Credit Card Fraud Detection

A synopsis submitted for partial fulfillment

for award of the degree of

Master in Computer Application

By

Avinandan Bose

Debasish Ghosh

Under the guidance of

Mrs. Debrupa Paul

Narula Institute of Technology

Date of Submission: 04/12/2021

Narula Institute of Technology

81, Nilgunj Road, Agarpara

Kolkata – 700109

West Bengal

Certificate of Originality

We have developed this project as partial fulfillment of master in computer application 3rd semester (Sub Code: MCA20-301). Numerous persons have numerous research on this project: “Credit Card Fraud Detection”, we are hereby to present an effort to this subject and it is as original to our effort taking help of such research papers and sites.

Name	**Roll no.**
Avinandan Bose	**434120010019**
Debasish Ghosh	**434120010022**

Certificate of Approval

This is to certify that the project entitled **"Credit Card Fraud Detection"** has been carried out by MCA 3rd semester student: **Avinandan Bose**, and **Debasish Ghosh**.

Name	**Roll No.**	**Registration No.**
Avinandan Bose	**434120010019**	
Debasish Ghosh	**434120010022**	

It is understood that by this approval the undersigned do not necessarily endorse any of the statements made or opinion expressed in their project. But we approve it only for the purpose for which it is submitted.

_____

Dept. Of CA
&
Project Guide
Narula Institute of Technology
Agarpara, Kolkata - 700109

Acknowledgement

We would like to express our special thanks of gratitude to our project guide: **“Mrs. Debrupa Paul”,** for her able guidance to complete our project.

The completion of this undertaking project could not have been possible without the participation and assistance of **Debasish Ghosh** and **Avinandan Bose** contributed and finishing the project .

The contributions are sincerely appreciated and gratefully acknowledged.

Lastly we want to thank **NIT -our college,** for giving us such a beautiful opportunity to present this project.

Avinandan Bose
Debasish Ghosh

Abstract

The project is aimed to analyze data of credit card transaction and study it to see probability of fraud and no fraud situation. Credit card fraud detection is presently the most frequently occurring problem in the present world. This is due to the rise in both online transactions and e-commerce platforms. Credit card fraud generally happens when the card was stolen for any of the unauthorized purposes or even when the fraudster uses the credit card information for his use. In the present world, we are facing a lot of credit card problems. To detect the fraudulent activities, the credit card fraud detection system was introduced. This project aims to focus mainly on machine learning algorithms. The project is aim to minimize fraud in credit card transaction.

Index

- ❖ **Synthetic minority Over Sampling**
- ❖ **Data-Reanalysis.**
- ❖ **Modelling**
    - **Machine Learning Algorithms**
    - **Logistic Regression**
    - **Decision Tree**
    - **Support Vector Machine**
- ❖ **Model Training**
    - **Data Preparation**
    - **Metric Trap**
    - **Training and Evaluation**
- ❖ **Generating and Plotting ROC Curve**
- ❖ **Confusion Matrices**
- ❖ **Classification Report**
- ❖ **Data Flow Diagram**
- ❖ **Entity Relationship and Activity Diagram**
- ❖ **USE CASE DIAGRAM**
- ❖ **Further Reading**
- ❖ **Conclusion**
- ❖ **Bibliography**

Introduction

Credit Card Fraud Detection is one of the hottest topics of Machine learning. For simplicity, we compare this same concept with any financial transaction. This project will cover various techniques to handle skewed data, and some relevant machine learning models to address this problem.

This project will give us the skill to handle feature just with their statistical analysis, when domain knowledge information is not available. In this project we will also find ways to handle a skewed dataset and how to draw information out of it.

This skill to understand different types of data and making an ML model is one of the crucial skills for any ML practitioner. It is important to know that data crunching concepts are “The Skills” to have for this role.

Hard ware Requirements:

Processor : Pentium i3 or higher(Better to start with i5).

RAM : 4 GB or higher(Better to start with 8gb ram).

Hard Disk Drive : 20 GB (free).

Peripheral Devices : Monitor, Mouse and Keyboard.

Software Requirements:

Operating System : Windows 8/10

IDE Tool : Jupyter notebook.

