In [37]:
```
df[['time','hour','day','amount','class']]
```

Out[37]:

	time	hour	day	amount	class
0	0.000000	0	1	149.62	0
1	0.000000	0	1	2.69	0
2	0.000278	0	1	378.66	0
3	0.000278	0	1	123.50	0
4	0.000556	0	1	69.99	0
...	...	...	...	...	...
273436	47.996111	23	2	0.77	0
273437	47.996389	23	2	24.79	0
273438	47.996667	23	2	67.88	0
273439	47.996667	23	2	10.00	0
273440	47.997778	23	2	217.00	0

273441 rows × 5 columns

Fraud and Genuine transaction Day wise

In [38]:
```
# calculating fraud transaction daywise
dayFrdTran = df[(df['class'] == 1)]['day'].value_counts()
# calculating genuine transaction daywise
dayGenuTran = df[(df['class'] == 0)]['day'].value_counts()
# calculating total transaction daywise
dayTran = df['day'].value_counts()

print("No of transaction Day wise:")
print(dayTran)

print("\n")

print("No of fraud transaction Day wise:")
print(dayFrdTran)

print("\n")

print("No of genuine transactions Day wise:")
print(dayGenuTran)

print("\n")

print("Percentage of fraud transactions Day wise:")
print((dayFrdTran/dayTran)*100)
```

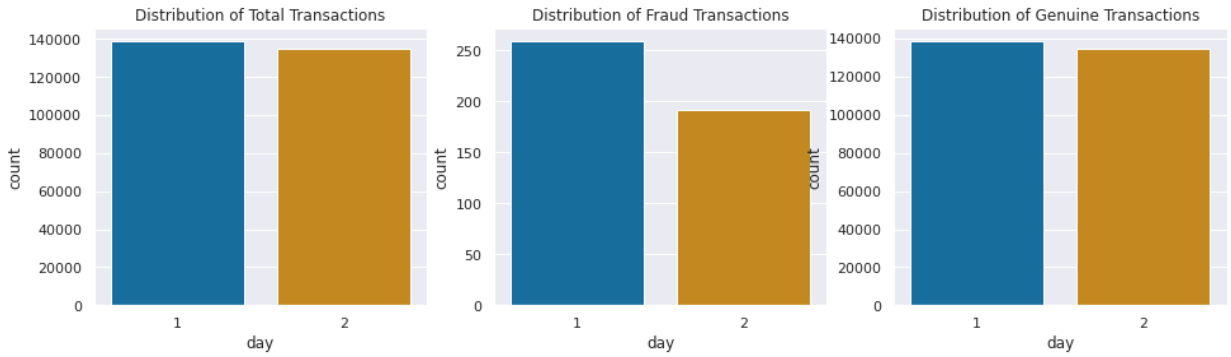
```
No of transaction Day wise:
1    138809
2    134632
Name: day, dtype: int64


No of fraud transaction Day wise:
1    259
2    192
Name: day, dtype: int64


No of genuine transactions Day wise:
1    138550
2    134440
Name: day, dtype: int64


Percentage of fraud transactions Day wise:
1    0.186587
2    0.142611
Name: day, dtype: float64
```

Highlights

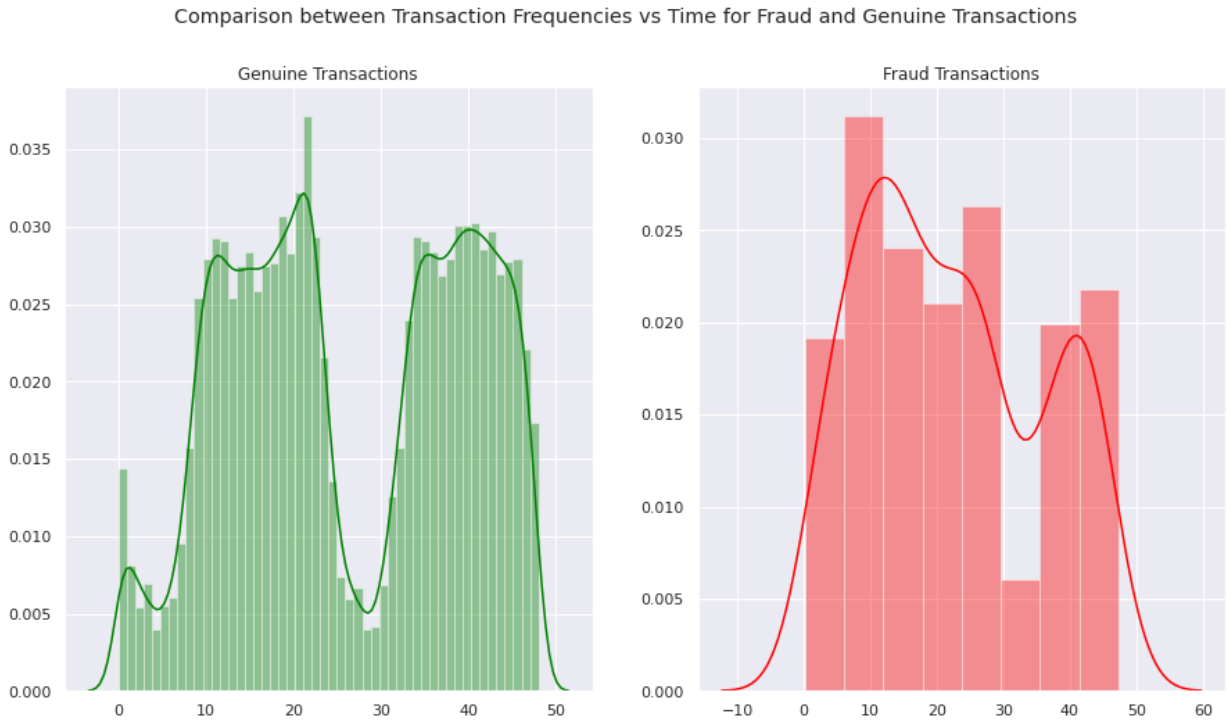
- Total number of transaction on Day 1 was 138809, out of which 259 was a fraud and 138550 was genuine. Fraud transaction was 0.19% of the total transaction on day 1.
- Total number of transaction on Day 2 was 134632, out of which 192 was a fraud and 134440 was genuine. Fraud transaction was 0.14% of the total transaction on day 2.
- Most of the transaction including the fraud transaction happened on day 1.

Let's see the above numbers in the graph.

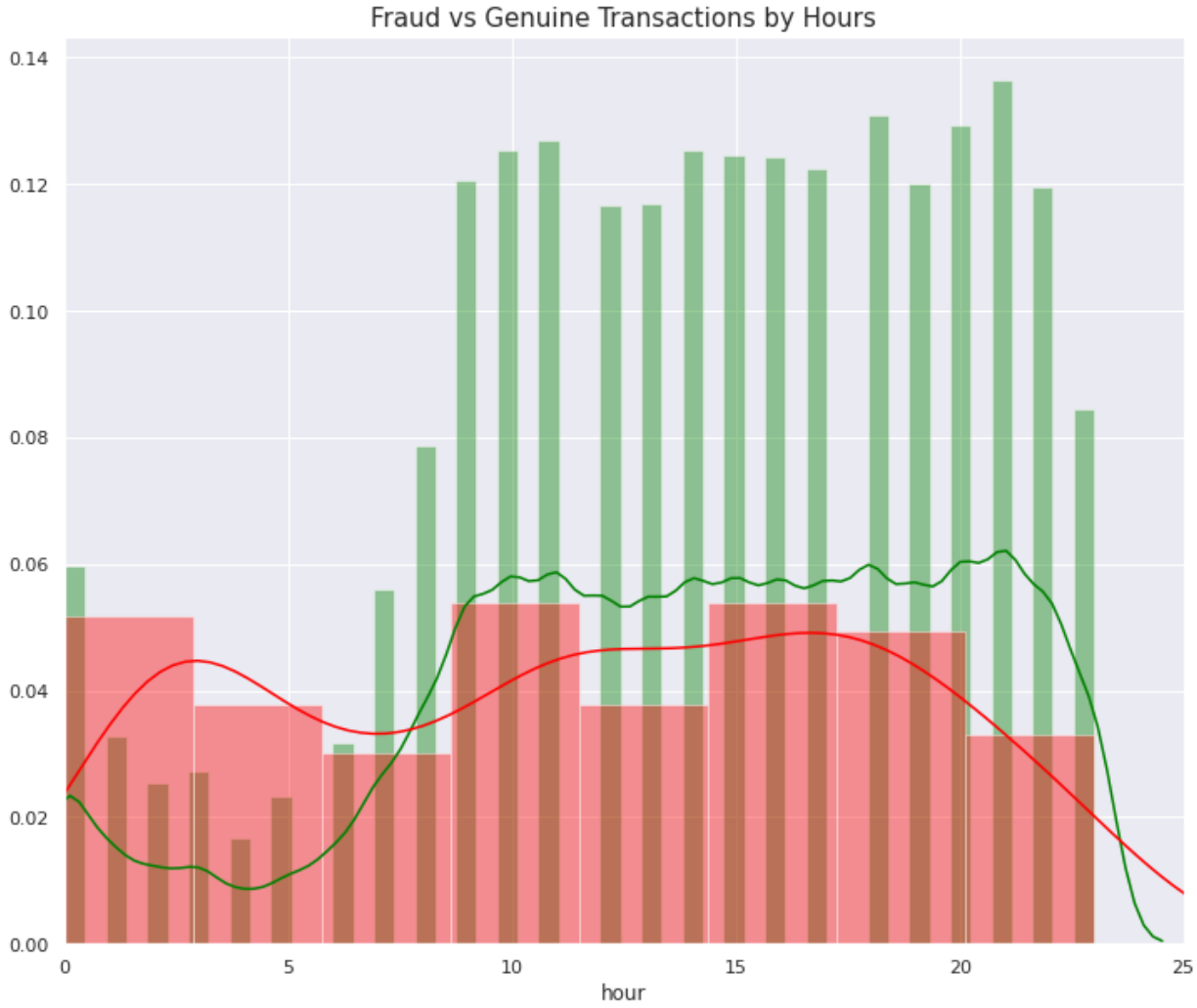
In [39]:
```
fig, axs = plt.subplots(ncols=3, figsize=(16,4))

sns.countplot(df['day'], ax=axs[0])
axs[0].set_title("Distribution of Total Transactions")

sns.countplot(df[(df['class'] == 1)]['day'], ax=axs[1])
axs[1].set_title("Distribution of Fraud Transactions")

sns.countplot(df[(df['class'] == 0)]['day'], ax=axs[2])
axs[2].set_title("Distribution of Genuine Transactions")

plt.show()
```



In [40]:
```
# Time plots
fig , axs = plt.subplots(nrows = 1 , ncols = 2 , figsize = (15,8))

sns.distplot(df[df['class']==0]['time'].values , color = 'green' , ax
= axs[0])
axs[0].set_title('Genuine Transactions')

sns.distplot(df[df['class']==1]['time'].values , color = 'red' ,ax = a
xs[1])
axs[1].set_title('Fraud Transactions')

fig.suptitle('Comparison between Transaction Frequencies vs Time for F
raud and Genuine Transactions')
plt.show()
```



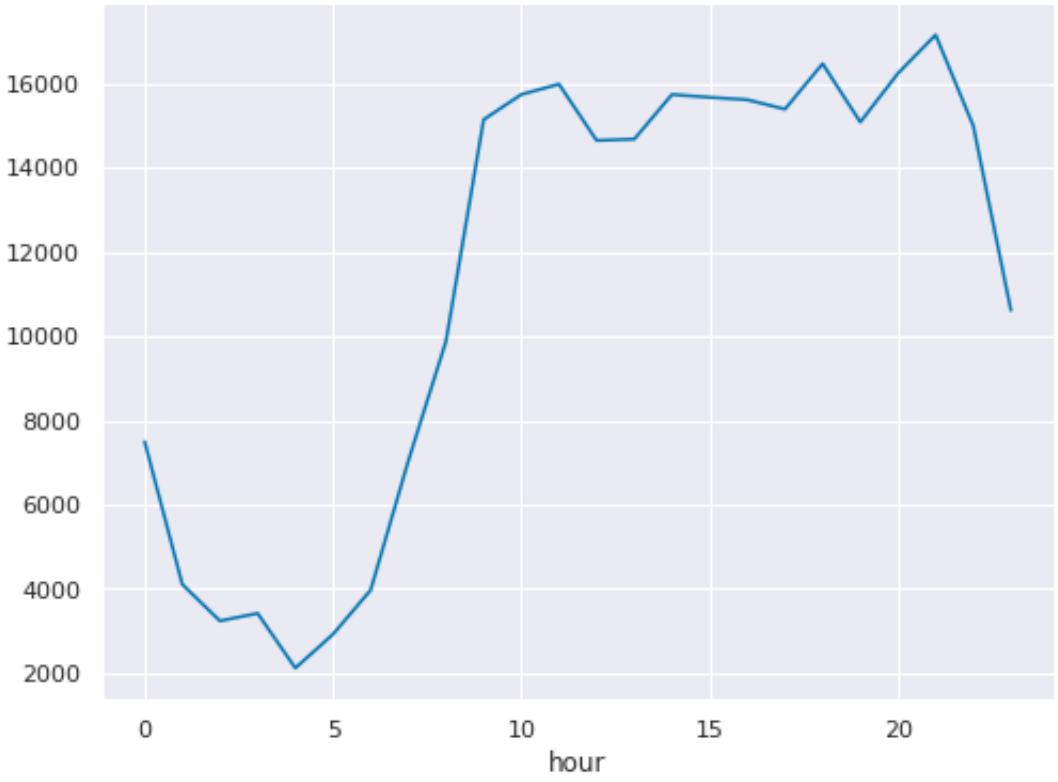
In [41]:
```
# Let's see if we find any particular pattern between time ( in hours ) and Fraud vs Genuine Transactions

plt.figure(figsize=(12,10))

sns.distplot(df[df['class'] == 0]["hour"], color='green') # Genuine - green
sns.distplot(df[df['class'] == 1]["hour"], color='red') # Fraudulent - Red

plt.title('Fraud vs Genuine Transactions by Hours', fontsize=15)
plt.xlim([0,25])
plt.show()
```



Highlights

Above graph shows that most of the Fraud transactions are happening at night time (0 to 7 hours) when most of the people are sleeping and Genuine transaction are happening during day time (9 to 21 hours).

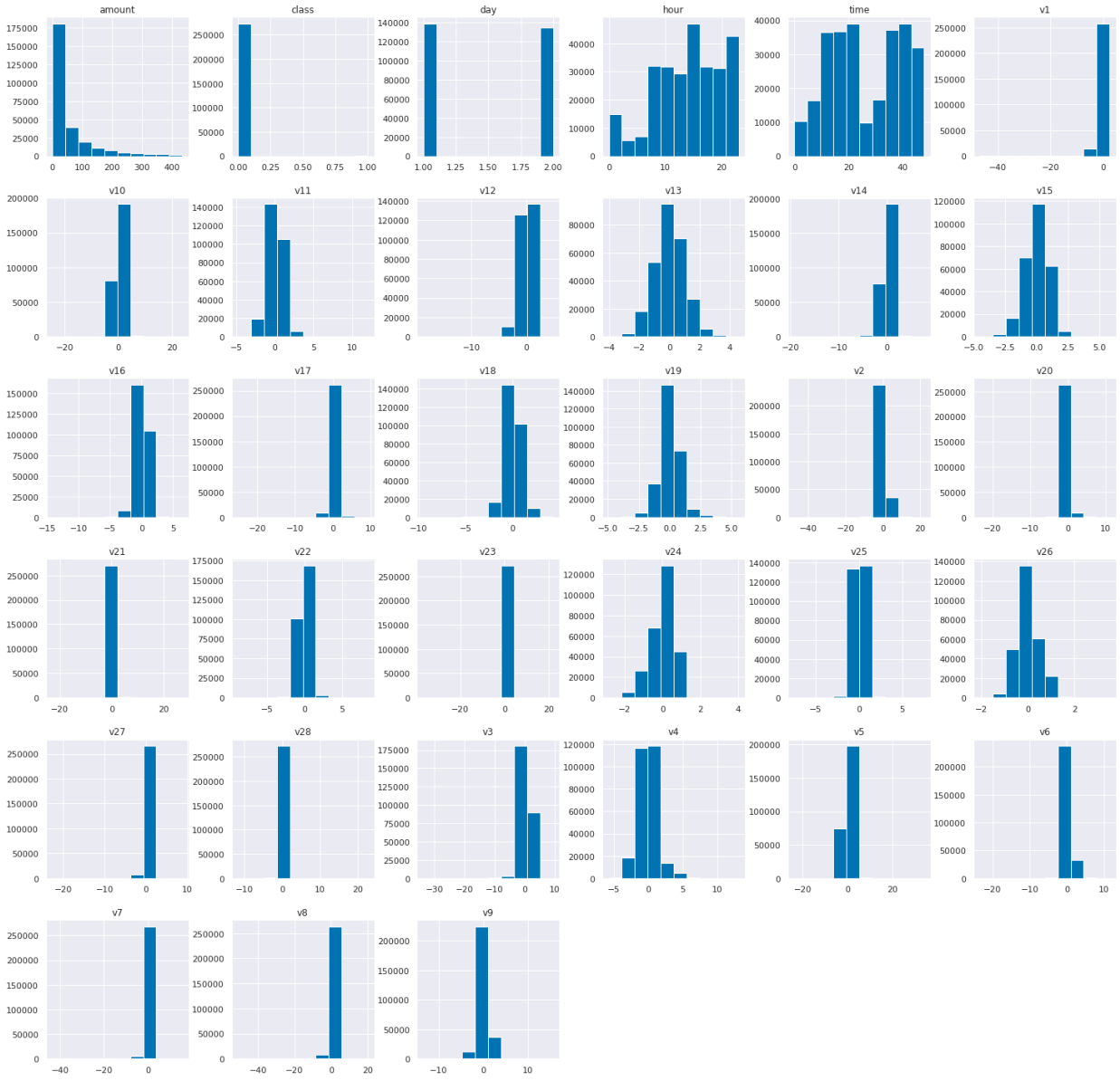
In [42]:
```
plt.figure(figsize=(8,6))
df[['time','hour','day','amount','class']].groupby('hour').count()['class'].plot()
plt.show()
```



Visualising Data for detecting any particular Pattern or Anomaly using Histogram Plots

Finally visulaising all columns once and for all to observe any abnormality

In [43]:
```
df.hist(figsize = (25,25))
plt.show()
```



Reset the index

In [44]:
```
df.reset_index(inplace = True , drop = True)
```

Scale Amount Feature

- It is a good idea to scale the data so that the column(feature) with lesser significance might not end up dominating the objective function due to its larger range. like a column like age has a range between 0 to 80, but a column like a salary has ranged from thousands to lakhs, hence, salary column will dominate to predict the outcome even if it may not be important.
- In addition, features having different unit should also be scaled thus providing each feature equal initial weightage. Like Age in years and Sales in Dollars must be brought down to a common scale before feeding it to the ML algorithm
- This will result in a better prediction model.

PCA Transformation: The description of the data says that all the features went through a PCA transformation (Dimensionality Reduction technique) except for time and amount.

Scaling: Keep in mind that in order to implement a PCA transformation features need to be previously scaled.

1. Scale amount by Log

Scaling using the log: There are two main reasons to use logarithmic scales in charts and graphs.

- The first is to respond to skewness towards large values; i.e., cases in which one or a few points are much larger than the bulk of the data.
- The second is to show per cent change or multiplicative factors.

In [45]:
```
# Scale amount by log
df['amount_log'] = np.log(df.amount + 0.01)
```

2. Scale amount by Standardization

Standardization is another scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

$$z = \frac{x_i - \mu}{\sigma}$$

In [46]:
```
from sklearn.preprocessing import StandardScaler # importing a class f
rom a module of a library

ss = StandardScaler() # object of the class StandardScaler ()
df['amount_scaled'] = ss.fit_transform(df['amount'].values.reshape(-1,
1))
```

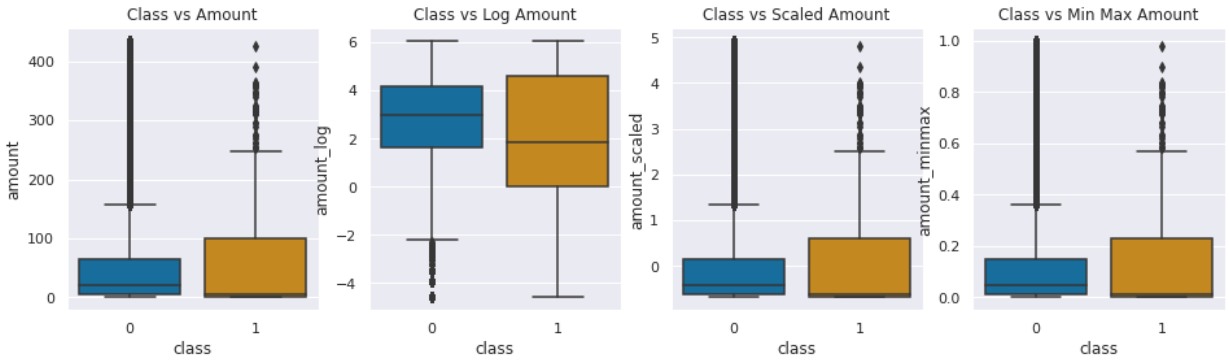
3. Scale amount by Normalization

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

$$x_{norm} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

In [47]:
```
from sklearn.preprocessing import MinMaxScaler

mm = MinMaxScaler() # object of the class StandardScaler ()
df['amount_minmax'] = mm.fit_transform(df['amount'].values.reshape(-1,
1))
```

In [48]:
```
#Feature engineering to a better visualization of the values

# Let's explore the Amount by Class and see the distribuition of Amoun
t transactions
fig , axs = plt.subplots(nrows = 1 , ncols = 4 , figsize = (16,4))

sns.boxplot(x ="class",y="amount",data=df, ax = axs[0])
axs[0].set_title("Class vs Amount")

sns.boxplot(x ="class",y="amount_log",data=df, ax = axs[1])
axs[1].set_title("Class vs Log Amount")

sns.boxplot(x ="class",y="amount_scaled",data=df, ax = axs[2])
axs[2].set_title("Class vs Scaled Amount")

sns.boxplot(x ="class",y="amount_minmax",data=df, ax = axs[3])
axs[3].set_title("Class vs Min Max Amount")

# fig.suptitle('Amount by Class', fontsize=20)
plt.show()
```



Hightlights

- We can see a slight difference in the log amount of our two Classes.
- The IQR of fraudulent transactions are higher than normal transactions, but normal transactions have the highest values.
- **By seeing the above three graphs, I think scaling the amount by log will best suit for our model.**

In [49]:
```
df[['time','hour','day','amount','amount_log','amount_scaled','amount_
minmax','class']]
```

Out[49]:

	time	hour	day	amount	amount_log	amount_scaled	amount_minmax	class
0	0.000000	0	1	149.62	5.008166	1.245932	0.343962	0
1	0.000000	0	1	2.69	0.993252	-0.649372	0.006184	0
2	0.000278	0	1	378.66	5.936665	4.200403	0.870503	0
3	0.000278	0	1	123.50	4.816322	0.909000	0.283915	0
4	0.000556	0	1	69.99	4.248495	0.218755	0.160900	0
...	...	...	...	...	...	...	...	...
273436	47.996111	23	2	0.77	-0.248461	-0.674139	0.001770	0
273437	47.996389	23	2	24.79	3.210844	-0.364296	0.056990	0
273438	47.996667	23	2	67.88	4.217889	0.191537	0.156050	0
273439	47.996667	23	2	10.00	2.303585	-0.555078	0.022989	0
273440	47.997778	23	2	217.00	5.379943	2.115091	0.498862	0

273441 rows × 8 columns

__Saving preprossed data as serialized files__

- To deploy the predictive models built we save them along with the required data files as serialized file objects
- We save cleaned and processed input data, tuned predictive models as files so that they can later be re-used/shared

In [50]:
```
CreditCardFraudDataCleaned = df

# Saving the Python objects as serialized files can be done using pick
le library
# Here let us save the Final Data set after all the transformations as
a file
with open('CreditCardFraudDataCleaned.pkl', 'wb') as fileWriteStream:
    pickle.dump(CreditCardFraudDataCleaned, fileWriteStream)
    # Don't forget to close the filestream!
    fileWriteStream.close()

print('pickle file is saved at Location:',os.getcwd())
```

```
pickle file is saved at Location: /kaggle/working
```

Load preprocessed data

In [51]:
```
# Reading a Pickle file
with open('CreditCardFraudDataCleaned.pkl', 'rb') as fileReadStream:
    CreditCardFraudDataFromPickle = pickle.load(fileReadStream)
    # Don't forget to close the filestream!
    fileReadStream.close()

# Checking the data read from pickle file. It is exactly same as the D
iamondPricesData
df = CreditCardFraudDataFromPickle
df.head()
```

Out[51]:

	time	v1	v2	v3	v4	v5	v6	v7	
0	0.000000	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098
1	0.000000	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085
2	0.000278	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247
3	0.000278	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377
4	0.000556	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270

In [52]:
```
df.shape
```

Out[52]:
```
(273441, 36)
```

In [53]:
```
df.head()
```

Out[53]:

	time	v1	v2	v3	v4	v5	v6	v7	
0	0.000000	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098
1	0.000000	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085
2	0.000278	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247
3	0.000278	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377
4	0.000556	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270

Splitting data into Training and Testing samples

We don't use the full data for creating the model. Some data is randomly selected and kept aside for checking how good the model is. This is known as Testing Data and the remaining data is called Training data on which the model is built. Typically 70% of data is used as training data and the rest 30% is used as testing data.

In [54]:
```
df.columns
```

Out[54]:
```
Index(['time', 'v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8', 'v9',
'v10',
       'v11', 'v12', 'v13', 'v14', 'v15', 'v16', 'v17', 'v18', 'v19'
, 'v20',
       'v21', 'v22', 'v23', 'v24', 'v25', 'v26', 'v27', 'v28', 'amou
nt',
       'class', 'hour', 'day', 'amount_log', 'amount_scaled', 'amoun
t_minmax'],
      dtype='object')
```

Highlights

- We have created few new features like an hour, day, scaled amount. However, these are just for visualization purpose only, not for building the model.

In [55]:
```
# Separate Target Variable and Predictor Variables
# Here I am keeping the log amount and dropping the amount and scaled
amount columns.
X = df.drop(['time','class','hour','day','amount','amount_minmax','amo
unt_scaled'],axis=1)
y = df['class']
```

In [56]:
```
X
```

Out[56]:

	v1	v2	v3	v4	v5	v6	v7	v8
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.09869
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.08510
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.24767
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.37743
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.27053
...	...	...	...	...	...	...	...	..
273436	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.30533
273437	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.29486
273438	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.70841
273439	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.67914
273440	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.41465

273441 rows × 29 columns

In [57]:
```
# Load the library for splitting the data
from sklearn.model_selection import train_test_split
```

In [58]:
```
# Split the data into training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
30, random_state=101)
```

In [59]:
```
# Quick sanity check with the shapes of Training and testing datasets
print("X_train - ",X_train.shape)
print("y_train - ",y_train.shape)
print("X_test - ",X_test.shape)
print("y_test - ",y_test.shape)
```

```
X_train -  (191408, 29)
y_train -  (191408,)
X_test -  (82033, 29)
y_test -  (82033,)
```

__Baseline for models__

We will train four types of classifiers and decide which classifier will be more effective in detecting **fraud transactions**.

Let's Discuss Next Steps -

1 **Classification Models**

- Logistic Regression
- Decision Trees
- Random Forest
- Naive Bayes Classifier

2 **Class Imbalance Solutions**

- Under Sampling
- Over Sampling
- SMOTE
- ADASYN

3 **Metrics**

- Accuracy Score
- Confusion Matrix
- Precision Score
- Recall Score
- ROC_AUC
- F1 Score

__Model Building__

We are aware that our dataset is highly imbalanced, however, we check the performance of imbalance dataset first and later we implement some techniques to balance the dataset and again check the performance of balanced dataset. Finally, we will compare each regression models performance.

__1. Logistic Regression__

1.1 Logistic Regression with __imbalanced__ data

In [60]:
```
from sklearn.linear_model import LogisticRegression # Importing Classifier Step
```

In [61]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)


logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

Out[61]:
```
LogisticRegression()
```

Predict from Test set

In [62]:
```
y_pred = logreg.predict(X_test)
```

Model Evolution

In [63]:
```
from sklearn import metrics
```

In [64]:
```
# https://en.wikipedia.org/wiki/Precision_and_recall
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     81897
           1       0.87      0.62      0.72       136

    accuracy                           1.00     82033
   macro avg       0.93      0.81      0.86     82033
weighted avg       1.00      1.00      1.00     82033
```

In [65]:
```
print('Accuracy :{0:0.5f}'.format(metrics.accuracy_score(y_pred , y_te
st)))
print('AUC : {0:0.5f}'.format(metrics.roc_auc_score(y_test , y_pred)))
print('Precision : {0:0.5f}'.format(metrics.precision_score(y_test , y
_pred)))
print('Recall : {0:0.5f}'.format(metrics.recall_score(y_test , y_pred)
))
print('F1 : {0:0.5f}'.format(metrics.f1_score(y_test , y_pred)))
# print('Confusion Matrix : \n', cnf_matrix)
print("\n")
```

```
Accuracy :0.99921
AUC : 0.80874
Precision : 0.86598
Recall : 0.61765
F1 : 0.72103
```

In [66]:
```
# Predicted values counts for fraud and genuine of test dataset
pd.Series(y_pred).value_counts()
```

Out[66]:
```
0    81936
1       97
dtype: int64
```

Highlights

Our model predicted 97 transactions as fraud and 81936 transactions as genuine from the test dataset.

In [67]:
```
# Actual values counts for fraud and genuine of test dataset
pd.Series(y_test).value_counts()
```

Out[67]:
```
0    81897
1      136
Name: class, dtype: int64
```

There are originally 136 fraud transactions and our model predicted only 97 fraud transaction. So the accuracy of our model should be $\frac{97}{136}$, right?

In [68]:
```
97/136
```

Out[68]:
```
0.7132352941176471
```

So 71.533% should be our accuracy.

However, this not the case. Actually there are originally 136 fraud transactions and 81897 genuine transactions in the test dataset. However, our model predicted only 97 fraud transaction. Also, it should be kept in mind that these 97 predicted fraud transaction may not be identified correctly. It means that these predicted 97 fraud transactions are NOT only from 136 originally fraud transaction, but they may also be from genuine transactions as well.

We will see our real accuracy in below cells.

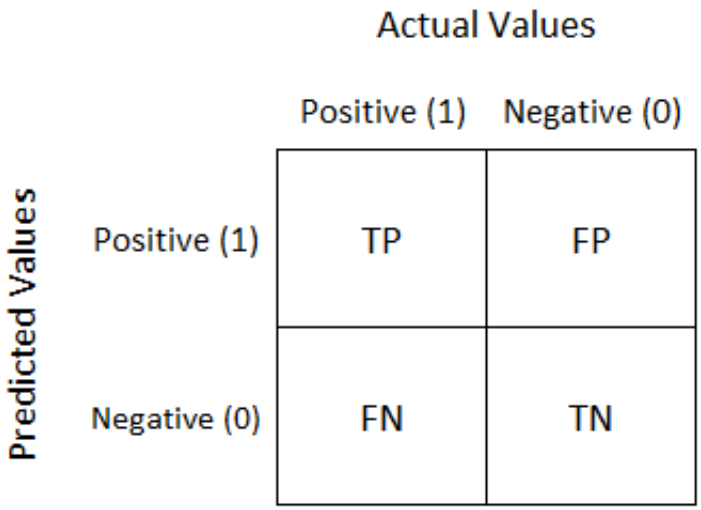
__Model Evolution Matrix__

Every problem is different and derives a different set of values for a particular business use case , thus every model must be evaluated differently.

Let's get to know the terminology and Structure first

A confusion matrix is defined into four parts : **{ TRUE , FALSE } (Actual) ,{POSITIVE , NEGATIVE} (Predicted)** Positive and Negative is what you predict , True and False is what you are told

Which brings us to 4 relations : True Positive , True Negative , False Positive , False Negative
P redicted - **R** ows and **A** ctual as **C** olumns



Accuracy , Precision and Recall

Accuracy : The most used and classic classification metric : Suited for binary classification problems.

$$\text{Accuracy} = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

Basically Rightly predicted results amongst all the results , used when the classes are balanced

Precision : What proportion of predicted positives are truly positive ? Used when we need to predict the positive thoroughly, sure about it !

$$\text{Precision} = \frac{(TP)}{(TP + FP)}$$

Sensitivity or Recall : What proportion of actual positives is correctly classified ? choice when we want to capture as many positives as possible

$$\text{Recall} = \frac{(TP)}{(TP + FN)}$$

F1 Score : Harmonic mean of Precision and Recall. It basically maintains a balance between the precision and recall for your classifier

$$F1 = \frac{2 * (\text{ precision } * \text{ recall })}{(\text{ precision } + \text{ recall })}$$

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

$$F1 = \frac{2 \times 0.3 \times 0.1}{0.3 + 0.1} \quad \therefore F1=0.15$$

Harmonic mean is conservative mean compared to Arithmetic meand and geometric mean.
It means that Harmonic mean is nearest to the smallest of the input numbers.

Precision as the name says, says how precise (how sure) is our model in detecting fraud transactions while recall is the amount of fraud cases our model is able to detect.

In reference of our case:

Recall (True Positive Rate): % of all fraudulent transactions cases captured.

Precision: Out of all items labeled as fraud, what percentage of them is actually fraud?

Accuracy: How correct the model is (misleading for fraud/imbalanced data)

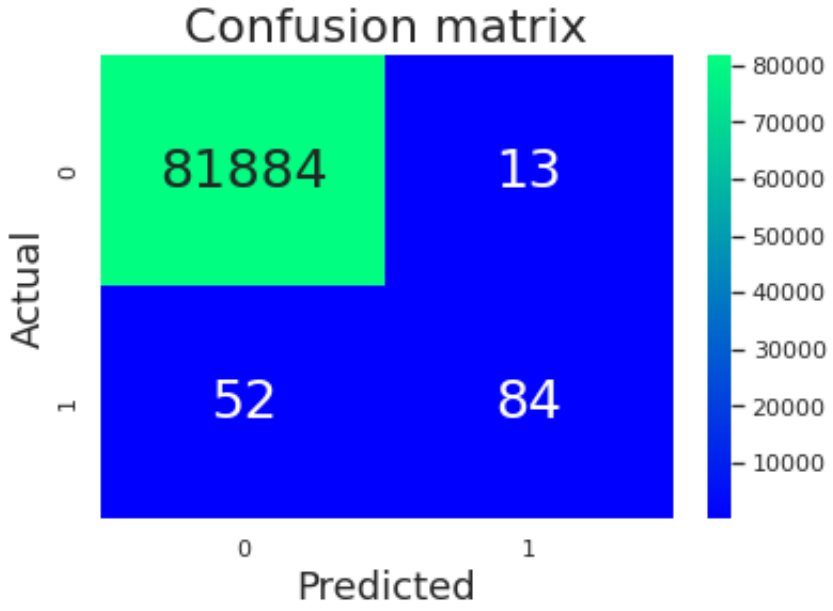
F1 score: combination of recall and precision into one metric. F1 score is the weighted average of precision and recall, taking BOTH false positives and false negatives into account. Usually much more useful than accuracy, especially with uneven classes.