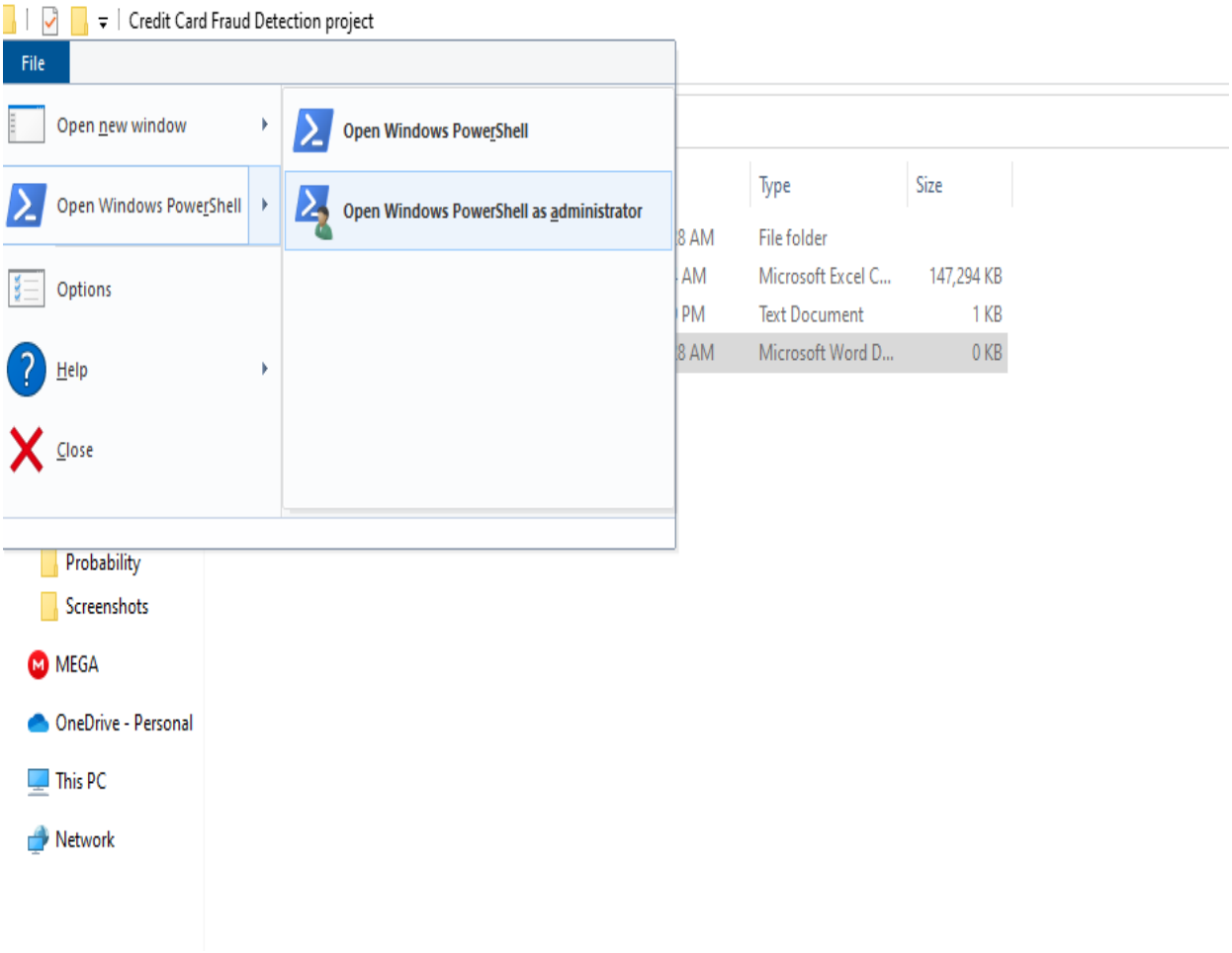
Coding Language : Python 3.8

Tools : Numpy, Matplotlib, Seaborn, Pandas,Scikit-learn etc.

Getting ready with the project:

1. We have to create virtual environment to the folder where we will do the project.

   Code : a. Run PowerShell or CMD as an administrator from the project depository folder.



   b. type : py –m venv virtual_env_name

   And press enter

```
PS C:\Users\Avinandan\Credit Card Fraud Detection project> py -m venv virtual_env_name
PS C:\Users\Avinandan\Credit Card Fraud Detection project>
```

2.Activate the environment :

To do that 1st Set execution policy remotely signed as if execution policy not changed then it will obstruct to run the script.
Then: run the script:

```
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you
 to the security risks described in the about_Execution_Policies help topic at
https:/go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): A
PS C:\Users\Avinandan\Credit Card Fraud Detection project> .\virtual_env_name\scripts\activate
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project>
```

3.Upgrade the pip version

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install --upgrade pip
Requirement already satisfied: pip in c:\users\avinandan\credit card fraud detection project\virtual_env_name\lib\site-pa
kages (21.2.3)
Collecting pip
  Downloading pip-21.3.1-py3-none-any.whl (1.7 MB)
     |                                | 1.7 MB 1.3 MB/s
```

4. Checking the pip version

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip --version
pip 21.3.1 from C:\Users\Avinandan\Credit Card Fraud Detection project\virtual_env_name\lib\site-packages\pip (python 3.10)
```