Confusion Matrix

In [69]:
```
cnf_matrix = metrics.confusion_matrix(y_test,y_pred)
cnf_matrix
```

Out[69]:
```
array([[81884,    13],
       [   52,    84]])
```

In [70]:
```
# Heatmap for Confusion Matrix
p = sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, annot_kws={"size
": 25}, cmap="winter" ,fmt='g')

plt.title('Confusion matrix', y=1.1, fontsize = 22)
plt.ylabel('Actual',fontsize = 18)
plt.xlabel('Predicted',fontsize = 18)

# ax.xaxis.set_ticklabels(['Genuine', 'Fraud']);
# ax.yaxis.set_ticklabels(['Genuine', 'Fraud']);

plt.show()
```



There are 84 transaction recognised as True Postive, means they are orignally fraud transactions and our model precited them as fraud.

True Negative - 81884 (truely saying negative - genuine transaction correctly identified as genuine)

True Postive - 84 (truely saying positive - fraud transaction correctly identified as fraud)

False Negative - 52 (falsely saying negative - fraud transaction incorrectly identified as genuine)

False Positive - 13 (falsely saying positive - genuine transaction incorrectly identified as fraud)

We already know that we have 136 fraud transaction in our test dataset, but our model predicted only 84 fraud transaction. So the real accuracy of our model is $\frac{84}{136}$

```
In [71]: 84/136
```

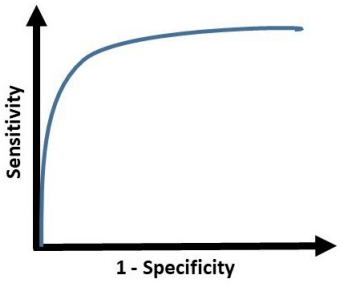
```
Out[71]: 0.6176470588235294
```

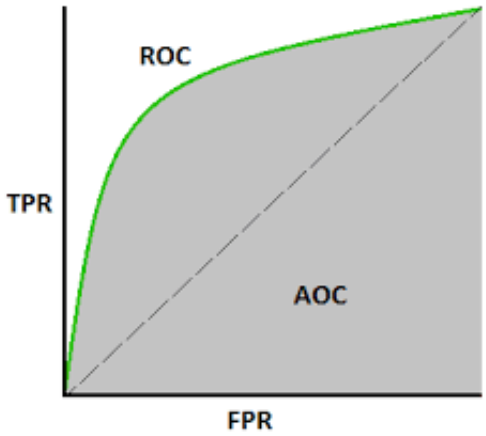
So, **61.77%** is the real accuracy of our model, which is nothing but the **Recall Score**. So we have the emphasis on Recall score and F1 score to measure the performance of our model, not the accuracy.

__Receiver Operating Characteristics (ROC)__

The ROC is a performance measurement for classification problems at various thresholds. It is essentially a probability curve, and the higher the Area Under the Curve (AUC) score the better the model is at predicting fraudulent/non-fraudulent transactions.

It is an evaluation metric that helps identify the strength of the model to distinguish between two outcomes. It defines if a model can create a clear boundary between the postive and the negative class.



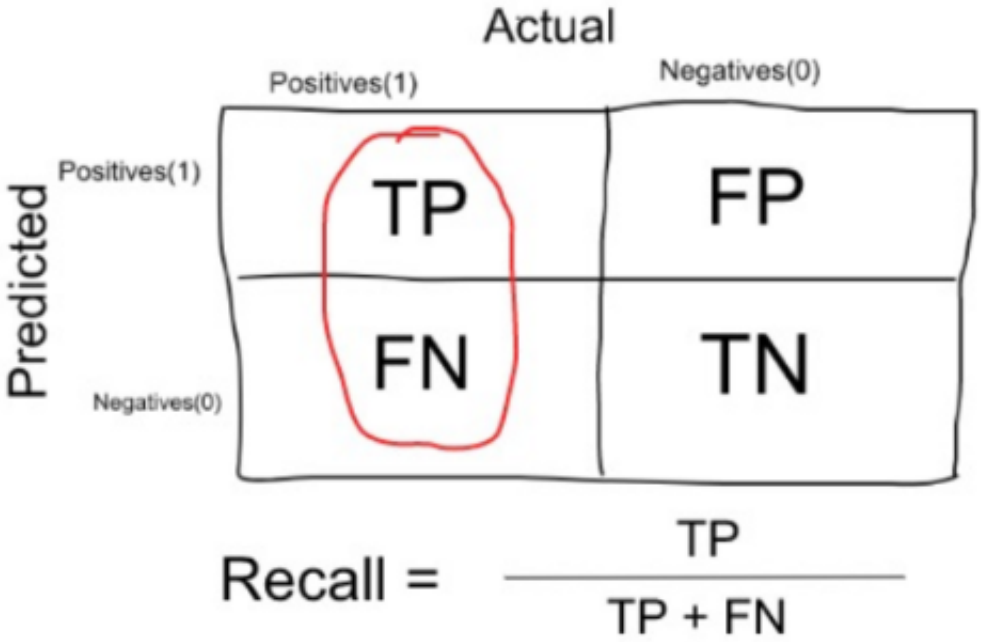


Let's talk about some definitions first:

Sensitivity or Recall

The sensitivity of a model is defined by the proportion of actual positives that are classified as Positives , i.e = TP / (TP + FN)

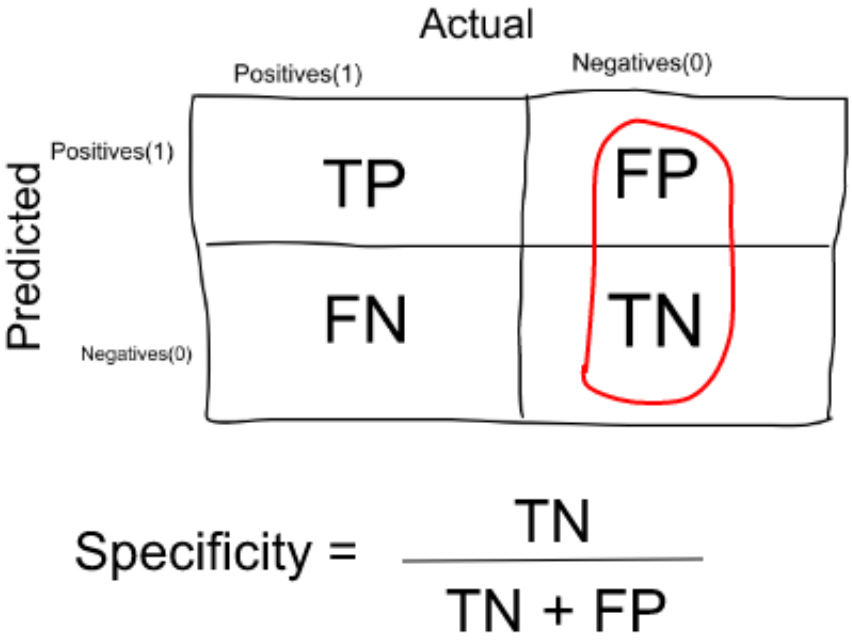
$$\text{Recall or Sensitivity} = \frac{(TP)}{(TP + FN)}$$



Specificity

The specificity of a model is defined by the proportion of actual negatives that are classified as Negatives , i.e = TN / (TN + FP)

$$\text{Specificity} = \frac{(TN)}{(TN + FP)}$$

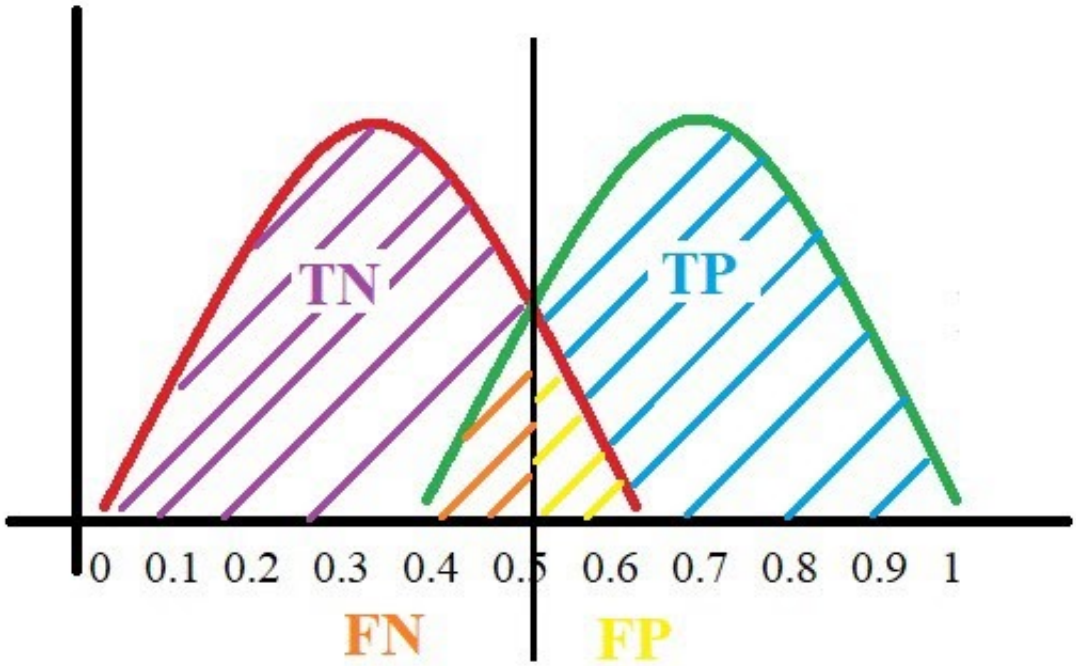


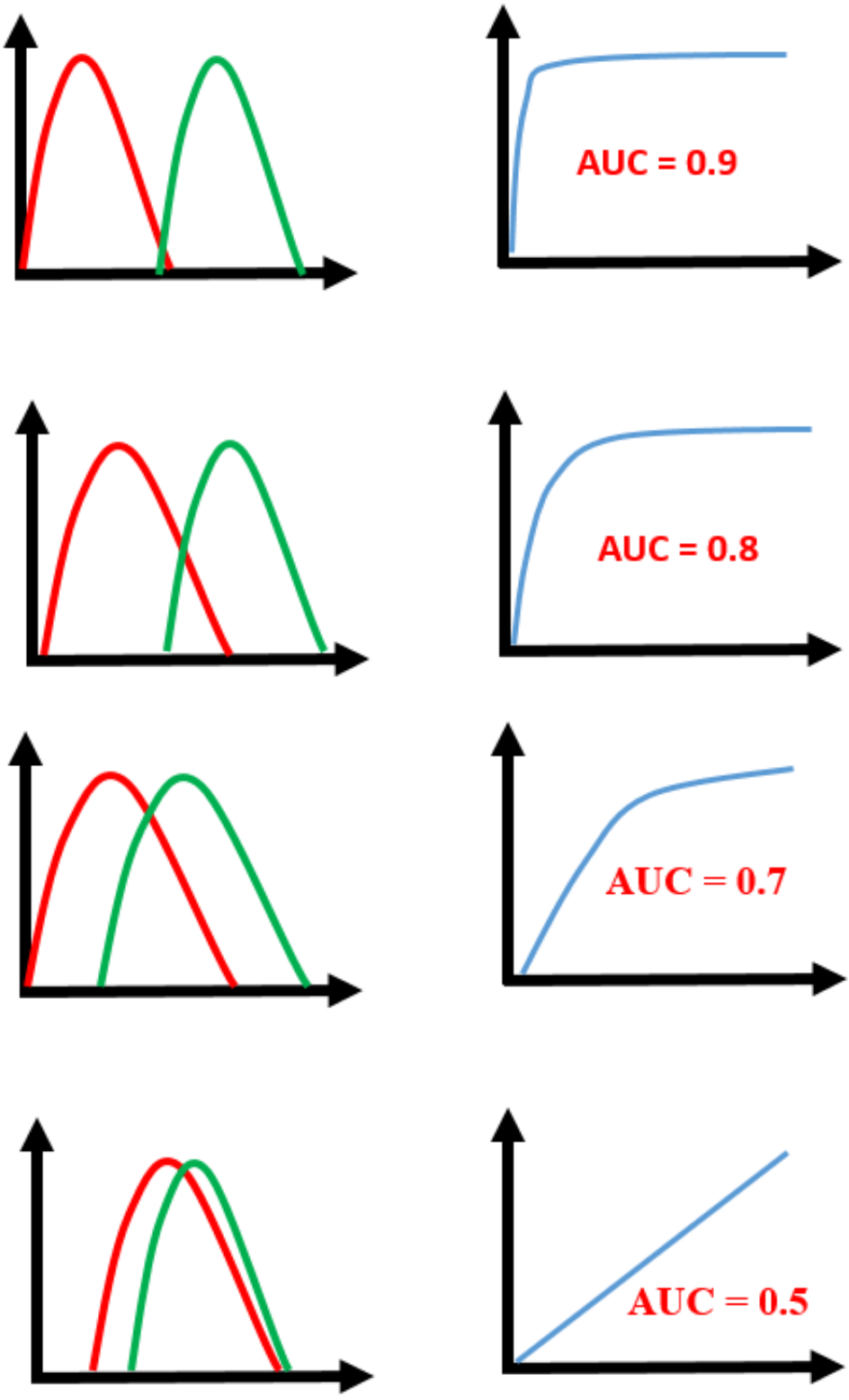
As we can see that both are independent of each other and lie in teo different quadrants , we can understand that they are inversely related to each other. Thus as Sensitivity goes up , Specificity goes down and vice versa.

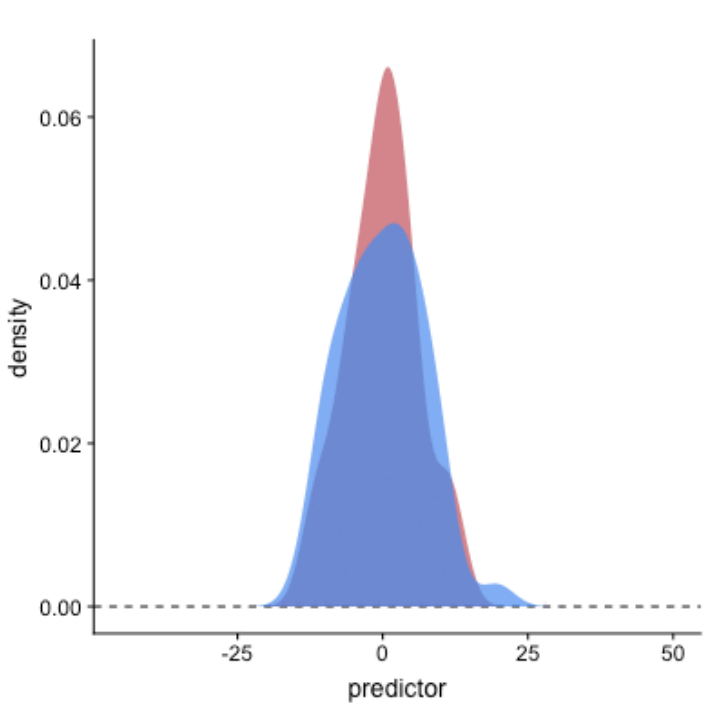
ROC CURVE

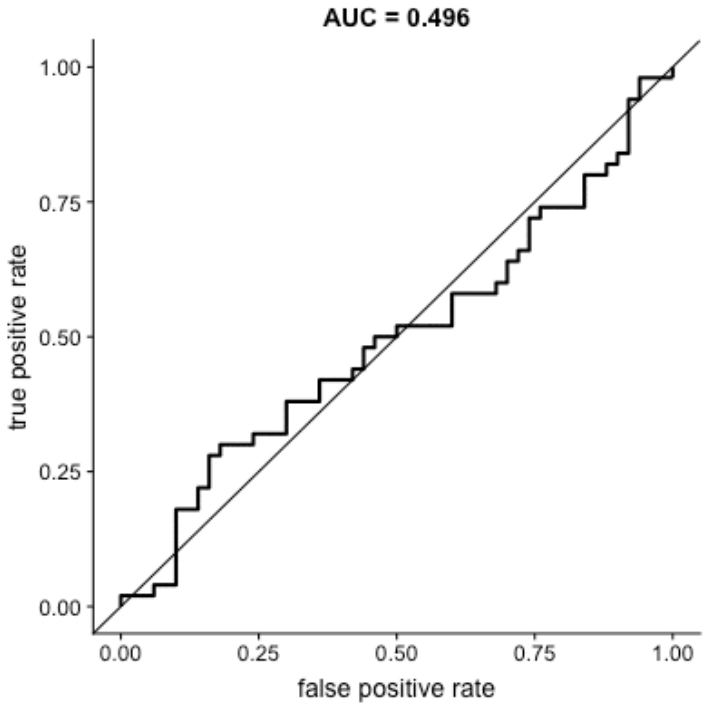
It is a plot between Sesitivity and (1 - Specificity) , which intuitively is a plot between True Positive Rate and False Positive Rate. It depicts if a model can clearly identify each class or not

Higher the area under the curve , better the model and it's ability to seperate the positive and negative class.




AUC = 0.9
AUC = 0.8
AUC = 0.7
AUC = 0.5





In [72]:
```
metrics.roc_auc_score(y_test , y_pred)
```

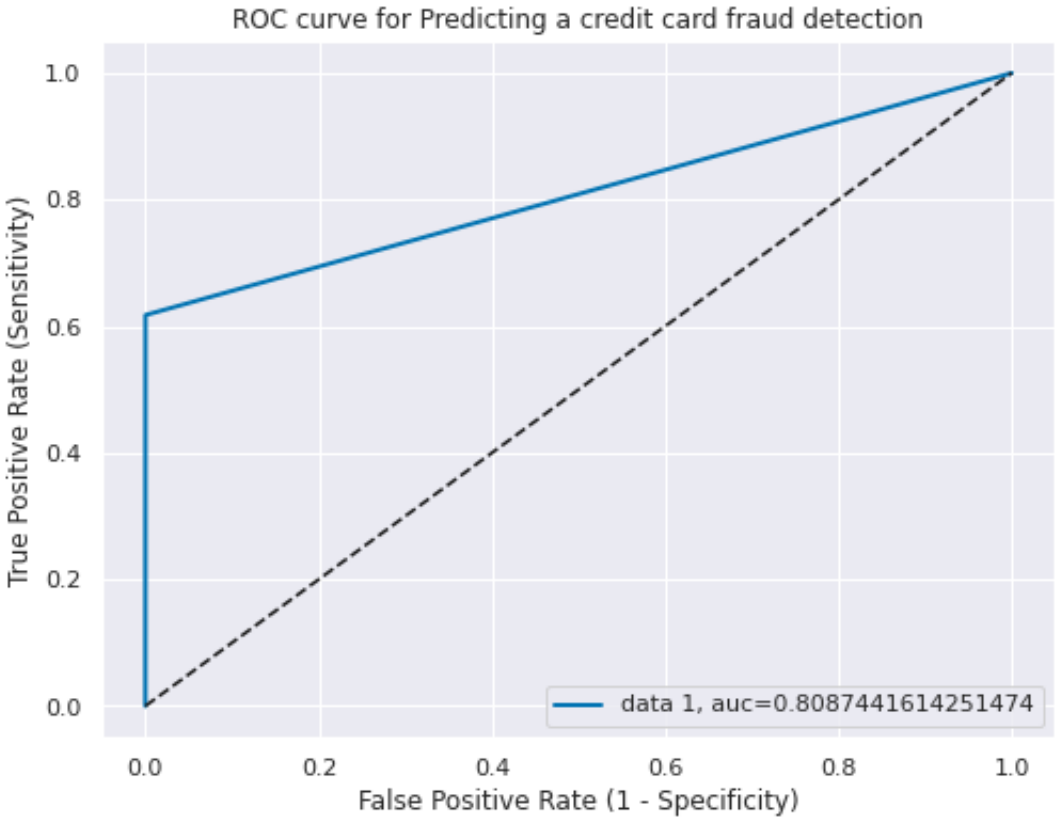
Out[72]:
```
0.8087441614251474
```

In [73]:
```
y_pred_proba = logreg.predict_proba(X_test)
y_pred_proba
```

Out[73]:
```
array([[9.99980849e-01, 1.91513420e-05],
       [9.99989046e-01, 1.09538269e-05],
       [9.99832228e-01, 1.67772145e-04],
       ...,
       [9.99982585e-01, 1.74150469e-05],
       [9.99204300e-01, 7.95699728e-04],
       [9.99814095e-01, 1.85904727e-04]])
```

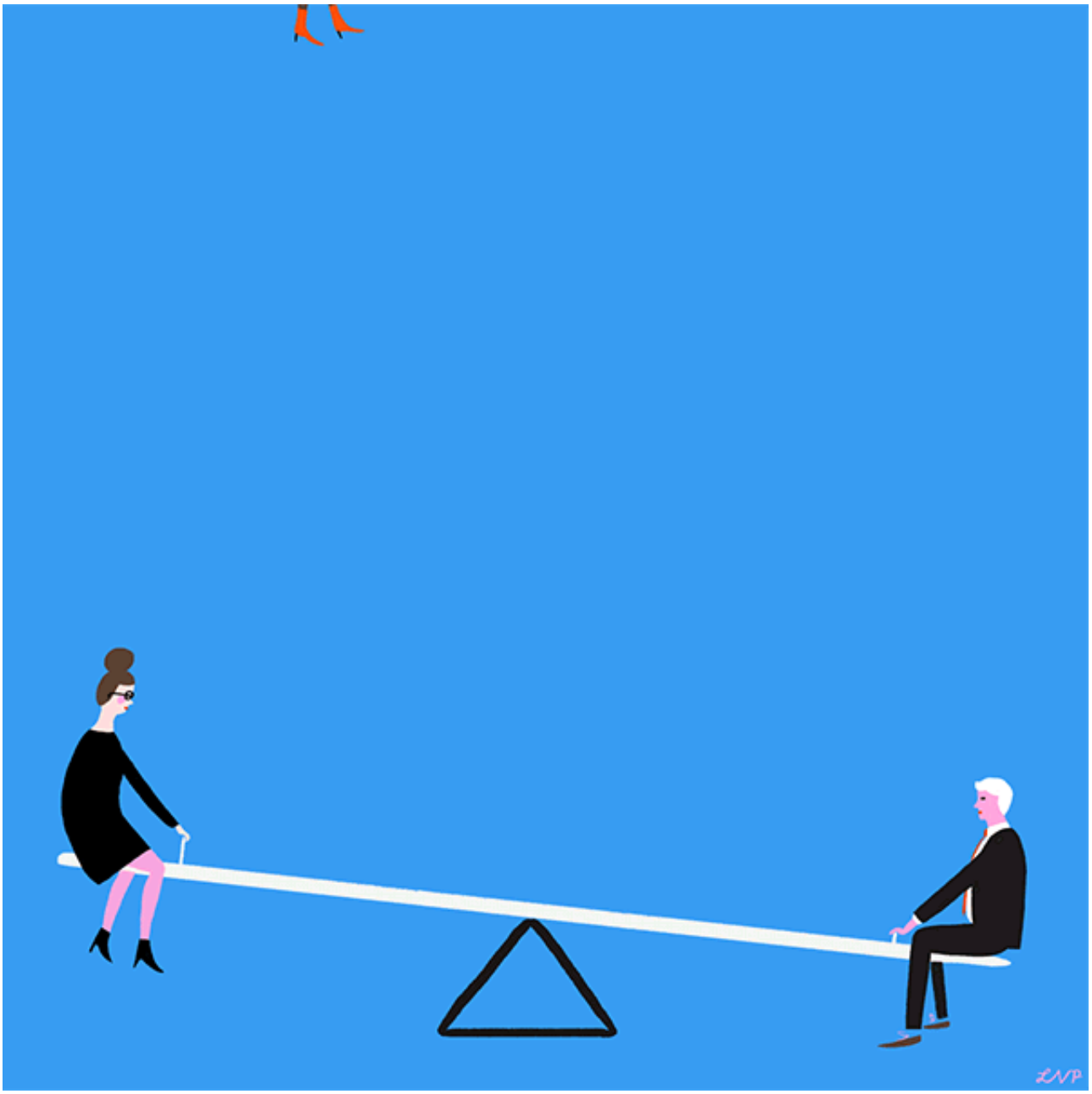
In [74]:
```
# plot ROC Curve

plt.figure(figsize=(8,6))

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)

auc = metrics.roc_auc_score(y_test, y_pred)
print("AUC - ",auc,"\n")

plt.plot(fpr,tpr,linewidth=2, label="data 1, auc="+str(auc))
plt.legend(loc=4)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12
plt.title('ROC curve for Predicting a credit card fraud detection')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```

```
AUC -  0.8087441614251474
```



__Class Imbalance__

Imbalanced data typically refers to a problem with classification problems where the classes are not represented equally. If one applies classifiers on the dataset, they are likely to predict everything as the majority class. This was often regarded as a problem in learning from highly imbalanced datasets.



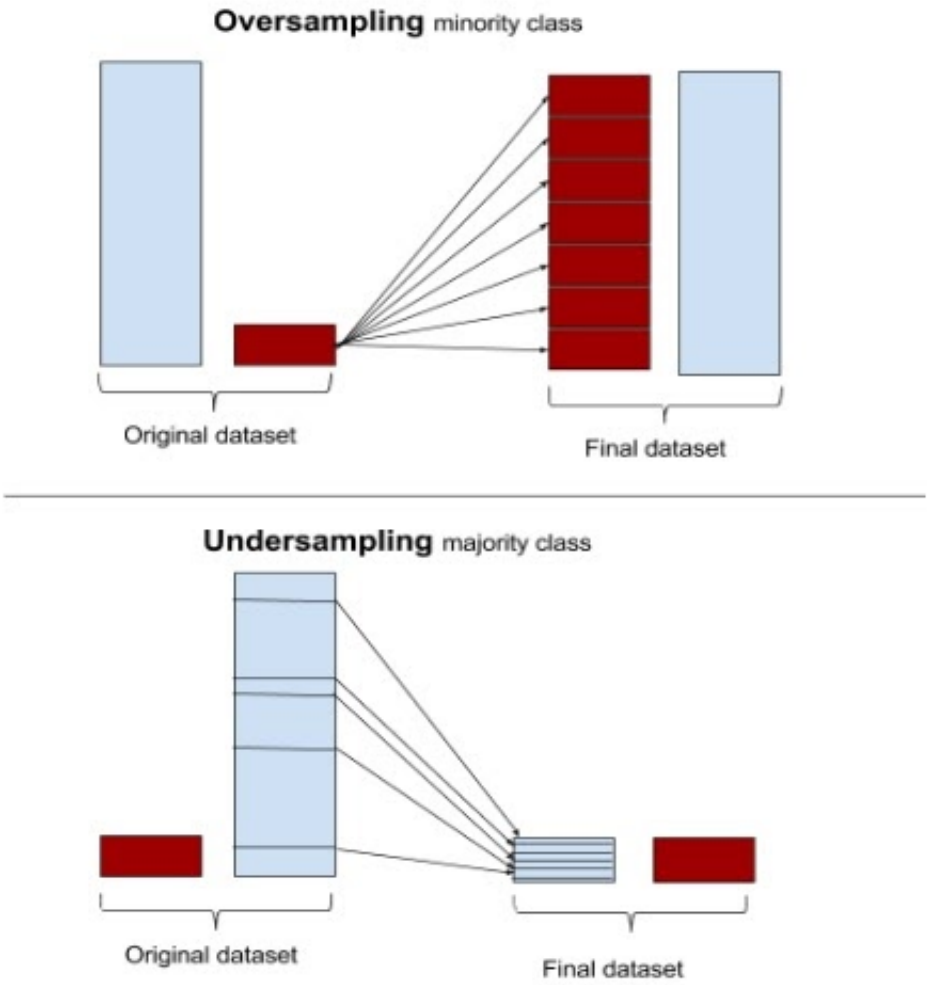
Let's Fix the class Imbalance and apply some sampling techniques.

Ref : [https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/ (https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/)](https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/)

Under Sampling and Over Sampling

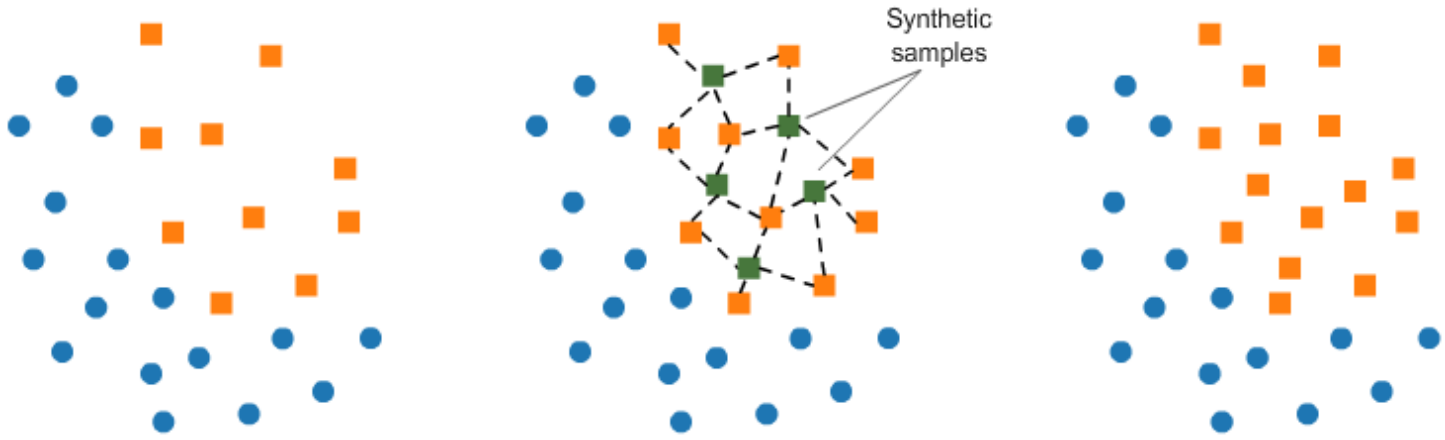
Oversampling and undersampling in data analysis are techniques used to adjust the class distribution of a data set.

- Random oversampling duplicates examples from the minority class in the training dataset and can result in overfitting for some models.
- Random undersampling deletes examples from the majority class and can result in losing information invaluable to a model.



Synthetic Minority OverSampling Technique (SMOTE)

In this technique, instead of simply duplicating data from the minority class, we synthesize new data from the minority class. This is a type of data augmentation for tabular data can be very effective. This approach to synthesizing new data is called the Synthetic Minority Oversampling TEchnique, or SMOTE for short.



Adaptive Synthetic Sampling Method for Imbalanced Data (ADASYN)

ADASYN (Adaptive Synthetic) is an algorithm that generates synthetic data, and its greatest advantages are not copying the same minority data, and generating more data for “harder to learn” examples.

Ref : https://medium.com/@ruinian/an-introduction-to-adasyn-with-code-1383a5ece7aa (https://medium.com/@ruinian/an-introduction-to-adasyn-with-code-1383a5ece7aa)

Import imbalace technique algorithims

In [75]:
```
# Import imbalace technique algorithims
from sklearn.metrics import precision_score, recall_score, f1_score, r
oc_auc_score, accuracy_score, classification_report
from imblearn.over_sampling import SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
```

1.2.Logistic Regression with Random Undersampling technique

In [76]:
```
from collections import Counter # counter takes values returns value_c
ounts dictionary
from sklearn.datasets import make_classification
```

In [77]:
```
print('Original dataset shape %s' % Counter(y))

rus = RandomUnderSampler(random_state=42)
X_res, y_res = rus.fit_resample(X, y)

print('Resampled dataset shape %s' % Counter(y_res))
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 451, 1: 451})
```

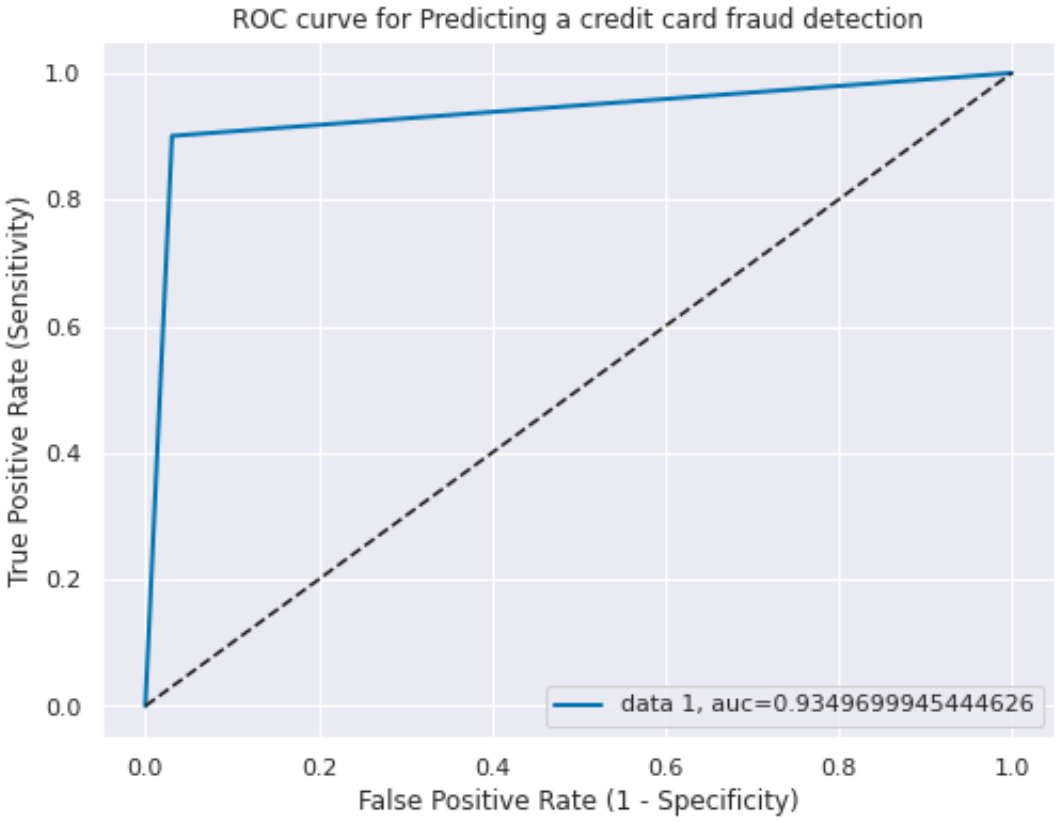
In [78]:
```
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test
_size=0.3, random_state=0)