5. Installing Jupyter Lab

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install jupyterlab
Collecting jupyterlab
  Downloading jupyterlab-3.2.1-py3-none-any.whl (8.6 MB)
     |████████████████████████████████| 8.6 MB 726 kB/s
Collecting ipython
  Downloading ipython-7.28.0-py3-none-any.whl (788 kB)
     |████████████████████████████████| 788 kB 1.1 MB/s
Collecting jupyter-core
  Downloading jupyter_core-4.9.1-py3-none-any.whl (86 kB)
     |████████████████████████████████| 86 kB 519 kB/s
Collecting nbclassic~=0.2
  Downloading nbclassic-0.3.4-py3-none-any.whl (25 kB)
Collecting tornado>=6.1.0
  Downloading tornado-6.1.tar.gz (497 kB)
     |████████████████████████████████| 497 kB 656 kB/s
  Preparing metadata (setup.py) ... done
```

6.Run Jupyter

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> jupyter-lab
```

7. Install Numpy and Pandas.

Note: Just installing pandas will install numpy too in the project folder.

```
Administrator: Windows PowerShell
PS C:\Users\Avinandan\Credit Card Fraud Detection project> .\virtual_env_name\scripts\activate
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install pandas
Collecting pandas
  Downloading pandas-1.3.4-cp310-cp310-win_amd64.whl (10.2 MB)
     |████████████████████████████████| 10.2 MB 595 kB/s
Requirement already satisfied: pytz>=2017.3 in c:\users\avinandan\credit card fraud detection proje
s) (2021.3)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\avinandan\credit card fraud detec
from pandas) (2.8.2)
Collecting numpy>=1.21.0
  Downloading numpy-1.21.3-cp310-cp310-win_amd64.whl (14.0 MB)
     |████████████████████████████████| 14.0 MB 726 kB/s
```

8. Install Scikit Learn

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install scikit-learn
Collecting scikit-learn
```

9. Install matplotlib

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install matplotlib
Collecting matplotlib
```

10. Install Seaborn

```
(virtual_env_name) PS C:\Users\Avinandan\Credit Card Fraud Detection project> pip install seaborn
```

11. Download the Credit Card dataset from:

https://www.kaggle.com/mlg-ulb/creditcardfraud/version/3

And keep it in the project folder.

Structure

- **Let's understand the data**
- **Perquisites**
- **Data analysis**
- **Modeling**

Let's understand the data

This is a credit card transaction data for two days of September 2013 for European cardholders. There are total 2,84,807 transactions. This is a highly imbalanced dataset that we need to handle.

Source of the dataset:

The dataset has been collected from (https://www.kaggle.com/mlg-ulb/creditcardfraud/version/3)

Lets us start by loading the dataset to a Pandas Data Frame.

To have 1st 5 rows with column we used df.head() function by importing pandas.

```
[1]: import pandas as pd
     df = pd.read_csv('creditcard.csv')
     df.head()
```

[1]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267

5 rows × 31 columns

Figure: 1st 5 rows and 31 columns

To look how many rows and columns in the data set.

We use:

df.shape()

```
[2]: df.shape
```

```
[2]: (284807, 31)
```

Figure: Shape of the data set.

In data set when a class is ‘0’ the transaction is non fraudulent and ‘1’ means transaction is fraudulent.

We view such classes in our data set by:

```
[3]: df['Class'].value_counts()
```

```
[3]: 0    284315
     1       492
     Name: Class, dtype: int64
```

Figure: Class Frequency

i.e. non-fraudulent classes are: 284315 and fraudulent transaction are: 492 in our data set.

To see the columns we all the columns in a data set, we observe there is a column like “Time”, “Class”, “Amount”, “V1”,…,”V28”.

```
[3]: df.columns

[3]: 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', 'Amount',
       'Class'],
      dtype='object')
```

Figure: Columns in dataset.

Column descriptions:

- **Time**: Number of seconds elapsed between this transaction and the first transaction in the dataset.
- **Amount**: Transaction Amount.
- **Class**: 1 for fraudulent transactions, 0 for non-fraudulent.
- **V1…V28**: These are the features of the dataset. It may be the result of a PCA Dimensionally reduction to protect user identities and sensitive features. But this is all we have to work with.

“Class” is the feature that we need to predict from this dataset using all the 28 features,”Time” and “Amount”.