# Undersampling with Logistic Regression
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
```

In [79]:
```
print('Accuracy :{0:0.5f}'.format(metrics.accuracy_score(y_pred , y_te
st)))
print('AUC : {0:0.5f}'.format(metrics.roc_auc_score(y_test , y_pred)))
print('Precision : {0:0.5f}'.format(metrics.precision_score(y_test , y
_pred)))
print('Recall : {0:0.5f}'.format(metrics.recall_score(y_test , y_pred)
))
print('F1 : {0:0.5f}'.format(metrics.f1_score(y_test , y_pred)))
```

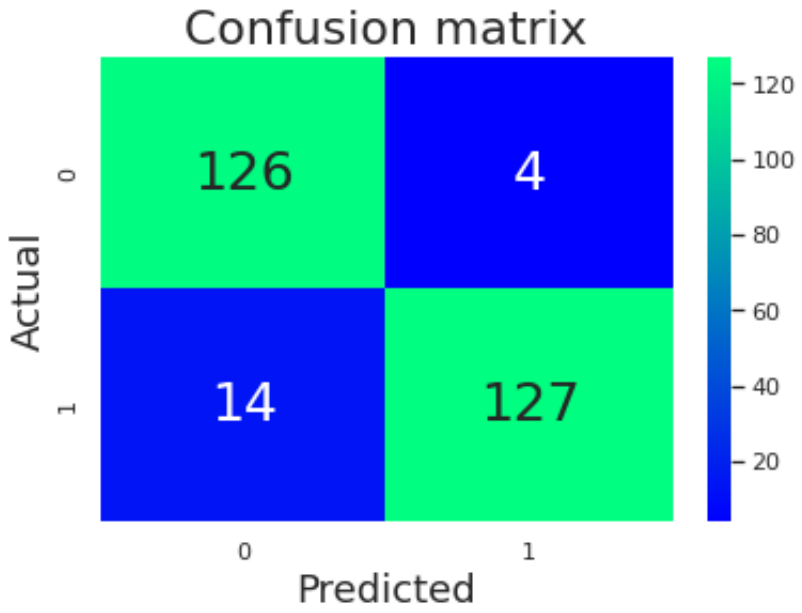
```
Accuracy :0.93358
AUC : 0.93497
Precision : 0.96947
Recall : 0.90071
F1 : 0.93382
```

In [80]:
```
# plot ROC Curve

plt.figure(figsize=(8,6))

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)

auc = metrics.roc_auc_score(y_test, y_pred)
print("AUC - ",auc,"\n")

plt.plot(fpr,tpr,linewidth=2, label="data 1, auc="+str(auc))
plt.legend(loc=4)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12
plt.title('ROC curve for Predicting a credit card fraud detection')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```

```
AUC -  0.9349699945444626
```



In [81]:
```
# Heatmap for Confusion Matrix

cnf_matrix = metrics.confusion_matrix(y_test , y_pred)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, annot_kws={"size": 25}, cmap="winter" ,fmt='g')

plt.title('Confusion matrix', y=1.1, fontsize = 22)
plt.xlabel('Predicted',fontsize = 18)
plt.ylabel('Actual',fontsize = 18)

# ax.xaxis.set_ticklabels(['Genuine', 'Fraud']);
# ax.yaxis.set_ticklabels(['Genuine', 'Fraud']);

plt.show()
```



1.3.Logistic Regression with Random Oversampling technique

In [82]:
```
from imblearn.over_sampling import RandomOverSampler
```

In [83]:
```
print('Original dataset shape %s' % Counter(y))
random_state = 42

ros = RandomOverSampler(random_state=random_state)
X_res, y_res = ros.fit_resample(X, y)

print('Resampled dataset shape %s' % Counter(y_res))
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 272990, 1: 272990})
```

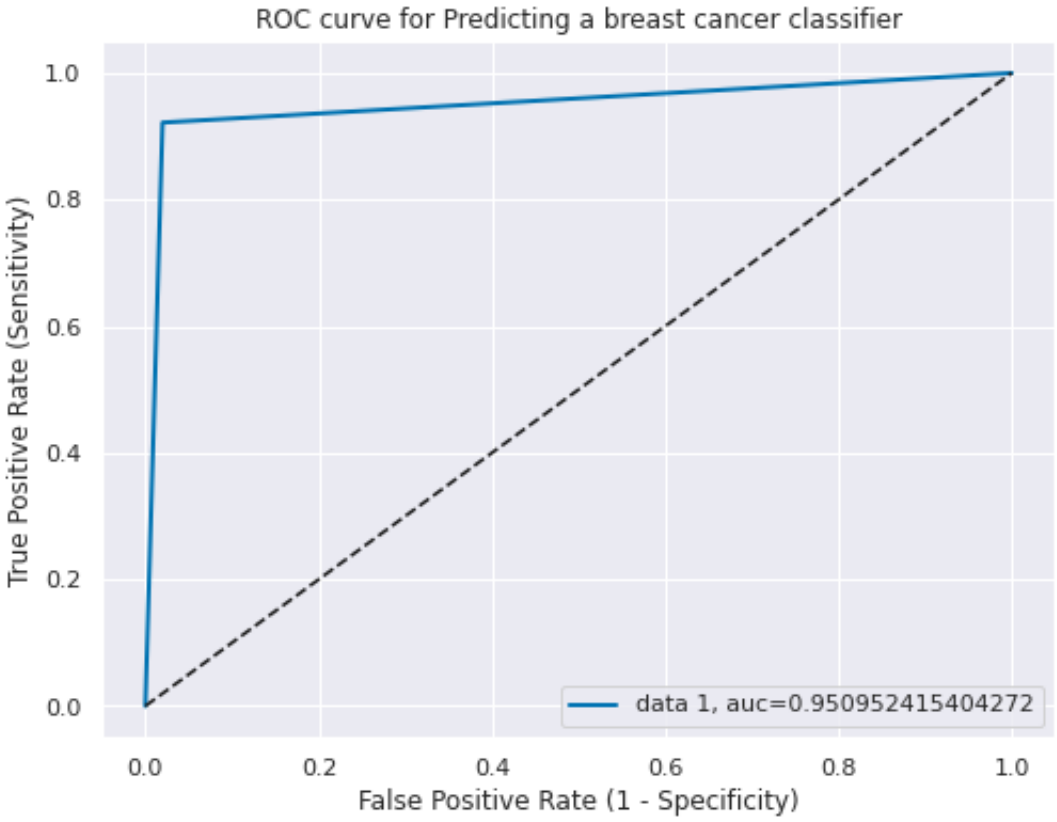
In [84]:
```
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test
_size=0.3, random_state=0)

# Oversampling with Logistic Regression
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
```

In [85]:
```
print('Accuracy :{0:0.5f}'.format(metrics.accuracy_score(y_pred , y_te
st)))
print('AUC : {0:0.5f}'.format(metrics.roc_auc_score(y_test , y_pred)))
print('Precision : {0:0.5f}'.format(metrics.precision_score(y_test , y
_pred)))
print('Recall : {0:0.5f}'.format(metrics.recall_score(y_test , y_pred)
))
print('F1 : {0:0.5f}'.format(metrics.f1_score(y_test , y_pred)))
```

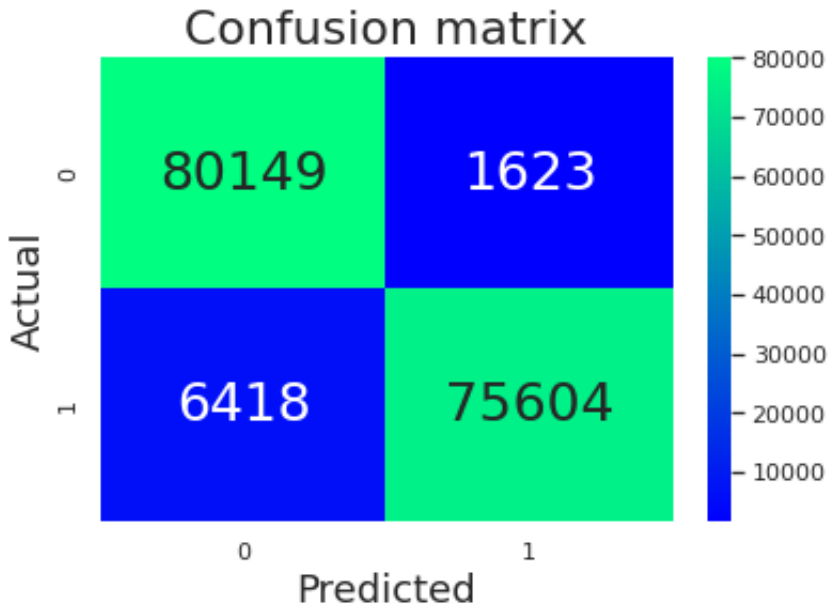
```
Accuracy :0.95091
AUC : 0.95095
Precision : 0.97898
Recall : 0.92175
F1 : 0.94951
```

In [86]:
```
# plot ROC Curve

plt.figure(figsize=(8,6))

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)

auc = metrics.roc_auc_score(y_test, y_pred)
print("AUC - ",auc,"\n")

plt.plot(fpr,tpr,linewidth=2, label="data 1, auc="+str(auc))
plt.legend(loc=4)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12
plt.title('ROC curve for Predicting a breast cancer classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```

```
AUC -  0.950952415404272
```



In [87]:
```
# Heatmap for Confusion Matrix

cnf_matrix = metrics.confusion_matrix(y_test , y_pred)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, annot_kws={"size": 25}, cmap="winter" ,fmt='g')

plt.title('Confusion matrix', y=1.1, fontsize = 22)
plt.xlabel('Predicted',fontsize = 18)
plt.ylabel('Actual',fontsize = 18)

# ax.xaxis.set_ticklabels(['Genuine', 'Fraud']);
# ax.yaxis.set_ticklabels(['Genuine', 'Fraud']);

plt.show()
```



1.4 Logistic Regression with __SMOTE__ data

In [88]:
```
from imblearn.over_sampling import SMOTE, ADASYN
```

In [89]:
```
print('Original dataset shape %s' % Counter(y))

smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X, y)

print('Resampled dataset shape %s' % Counter(y_res))
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 272990, 1: 272990})
```

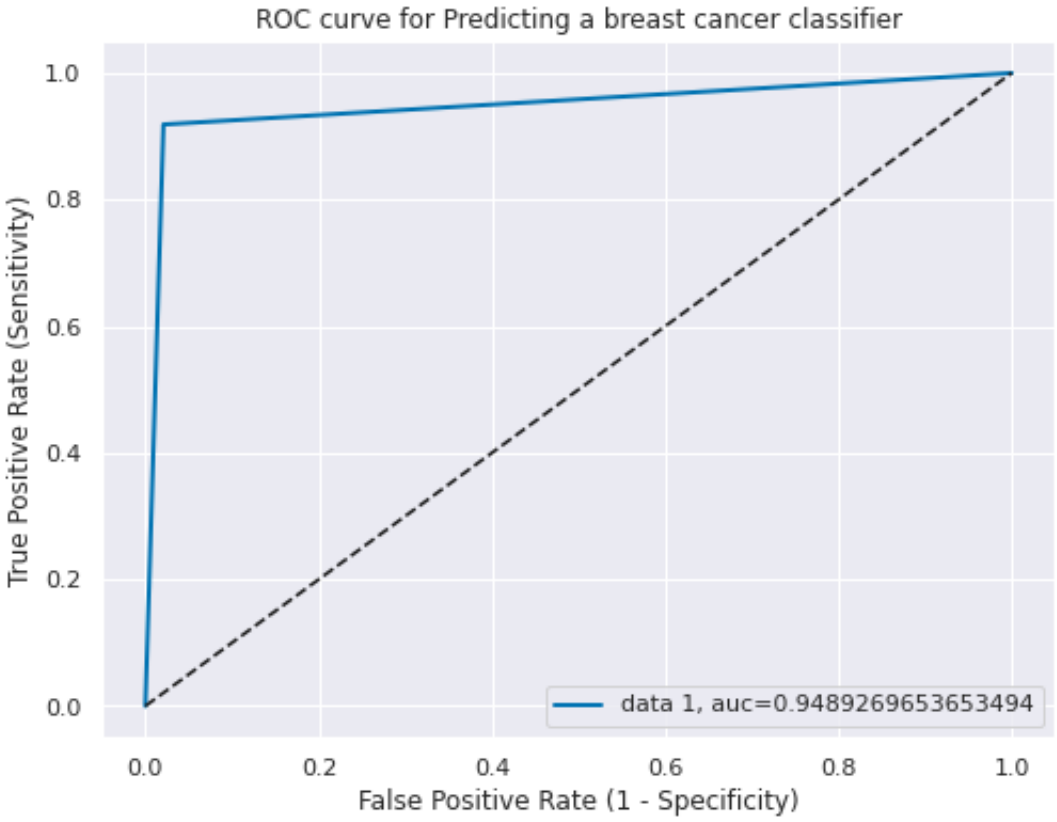
In [90]:
```
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test
_size=0.3, random_state=0)

# SMOTE Sampling with Logistic Regression
logreg = LogisticRegression(max_iter=1000)
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
```

In [91]:
```
print('Accuracy :{0:0.5f}'.format(metrics.accuracy_score(y_pred , y_te
st)))
print('AUC : {0:0.5f}'.format(metrics.roc_auc_score(y_test , y_pred)))
print('Precision : {0:0.5f}'.format(metrics.precision_score(y_test , y
_pred)))
print('Recall : {0:0.5f}'.format(metrics.recall_score(y_test , y_pred)
))
print('F1 : {0:0.5f}'.format(metrics.f1_score(y_test , y_pred)))
```

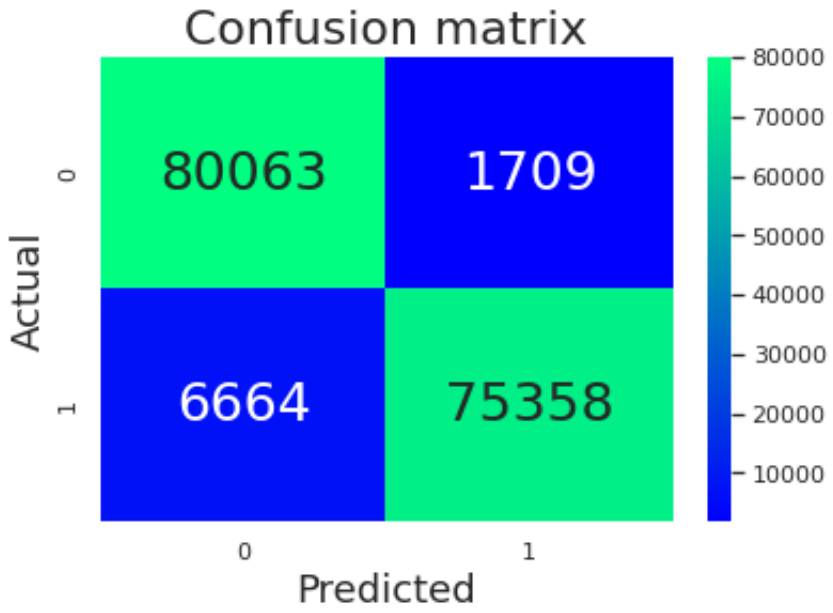
```
Accuracy :0.94888
AUC : 0.94893
Precision : 0.97782
Recall : 0.91875
F1 : 0.94737
```

In [92]:
```
# plot ROC Curve

plt.figure(figsize=(8,6))

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)

auc = metrics.roc_auc_score(y_test, y_pred)
print("AUC - ",auc,"\n")

plt.plot(fpr,tpr,linewidth=2, label="data 1, auc="+str(auc))
plt.legend(loc=4)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12
plt.title('ROC curve for Predicting a breast cancer classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```

```
AUC -  0.9489269653653494
```



In [93]:
```
# Heatmap for Confusion Matrix

cnf_matrix = metrics.confusion_matrix(y_test , y_pred)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, annot_kws={"size": 25}, cmap="winter" ,fmt='g')

plt.title('Confusion matrix', y=1.1, fontsize = 22)
plt.xlabel('Predicted',fontsize = 18)
plt.ylabel('Actual',fontsize = 18)

# ax.xaxis.set_ticklabels(['Genuine', 'Fraud']);
# ax.yaxis.set_ticklabels(['Genuine', 'Fraud']);

plt.show()
```



1.5 Logistic Regression with __ADASYN__ data

In [94]:
```
print('Original dataset shape %s' % Counter(y))

adasyn = ADASYN(random_state=42)

X_res, y_res = adasyn.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_res))
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({1: 273021, 0: 272990})
```

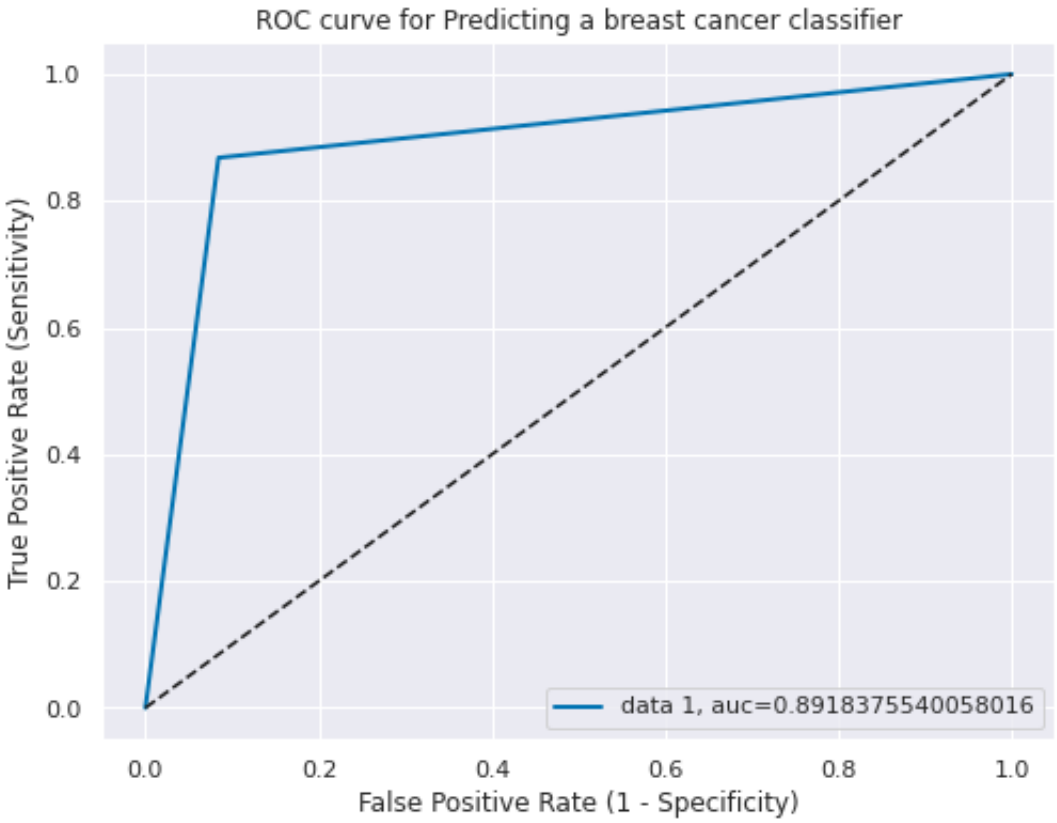
In [95]:
```
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test
_size=0.3, random_state=0)

#  ADASYN Sampling with Logistic Regression
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
```

In [96]:
```
print('Accuracy :{0:0.5f}'.format(metrics.accuracy_score(y_pred , y_te
st)))
print('AUC : {0:0.5f}'.format(metrics.roc_auc_score(y_test , y_pred)))
print('Precision : {0:0.5f}'.format(metrics.precision_score(y_test , y
_pred)))
print('Recall : {0:0.5f}'.format(metrics.recall_score(y_test , y_pred)
))
print('F1 : {0:0.5f}'.format(metrics.f1_score(y_test , y_pred)))
```

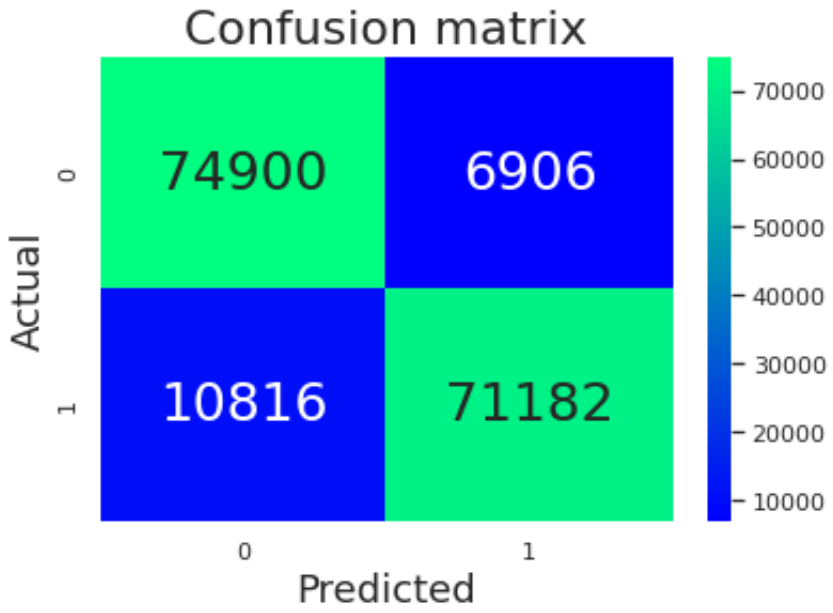
```
Accuracy :0.89181
AUC : 0.89184
Precision : 0.91156
Recall : 0.86809
F1 : 0.88930
```

In [97]:
```
# plot ROC Curve

plt.figure(figsize=(8,6))

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)

auc = metrics.roc_auc_score(y_test, y_pred)
print("AUC - ",auc,"\n")

plt.plot(fpr,tpr,linewidth=2, label="data 1, auc="+str(auc))
plt.legend(loc=4)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12
plt.title('ROC curve for Predicting a breast cancer classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```

```
AUC -  0.8918375540058016
```



In [98]:
```
# Heatmap for Confusion Matrix

cnf_matrix = metrics.confusion_matrix(y_test , y_pred)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, annot_kws={"size": 25}, cmap="winter" ,fmt='g')

plt.title('Confusion matrix', y=1.1, fontsize = 22)
plt.xlabel('Predicted',fontsize = 18)
plt.ylabel('Actual',fontsize = 18)

# ax.xaxis.set_ticklabels(['Genuine', 'Fraud']);
# ax.yaxis.set_ticklabels(['Genuine', 'Fraud']);

plt.show()
```



We have seen that imbalance dataset have Recall score of only 61.77%. It means that creating a model from the imbalanced dataset is highly biased towards genuine transactions and creates a model which is not able to predict the fraudulent transactions correctly. However, the balanced dataset has Recall score of above 87%.

__Spatial nature of class imbalance__

I will reduce 29 columns to 2 columns with the help of **Principal Component Analysis** so that I can look at them in a plot! (because to plot graph we need two dimensions)

In [99]:
```
from sklearn.decomposition import PCA
```

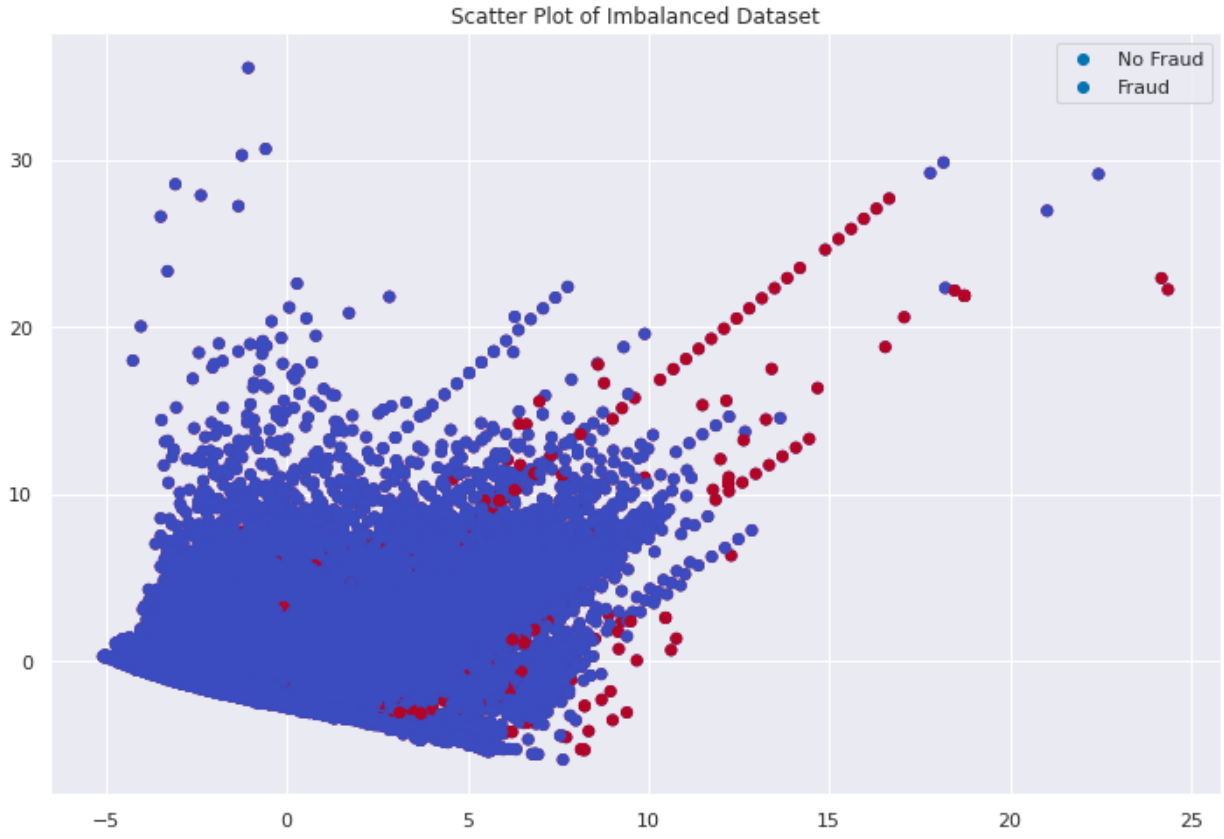
Distribution of balaced dataset

Finally, we can create a scatter plot of the dataset and colour the examples for each class a different colour to clearly see the spatial nature of the class imbalance.

A scatter plot of the dataset is created showing the large mass of points that belong to the minority class (red) and a small number of points spread out for the minority class (blue). We can see some measure of overlap between the two classes.

In [100]:
```
X_reduced_pca_im = PCA(n_components=2, random_state=42).fit_transform(
X)
```

In [101]:
```
# Generate and plot a synthetic imbalanced classification dataset
plt.figure(figsize=(12,8))

plt.scatter(X_reduced_pca_im[:,0], X_reduced_pca_im[:,1], c=(y == 0),
label='No Fraud', cmap='coolwarm', linewidths=1)
plt.scatter(X_reduced_pca_im[:,0], X_reduced_pca_im[:,1], c=(y == 1),
label='Fraud', cmap='coolwarm', linewidths=1)

plt.title("Scatter Plot of Imbalanced Dataset")
plt.legend()
plt.show()
```



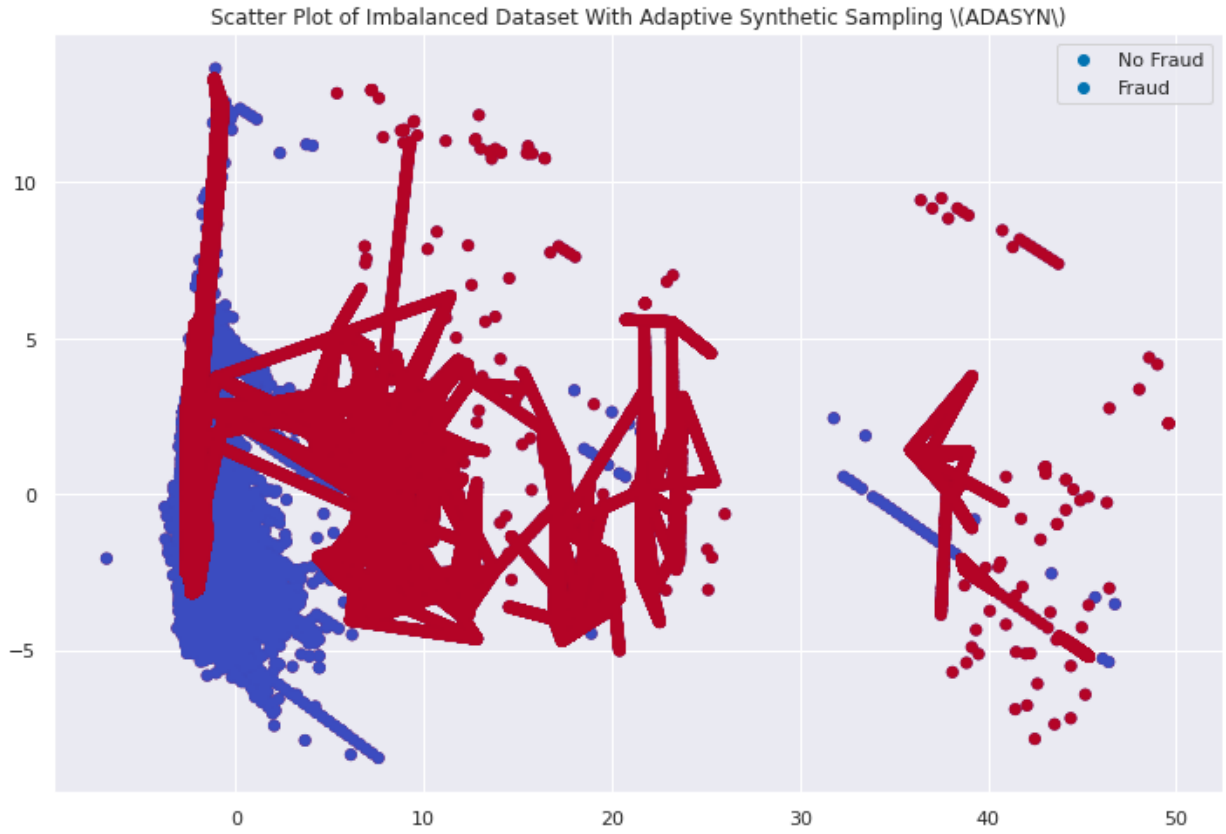
Distribution of balaced dataset

Finally, a scatter plot of the transformed dataset is created.

It shows many more examples in the minority class created along the lines between the original examples in the minority class.

In [102]:
```
X_reduced_pca = PCA(n_components=2, random_state=42).fit_transform(X_r
es)
```

In [103]:
```
# Oversample and plot imbalanced dataset with ADASYN
plt.figure(figsize=(12,8))

plt.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y_res == 0), cm
ap='coolwarm', label='No Fraud', linewidths=1)
plt.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y_res == 1), cm
ap='coolwarm', label='Fraud', linewidths=1)

plt.title("Scatter Plot of Imbalanced Dataset With Adaptive Synthetic
Sampling \(ADASYN\)")
plt.legend()
plt.show()
```



Building different models with different balanced datasets

Let's now try different models , first by creating multiple datasets for undersampled , oversampled and SMOTE sampled

1. Undersampled Data

In [104]:
```
print('Original dataset shape %s' % Counter(y))

rus = RandomUnderSampler(random_state=42)
X_under, y_under = rus.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_under))

# Slit into train and test datasets
X_train_under, X_test_under, y_train_under, y_test_under = train_test_
split(X_under, y_under, test_size=0.3, random_state=0)
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 451, 1: 451})
```

2. Oversampled Data

In [105]:
```
print('Original dataset shape %s' % Counter(y))

ros = RandomOverSampler(random_state=42)
X_over, y_over = ros.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_over))

# Slit into train and test datasets
X_train_over, X_test_over, y_train_over, y_test_over = train_test_spli
t(X_over, y_over, test_size=0.3, random_state=0)
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 272990, 1: 272990})
```

3. SMOTE Data

In [106]:
```
print('Original dataset shape %s' % Counter(y))

smote = SMOTE(random_state=42)
X_smote, y_smote = smote.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_smote))

# Slit into train and test datasets
X_train_smote, X_test_smote, y_train_smote, y_test_smote = train_test_
split(X_smote, y_smote, test_size=0.3, random_state=0)
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({0: 272990, 1: 272990})
```

4. ADASYN Data

In [107]:
```
print('Original dataset shape %s' % Counter(y))

adasyn = ADASYN(random_state=42)
X_adasyn, y_adasyn = adasyn.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_adasyn))

# Slit into train and test datasets
X_train_adasyn, X_test_adasyn, y_train_adasyn, y_test_adasyn = train_t
est_split(X_adasyn, y_adasyn, test_size=0.3, random_state=0)
```

```
Original dataset shape Counter({0: 272990, 1: 451})
Resampled dataset shape Counter({1: 273021, 0: 272990})
```

In [108]:
```
# from sklearn.model_selection import cross_val_score
# from sklearn.model_selection import StratifiedKFold
# from sklearn.linear_model import LogisticRegression
# from sklearn.tree import DecisionTreeClassifier
# # from sklearn.ensemble import RandomForestClassifier
# from sklearn.svm import SVC
# from sklearn.neighbors import KNeighborsClassifier
# from sklearn.naive_bayes import GaussianNB

# # Build Models
# # Let's test 5 different algorithms:

# # Spot Check Algorithms
# models = []

# #------------------ Logistic Regression (LR) ------------------#
# models.append(('LR imbalance', LogisticRegression(solver='liblinear'
, multi_class='ovr'),X,y))
```