The features V1,…,V28 are dimensionality reduction of some original features. The problem with this feature is that we are not aware of what feature it is, so logically tackling the features; in this case, is not possible. Say, if we have a feature called “Name” and we wanted to predict whether a transaction is fraudulent or not. Here , we can clearly say that the feature “Name” is not correlated with the type of prediction. But in this case of the dataset we are using, we have no clue about the feature.

In this case we will only be dependent on statistical metrics and relations to choose the best features for predicting the desired class.

Now we will see another problem , i.e., imbalanced data, which we need to tackle for getting a good model for the problem space.

What is an imbalanced dataset?

An **imbalanced dataset** is when one ouput class has extremely high entries compared to the other output class. In this case , the positive class, i.e. , the fraudulent transaction class , has a few entries compared to the non-fraudulent/normal transaction class. So, the positive class(i.e., frauds) only accounts for 0.172% of the total transactions.

We will verify these numbers when the dataset is loaded.

```
[3]: df['Class'].value_counts()

[3]: 0    284315
     1       492
     Name: Class, dtype: int64
```

Figure : Class Frequency

It is evident that this is a highly imbalanced dataset, and among 2,84,807 total records are class 1, i.e., those records are a fraudulent transaction. So, we can only confirm that the positive class only accounts for 0.172% of the entire dataset.

Now let us visually see how imbalanced the dataset is?

```
[5]: import seaborn as sns
     from matplotlib import pyplot as plt
     sns.countplot('Class', data=df)
     plt.title('Class Distribution \n (0: No Fraud || 1: Fraud)', fontsize=14)

     c:\users\avinandan\credit card fraud detection project\virtual_env_name\lib\site-packages\seaborn\_de
     ing: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional ar
     passing other arguments without an explicit keyword will result in an error or misinterpretation.
       warnings.warn(

[5]: Text(0.5, 1.0, 'Class Distribution \n (0: No Fraud || 1: Fraud)')
```

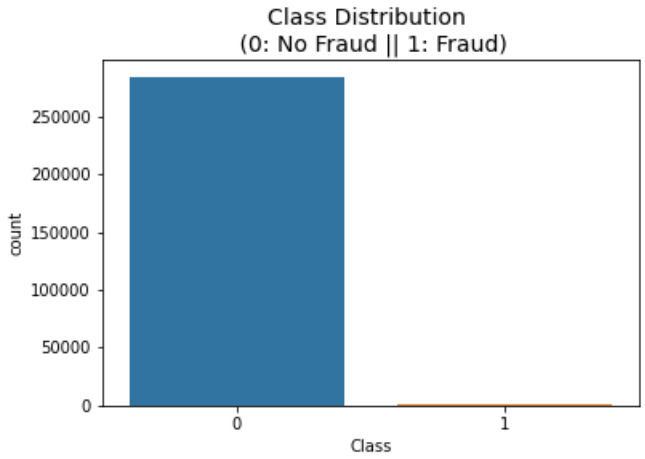


Figure: Class Distribution

We can see now the difference visually and the difference is so high that we can coin this dataset as a highly imbalanced dataset. Next we can check out for NULL values in the dataset.

```
[6]: df.isnull().sum().max()

[6]: 0
```

Figure: NULL count

Hopefully, we don't have any NULL values to handle in the dataset. So, going ahead, we will only take care of the imbalanced nature of the dataset and making sense of the features.

Knowing the features

Till now, we have seen many things about the dataset and its nature. In this section, we will analyze the features, know their types, and some similar information.

Let us see some of the values for all the features and know, how the value looks.

```
[7]: df.sort_index(axis=1).head(3)
```

[7]:

	Amount	Class	Time	V1	V10	V11	V12	V13	V14	V15	...	V26	V27	V28	V3	
0	149.62	0	0.0	-1.359807	0.090794	-0.551600	-0.617801	-0.991390	-0.311169	1.468177	...	-0.189115	0.133558	-0.021053	2.536347	1.378
1	2.69	0	0.0	1.191857	-0.166974	1.612727	1.065235	0.489095	-0.143772	0.635558	...	0.125895	-0.008983	0.014724	0.166480	0.448
2	378.66	0	1.0	-1.358354	0.207643	0.624501	0.066084	0.717293	-0.165946	2.345865	...	-0.139097	-0.055353	-0.059752	1.773209	0.379

3 rows × 31 columns

Figure: Dataset rows

We can observe a few things from the above code output.

- Class is either 0 or 1.
- An amount is a floating-point number (and it cannot be negative), any financial transaction cannot be negative.

- Time as we know it will start from 0 and always be a positive integer.

- V1…V28 can be negative or positive.