```
# models.append(('LR Undersampling', LogisticRegression(solver='liblin
ear', multi_class='ovr'),X_under,y_under))
# models.append(('LR Oversampling', LogisticRegression(solver='libline
ar', multi_class='ovr'),X_over,y_over))
# models.append(('LR SMOTE', LogisticRegression(solver='liblinear', mu
lti_class='ovr'),X_smote,y_smote))
# # models.append(('LR ADASYN', LogisticRegression(solver='liblinear',
multi_class='ovr'),X_adasyn,y_adasyn))

# #-----------------Decision Tree (DT)------------------#
# models.append(('DT imbalance', DecisionTreeClassifier(),X,y))
# models.append(('DT Undersampling', DecisionTreeClassifier(),X_under,
y_under))
# models.append(('DT Oversampling', DecisionTreeClassifier(),X_over,y_
over))
# models.append(('DT SMOTE', DecisionTreeClassifier(),X_smote,y_smote)
)
# # models.append(('DT ADASYN', DecisionTreeClassifier(),X_adasyn,y_ad
asyn))

# #------------------ K-Nearest Neighbors (KNN) ------------------#
# models.append(('KNN imbalance', KNeighborsClassifier(),X,y))
# models.append(('KNN Undersampling', KNeighborsClassifier(),X_under,y
_under))
# models.append(('KNN Oversampling', KNeighborsClassifier(),X_over,y_o
ver))
# models.append(('KNN SMOTE', KNeighborsClassifier(),X_smote,y_smote))
# # models.append(('DT ADASYN', KNeighborsClassifier(),X_adasyn,y_adas
yn))

# #------------------ Support Vector Machines (SVM) ------------------
#
# # models.append(('SVM imbalance', SVC(gamma='auto'),X,y))
# # models.append(('SVM Undersampling', SVC(gamma='auto'),X_under,y_un
der))
# # models.append(('SVM Oversampling', SVC(gamma='auto'),X_over,y_over
))
# # models.append(('SVM SMOTE', SVC(gamma='auto'),X_smote,y_smote))
# # # models.append(('SVM ADASYN', SVC(gamma='auto'),X_adasyn,y_adasyn
))

# #------------------ Gaussian Naive Bayes (NB) ------------------#
# models.append(('NB imbalance', GaussianNB(),X,y))
# models.append(('NB Undersampling', GaussianNB(),X_under,y_under))
# models.append(('NB Oversampling', GaussianNB(),X_over,y_over))
# models.append(('NB SMOTE', GaussianNB(),X_smote,y_smote))
# # models.append(('NB ADASYN', GaussianNB(),X_adasyn,y_adasyn))

# # evaluate each model in turn
# names_lst = []
```

```
# aucs_lst = []
# accuracy_lst = []
# precision_lst = []
# recall_lst = []
# f1_lst = []

# plt.figure(figsize=(14,8))

# for name, model,Xdata,ydata in models:

#     names_lst.append(name)

#     # split data in train test set
#     X_train, X_test, y_train, y_test = train_test_split(Xdata, ydata
, test_size=0.3, random_state=0)
#     # Build model
#     model.fit(X_train, y_train)
#     # Predict
#     y_pred = model.predict(X_test)

#     # calculate accuracy
#     Accuracy = metrics.accuracy_score(y_pred , y_test)
#     accuracy_lst.append(Accuracy)

#     # calculate auc
#     Aucs = metrics.roc_auc_score(y_test , y_pred)
#     aucs_lst.append(Aucs)

#     # calculate precision
#     PrecisionScore = metrics.precision_score(y_test , y_pred)
#     precision_lst.append(PrecisionScore)

#     # calculate recall
#     RecallScore = metrics.recall_score(y_test , y_pred)
#     recall_lst.append(RecallScore)

#     # calculate f1 score
#     F1Score = metrics.f1_score(y_test , y_pred)
#     f1_lst.append(F1Score)

#     print('F1 Score of '+ name +' model : {0:0.5f}'.format(F1Score))

# #     draw confusion matrix
# #     cnf_matrix = metrics.confusion_matrix(y_test , y_pred)

# #     print("Model Name :", name)
# #     print('Accuracy :{0:0.5f}'.format(Accuracy))
# #     print('AUC : {0:0.5f}'.format(Aucs))
# #     print('Precision : {0:0.5f}'.format(PrecisionScore))
# #     print('Recall : {0:0.5f}'.format(RecallScore))
```

```
# #     print('F1 : {0:0.5f}'.format(F1Score))
# #     print('Confusion Matrix : \n', cnf_matrix)
# #     print("\n")


#     # plot ROC Curve
#     fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)
#     auc = metrics.roc_auc_score(y_test, y_pred)
#     plt.plot(fpr,tpr,linewidth=2, label=name + ", auc="+str(auc))
#     #---------- For loops ends here--------#


# plt.legend(loc=4)
# plt.plot([0,1], [0,1], 'k--' )
# plt.rcParams['font.size'] = 12
# plt.title('ROC curve for Predicting a credit card fraud detection')
# plt.xlabel('False Positive Rate (1 - Specificity)')
# plt.ylabel('True Positive Rate (Sensitivity)')
# plt.show()

# data = {'Model':names_lst,
#         'Accuracy':accuracy_lst,
#         'AUC':aucs_lst,
#         'PrecisionScore':precision_lst,
#         'RecallScore':recall_lst,
#         'F1Score':f1_lst}

# print("Performance measures of various classifiers: \n")
# performance_df = pd.DataFrame(data)
# performance_df.sort_values(['AUC','RecallScore','F1Score','Precision
Score'],ascending=False)
```

In [109]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
```

In [110]:
```
names_lst = []
aucs_lst = []
accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
```

```
# Function for model building and performance measure

def build_measure_model(models):
    plt.figure(figsize=(12,6))

    for name, model,Xdata,ydata in models:

        names_lst.append(name)

        # split data in train test set
        X_train, X_test, y_train, y_test = train_test_split(Xdata, yda
ta, test_size=0.3, random_state=0)
        # Build model
        model.fit(X_train, y_train)
        # Predict
        y_pred = model.predict(X_test)

        # calculate accuracy
        Accuracy = metrics.accuracy_score(y_pred , y_test)
        accuracy_lst.append(Accuracy)

        # calculate auc
        Aucs = metrics.roc_auc_score(y_test , y_pred)
        aucs_lst.append(Aucs)

        # calculate precision
        PrecisionScore = metrics.precision_score(y_test , y_pred)
        precision_lst.append(PrecisionScore)

        # calculate recall
        RecallScore = metrics.recall_score(y_test , y_pred)
        recall_lst.append(RecallScore)

        # calculate f1 score
        F1Score = metrics.f1_score(y_test , y_pred)
        f1_lst.append(F1Score)

        #print('F1 Score of '+ name +' model : {0:0.5f}'.format(F1Scor
e))

        # draw confusion matrix
        cnf_matrix = metrics.confusion_matrix(y_test , y_pred)

        print("Model Name :", name)
        print('Accuracy :{0:0.5f}'.format(Accuracy))
        print('AUC : {0:0.5f}'.format(Aucs))
        print('Precision : {0:0.5f}'.format(PrecisionScore))
        print('Recall : {0:0.5f}'.format(RecallScore))
        print('F1 : {0:0.5f}'.format(F1Score))
        print('Confusion Matrix : \n', cnf_matrix)
```

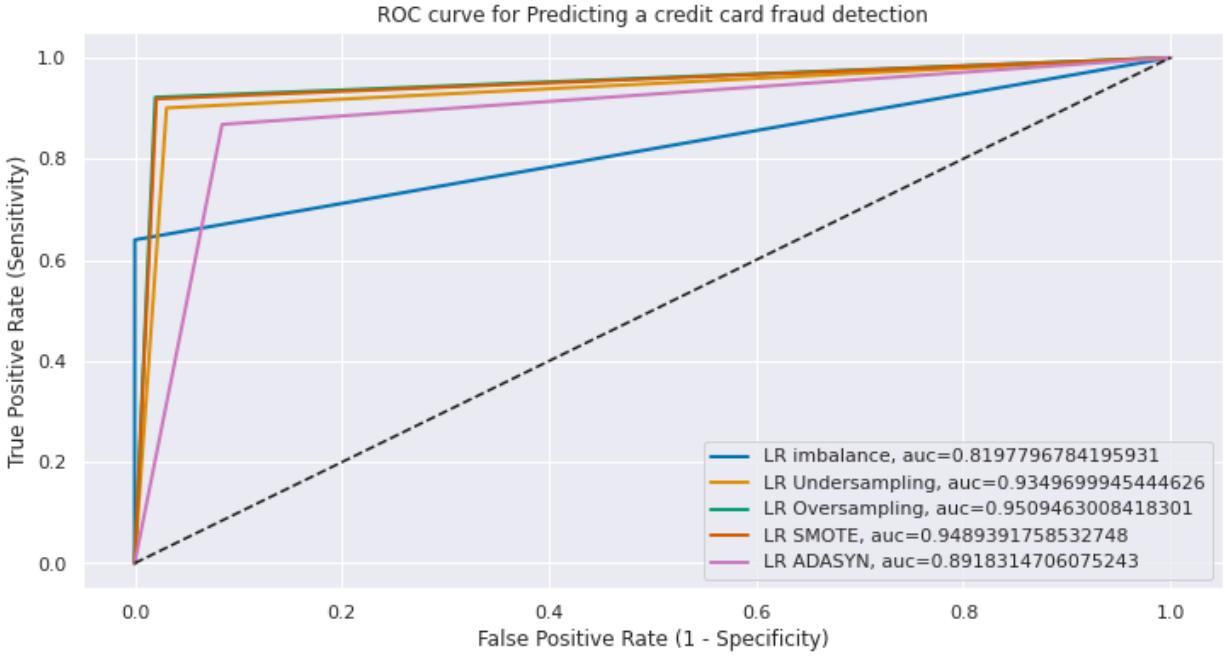
```
        print("\n")


        # plot ROC Curve
        fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred)
        auc = metrics.roc_auc_score(y_test, y_pred)
        plt.plot(fpr,tpr,linewidth=2, label=name + ", auc="+str(auc))

        #---------- For loops ends here--------#


    plt.legend(loc=4)
    plt.plot([0,1], [0,1], 'k--' )
    plt.rcParams['font.size'] = 12
    plt.title('ROC curve for Predicting a credit card fraud detection'
)
    plt.xlabel('False Positive Rate (1 - Specificity)')
    plt.ylabel('True Positive Rate (Sensitivity)')
    plt.show()
```

Logistic Regression (LR)

In [111]:
```
#------------------ Logistic Regression (LR) ------------------#
LRmodels = []

LRmodels.append(('LR imbalance', LogisticRegression(solver='liblinear'
, multi_class='ovr'),X,y))
LRmodels.append(('LR Undersampling', LogisticRegression(solver='liblin
ear', multi_class='ovr'),X_under,y_under))
LRmodels.append(('LR Oversampling', LogisticRegression(solver='libline
ar', multi_class='ovr'),X_over,y_over))
LRmodels.append(('LR SMOTE', LogisticRegression(solver='liblinear', mu
lti_class='ovr'),X_smote,y_smote))
LRmodels.append(('LR ADASYN', LogisticRegression(solver='liblinear', m
ulti_class='ovr'),X_adasyn,y_adasyn))

# Call function to create model and measure its performance
build_measure_model(LRmodels)
```

```
Model Name : LR imbalance
Accuracy :0.99926
AUC : 0.81978
Precision : 0.87879
Recall : 0.63971
F1 : 0.74043
Confusion Matrix :
 [[81885    12]
 [   49    87]]
```

```
Model Name : LR Undersampling
Accuracy :0.93358
AUC : 0.93497
Precision : 0.96947
Recall : 0.90071
F1 : 0.93382
Confusion Matrix :
 [[126   4]
 [ 14 127]]


Model Name : LR Oversampling
Accuracy :0.95090
AUC : 0.95095
Precision : 0.97897
Recall : 0.92175
F1 : 0.94950
Confusion Matrix :
 [[80148  1624]
 [ 6418 75604]]


Model Name : LR SMOTE
Accuracy :0.94889
AUC : 0.94894
Precision : 0.97784
Recall : 0.91877
F1 : 0.94738
Confusion Matrix :
 [[80064  1708]
 [ 6663 75359]]


Model Name : LR ADASYN
Accuracy :0.89180
AUC : 0.89183
Precision : 0.91157
Recall : 0.86807
F1 : 0.88929
Confusion Matrix :
 [[74901  6905]
 [10818 71180]]
```

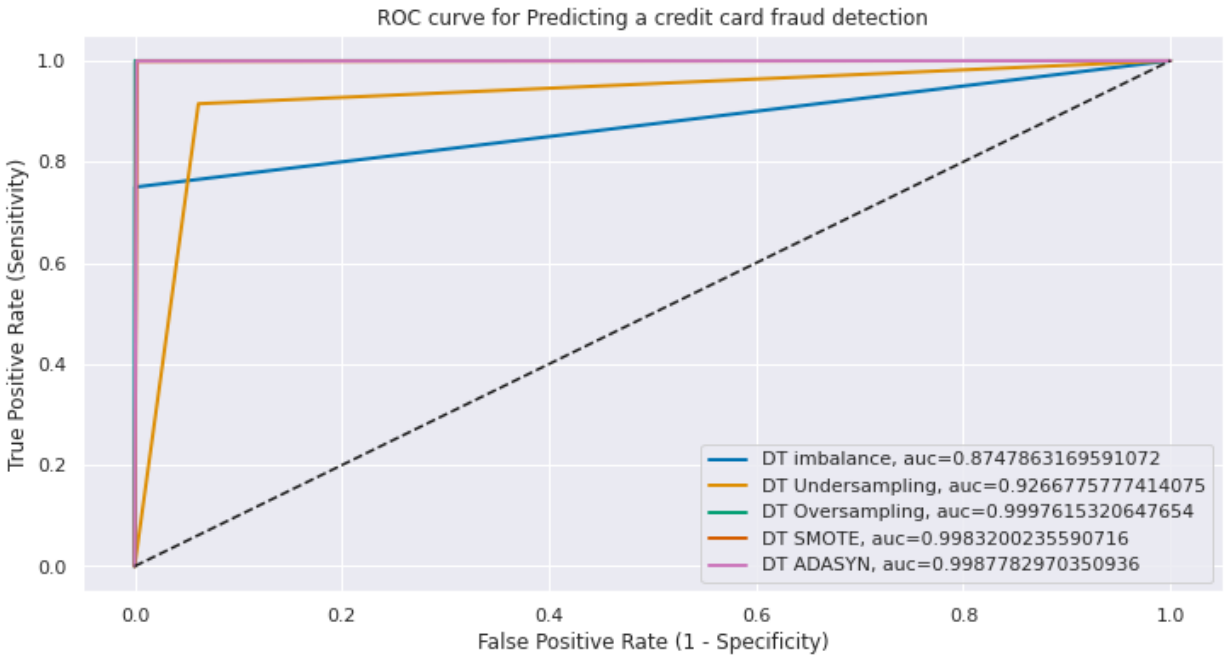


Decision Tree (DT)

In [112]:
```
#-----------------Decision Tree (DT)-----------------#
DTmodels = []

dt = DecisionTreeClassifier()

DTmodels.append(('DT imbalance', dt,X,y))
DTmodels.append(('DT Undersampling', dt,X_under,y_under))
DTmodels.append(('DT Oversampling', dt,X_over,y_over))
DTmodels.append(('DT SMOTE', dt,X_smote,y_smote))
DTmodels.append(('DT ADASYN', dt,X_adasyn,y_adasyn))

# Call function to create model and measure its performance
build_measure_model(DTmodels)
```

```
Model Name : DT imbalance
Accuracy :0.99916
AUC : 0.87479
Precision : 0.74453
Recall : 0.75000
F1 : 0.74725
Confusion Matrix :
 [[81862    35]
 [   34   102]]


Model Name : DT Undersampling
```

```
Accuracy :0.92620
AUC : 0.92668
Precision : 0.94161
Recall : 0.91489
F1 : 0.92806
Confusion Matrix :
 [[122   8]
 [ 12 129]]


Model Name : DT Oversampling
Accuracy :0.99976
AUC : 0.99976
Precision : 0.99952
Recall : 1.00000
F1 : 0.99976
Confusion Matrix :
 [[81733    39]
 [    0 82022]]


Model Name : DT SMOTE
Accuracy :0.99832
AUC : 0.99832
Precision : 0.99765
Recall : 0.99900
F1 : 0.99832
Confusion Matrix :
 [[81579   193]
 [   82 81940]]


Model Name : DT ADASYN
Accuracy :0.99878
AUC : 0.99878
Precision : 0.99816
Recall : 0.99940
F1 : 0.99878
Confusion Matrix :
 [[81655   151]
 [   49 81949]]
```

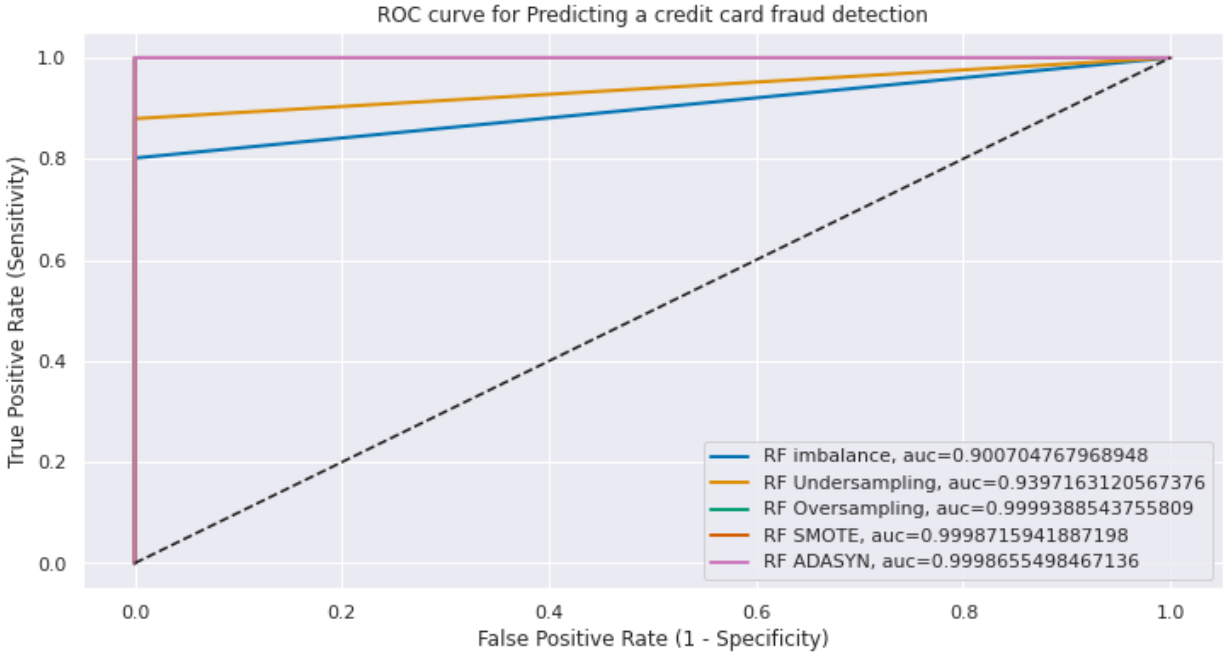


Random Forest (RF)

In [113]:
```
#-----------------Random Forest (RF) -----------------#
RFmodels = []

RFmodels.append(('RF imbalance', RandomForestClassifier(),X,y))
RFmodels.append(('RF Undersampling', RandomForestClassifier(),X_under,
y_under))
RFmodels.append(('RF Oversampling', RandomForestClassifier(),X_over,y_
over))
RFmodels.append(('RF SMOTE', RandomForestClassifier(),X_smote,y_smote)
)
RFmodels.append(('RF ADASYN', RandomForestClassifier(),X_adasyn,y_adas
yn))

# Call function to create model and measure its performance
build_measure_model(RFmodels)
```

```
Model Name : RF imbalance
Accuracy :0.99961
AUC : 0.90070
Precision : 0.95614
Recall : 0.80147
F1 : 0.87200
Confusion Matrix :
 [[81892     5]
 [   27   109]]
```

```
Model Name : RF Undersampling
Accuracy :0.93727
AUC : 0.93972
Precision : 1.00000
Recall : 0.87943
F1 : 0.93585
Confusion Matrix :
 [[130   0]
 [ 17 124]]


Model Name : RF Oversampling
Accuracy :0.99994
AUC : 0.99994
Precision : 0.99988
Recall : 1.00000
F1 : 0.99994
Confusion Matrix :
 [[81762    10]
 [    0 82022]]


Model Name : RF SMOTE
Accuracy :0.99987
AUC : 0.99987
Precision : 0.99974
Recall : 1.00000
F1 : 0.99987
Confusion Matrix :
 [[81751    21]
 [    0 82022]]


Model Name : RF ADASYN
Accuracy :0.99987
AUC : 0.99987
Precision : 0.99974
Recall : 0.99999
F1 : 0.99987
Confusion Matrix :
 [[81785    21]
 [    1 81997]]
```



In [114]:
```
# #------------------ K-Nearest Neighbors (KNN) ------------------#
# KNNmodels = []

# KNNmodels.append(('KNN imbalance', KNeighborsClassifier(),X,y))
# KNNmodels.append(('KNN Undersampling', KNeighborsClassifier(),X_under,y_under))
# KNNmodels.append(('KNN Oversampling', KNeighborsClassifier(),X_over,y_over))
# KNNmodels.append(('KNN SMOTE', KNeighborsClassifier(),X_smote,y_smote))
# KNNmodels.append(('KNN ADASYN', KNeighborsClassifier(),X_adasyn,y_adasyn))

# Call function to create model and measure its performance
# build_measure_model(KNNmodels)
```

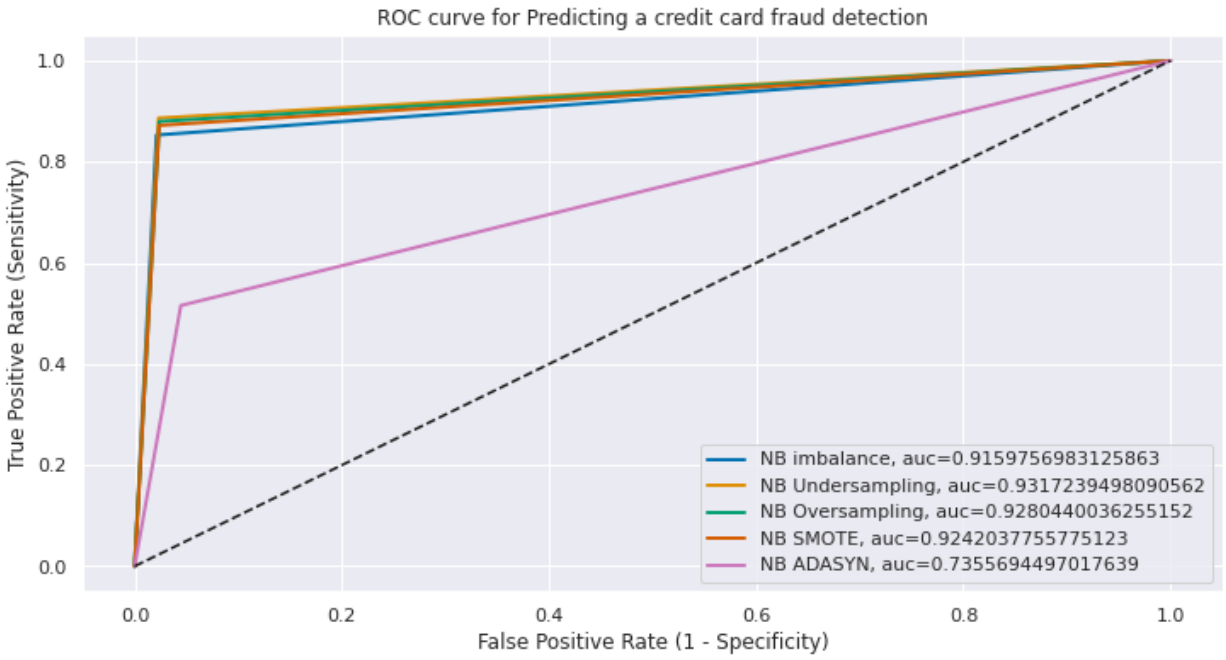
In [115]:
```
# #------------------ Support Vector Machines (SVM) ------------------
#
# SVMmodels = []

# SVMmodels.append(('SVM imbalance', SVC(gamma='auto'),X,y))
# SVMmodels.append(('SVM Undersampling', SVC(gamma='auto'),X_under,y_u
nder))
# SVMmodels.append(('SVM Oversampling', SVC(gamma='auto'),X_over,y_ove
r))
# SVMmodels.append(('SVM SMOTE', SVC(gamma='auto'),X_smote,y_smote))
# SVMmodels.append(('SVM ADASYN', SVC(gamma='auto'),X_adasyn,y_adasyn)
)

# Call function to create model and measure its performance
# build_measure_model(SVMmodels)
```

Naive Bayes (NB)

In [116]:
```
#------------------ Gaussian Naive Bayes (NB) ------------------#
NBmodels = []

NBmodels.append(('NB imbalance', GaussianNB(),X,y))
NBmodels.append(('NB Undersampling', GaussianNB(),X_under,y_under))
NBmodels.append(('NB Oversampling', GaussianNB(),X_over,y_over))
NBmodels.append(('NB SMOTE', GaussianNB(),X_smote,y_smote))
NBmodels.append(('NB ADASYN', GaussianNB(),X_adasyn,y_adasyn))

# Call function to create model and measure its performance
build_measure_model(NBmodels)
```

```
Model Name : NB imbalance
Accuracy :0.97880
AUC : 0.91598
Precision : 0.06322
Recall : 0.85294
F1 : 0.11771
Confusion Matrix :
 [[80178  1719]
 [   20   116]]


Model Name : NB Undersampling
Accuracy :0.92989
AUC : 0.93172
Precision : 0.97656
Recall : 0.88652
F1 : 0.92937
```

```
Confusion Matrix :
 [[127   3]
 [ 16 125]]


Model Name : NB Oversampling
Accuracy :0.92797
AUC : 0.92804
Precision : 0.97374
Recall : 0.87989
F1 : 0.92444
Confusion Matrix :
 [[79826  1946]
 [ 9852 72170]]


Model Name : NB SMOTE
Accuracy :0.92412
AUC : 0.92420
Precision : 0.97366
Recall : 0.87207
F1 : 0.92007
Confusion Matrix :
 [[79837  1935]
 [10493 71529]]


Model Name : NB ADASYN
Accuracy :0.73531
AUC : 0.73557
Precision : 0.92066
Recall : 0.51568
F1 : 0.66108
Confusion Matrix :
 [[78162  3644]
 [39713 42285]]
```



Performance measures of various classifiers

In [117]:
```
data = {'Model':names_lst,
        'Accuracy':accuracy_lst,
        'AUC':aucs_lst,
        'PrecisionScore':precision_lst,
        'RecallScore':recall_lst,
        'F1Score':f1_lst}

print("Performance measures of various classifiers: \n")
performance_df = pd.DataFrame(data)
performance_df.sort_values(['AUC','RecallScore','F1Score'],ascending=F
alse)
```

Performance measures of various classifiers:

Out[117]:

	Model	Accuracy	AUC	PrecisionScore	RecallScore	F1Score
12	RF Oversampling	0.999939	0.999939	0.999878	1.000000	0.999939
13	RF SMOTE	0.999872	0.999872	0.999744	1.000000	0.999872
14	RF ADASYN	0.999866	0.999866	0.999744	0.999988	0.999866
7	DT Oversampling	0.999762	0.999762	0.999525	1.000000	0.999762
9	DT ADASYN	0.998779	0.998778	0.998161	0.999402	0.998781
8	DT SMOTE	0.998321	0.998320	0.997650	0.999000	0.998325
2	LR Oversampling	0.950902	0.950946	0.978971	0.921753	0.949501
3	LR SMOTE	0.948893	0.948939	0.977837	0.918766	0.947382
11	RF Undersampling	0.937269	0.939716	1.000000	0.879433	0.935849
1	LR Undersampling	0.933579	0.934970	0.969466	0.900709	0.933824
16	NB Undersampling	0.929889	0.931724	0.976562	0.886525	0.929368
17	NB Oversampling	0.927970	0.928044	0.973744	0.879886	0.924439
6	DT Undersampling	0.926199	0.926678	0.941606	0.914894	0.928058
18	NB SMOTE	0.924124	0.924204	0.973661	0.872071	0.920070
15	NB imbalance	0.978801	0.915976	0.063215	0.852941	0.117707
10	RF imbalance	0.999610	0.900705	0.956140	0.801471	0.872000
4	LR ADASYN	0.891804	0.891831	0.911571	0.868070	0.889289
5	DT imbalance	0.999159	0.874786	0.744526	0.750000	0.747253
0	LR imbalance	0.999256	0.819780	0.878788	0.639706	0.740426
19	NB ADASYN	0.735312	0.735569	0.920660	0.515683	0.661080

Hightlights

After training each of the models, these are the final results. All of the scores for Random Forest with Oversampling technique and the Random Forest with SMOTE technique models are very promising for our dataset! Each model has a high true positive rate and a low false-positive rate, which is exactly what we're looking for.

In the ROC graph above, the AUC scores for Random Forest with Oversampling technique is pretty high, which is what we'd like to see. As we move further right along the curve, we both capture more True Positives but also incur more False Positives. This means we capture more fraudulent transactions, but also flag even more normal transactions as fraudulent. **So Random Forest with Oversampling technique is our final model, as this gives highest Recall score.**

__Grid Search__

Grid search is the process of performing hyper parameter tuning in order to determine the optimal values for a given model. This is significant as the performance of the entire model is based on the hyper parameter values specified.

A **model hyperparameter** is a characteristic of a model that is external to the model and whose value cannot be estimated from data. The value of the hyperparameter has to be set before the learning process begins. For example, c in Support Vector Machines, k in k-Nearest Neighbors, the number of hidden layers in Neural Networks.

In contrast, a **parameter** is an internal characteristic of the model and its value can be estimated from data. Example, beta coefficients of linear/logistic regression or support vectors in Support Vector Machines.

Ref:

- https://medium.com/datadriveninvestor/an-introduction-to-grid-search-ff57adcc0998 (https://medium.com/datadriveninvestor/an-introduction-to-grid-search-ff57adcc0998)
- https://towardsdatascience.com/grid-search-for-hyperparameter-tuning-9f63945e8fec (https://towardsdatascience.com/grid-search-for-hyperparameter-tuning-9f63945e8fec)
- https://www.youtube.com/watch?v=GoI_qOgRqfA (https://www.youtube.com/watch?v=GoI_qOgRqfA)

Youtube

In [118]:
```
YouTubeVideo('Gol_qOgRqfA', width=800, height=400)
```

Out[118]:

In [119]:
```
# Use GridSearchCV to find the best parameters.
from sklearn.model_selection import GridSearchCV
```

1. Grid Search with Logistic Regression

In [120]:
```
#------------ Logistic Regression ------------#
log_reg_params = {"solver": ['saga'],
                  "penalty": ['l1', 'l2'],
                  'C':  [0.01, 0.1, 1, 10, 100],
                  "max_iter" : [100000]},

grid_log_reg = GridSearchCV(LogisticRegression(), log_reg_params)
grid_log_reg.fit(X_train_under,y_train_under)

# Logistic Regression best estimator
print("Logistic Regression best estimator : \n",grid_log_reg.best_esti
mator_)

# predict test dataset
y_pred_lr = grid_log_reg.predict(X_test_under)

# f1 score
print('\nLogistic Regression f1 Score : {0:0.5f}'.format(metrics.f1_sc
ore(y_test_under , y_pred_lr)))
```

```
Logistic Regression best estimator :
 LogisticRegression(C=1, max_iter=100000, solver='saga')

Logistic Regression f1 Score : 0.93040
```

2. Grid Search with K Nearest Neighbour Classifier

In [121]:
```
#------------ K Nearest Neighbour ------------#
knears_params = {"n_neighbors": list(range(2,60,1)),
                 'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'
]}

grid_knears = GridSearchCV(KNeighborsClassifier(), knears_params)

grid_knears.fit(X_train_under,y_train_under)

# KNears best estimator
print("KNN best estimator : \n",grid_knears.best_estimator_)

# predict test dataset
y_pred_knn = grid_knears.predict(X_test_under)

# f1 score
print('\nKNN f1 Score : {0:0.5f}'.format(metrics.f1_score(y_test_under
, y_pred_knn)))
```

```
KNN best estimator :
 KNeighborsClassifier(n_neighbors=3)

KNN f1 Score : 0.93333
```

3. Grid Search with Support Vector Classifier

In [122]:
```
#------------ Support Vector Classifier ------------#
svc_params = {'C': [0.5, 0.7, 0.9, 1],
              'kernel': ['rbf', 'poly', 'sigmoid', 'linear']}

grid_svc = GridSearchCV(SVC(), svc_params)
grid_svc.fit(X_train_under,y_train_under)

# SVC best estimator
print("SVC best estimator : \n",grid_svc.best_estimator_)

# predict test dataset
y_pred_svc = grid_svc.predict(X_test_under)

# f1 score
print('\nSVC f1 Score : {0:0.5f}'.format(metrics.f1_score(y_test_under
, y_pred_svc)))
```

```
SVC best estimator :
 SVC(C=0.5, kernel='linear')

SVC f1 Score : 0.93727
```

4. Grid Search with Decision Tree Classifier

In [123]:
```
#------------ DecisionTree Classifier ------------#
tree_params = {"criterion": ["gini", "entropy"],
               "max_depth": list(range(2,4,1)),
               "min_samples_leaf": list(range(5,7,1))}

grid_tree = GridSearchCV(estimator = DecisionTreeClassifier(),
                         param_grid = tree_params,
                         scoring = 'accuracy',
                         cv = 5,
                         verbose = 1,
                         n_jobs = -1)


grid_tree.fit(X_train_under,y_train_under)

# tree best estimator
print("Decision Tree best estimator : \n",grid_tree.best_estimator_)

# predict test dataset
y_pred_dt = grid_tree.predict(X_test_under)


# f1 score
print('\nf1 Score : {0:0.5f}'.format(metrics.f1_score(y_test_under , y
_pred_dt)))
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent w
orkers.

Decision Tree best estimator :
 DecisionTreeClassifier(max_depth=2, min_samples_leaf=5)

f1 Score : 0.90714

[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:    1.9s finish
ed
```

__Conclusion__

We were able to accurately identify fraudulent credit card transactions using a random forest model with oversampling technique. We, therefore, chose the random forest model with oversampling technique as the better model, which obtained recall score of 99% on the test set.

I hope I was able to explain my findings well and thanks so much for reading!