Let us see all the information about the data frame now.

```
[9]: df.sort_index(axis=1).info()
```

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

Figure : Dataset Information

We can see it's a relatively large dataset compared to the previous dataset used. All of them are non-null, so we don't have to take care of null values for this dataset.

Training a model with all the features generally don't make sense. We need to choose the features in such a way that those features generally don't make sense. We need to choose the features in such a way that those features will have some contribution to the decision of the output class.

Correlation of the dataset will give us a rough idea of what we are dealing with.

We will try to see a correlation plot with a heat-map to figure out the highly correlated and non-correlated and non-correlated features.

[10]:
```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(12,8))
sns.heatmap(df.corr(),cmap="coolwarm_r",annot=False)
```

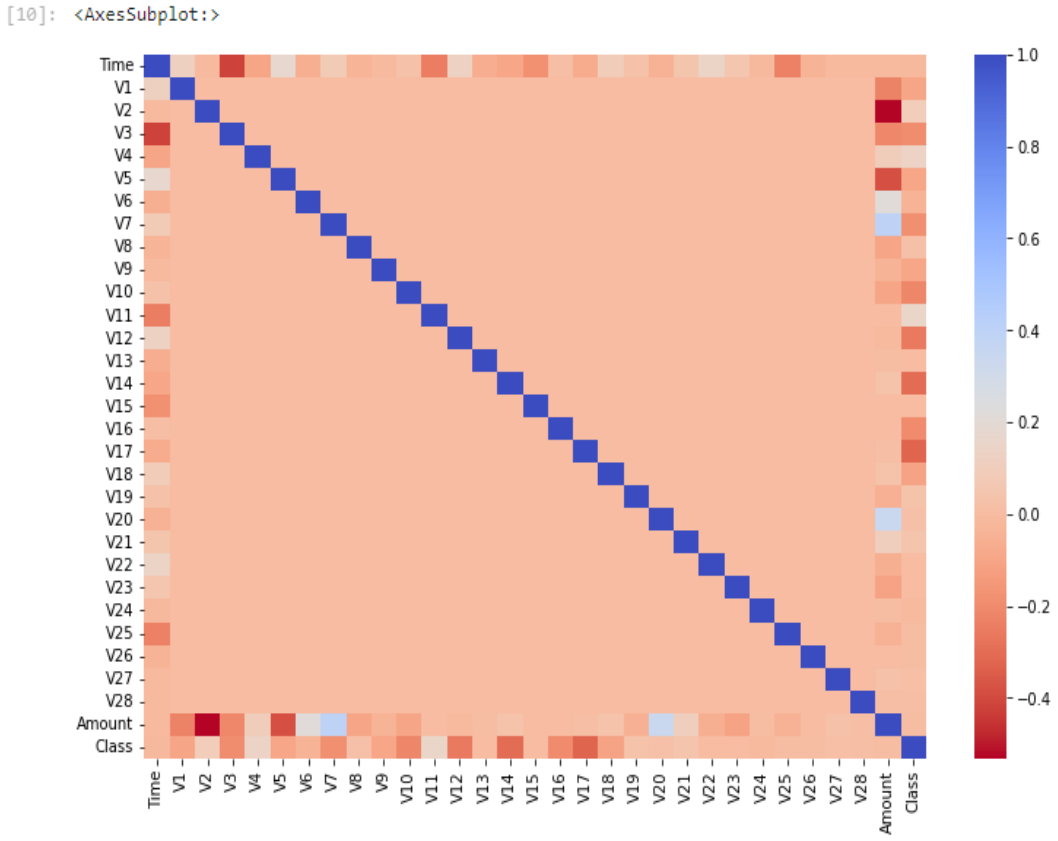


Figure: *Correlation of heat map*

We now need to find out some features among 28 dimensionally reduced features that we can use to train model. We went with the heat map and not the table because seeing the numeric table will be confusing for everyone, and the number of features is so high tracking them is a real problem.

But anything visually gives us higher information compared to reading through a large table.

Let us analyze the heat map.

Here red is ***negatively*** correlated, and **blue** is ***positively*** correlated, and **peach** represents not correlated.

For simplicity, we will only compare the correlation against "Class," because in the end, we will try to predict the "Class" of the record.

If we analyze the features, then we can eliminate some of the features seeing the correlation between the features. In this case, we take features that are not highly correlated and zero as well. For this data set, we will use both linear and non-linear models.

And also to start with we need to apply the normalization to rest of the columns as well.

Perquisites:

We will introduce some concepts:

1. Normalization.

2. Cross-Validation
3. PCA

Into our model to make it more efficient.

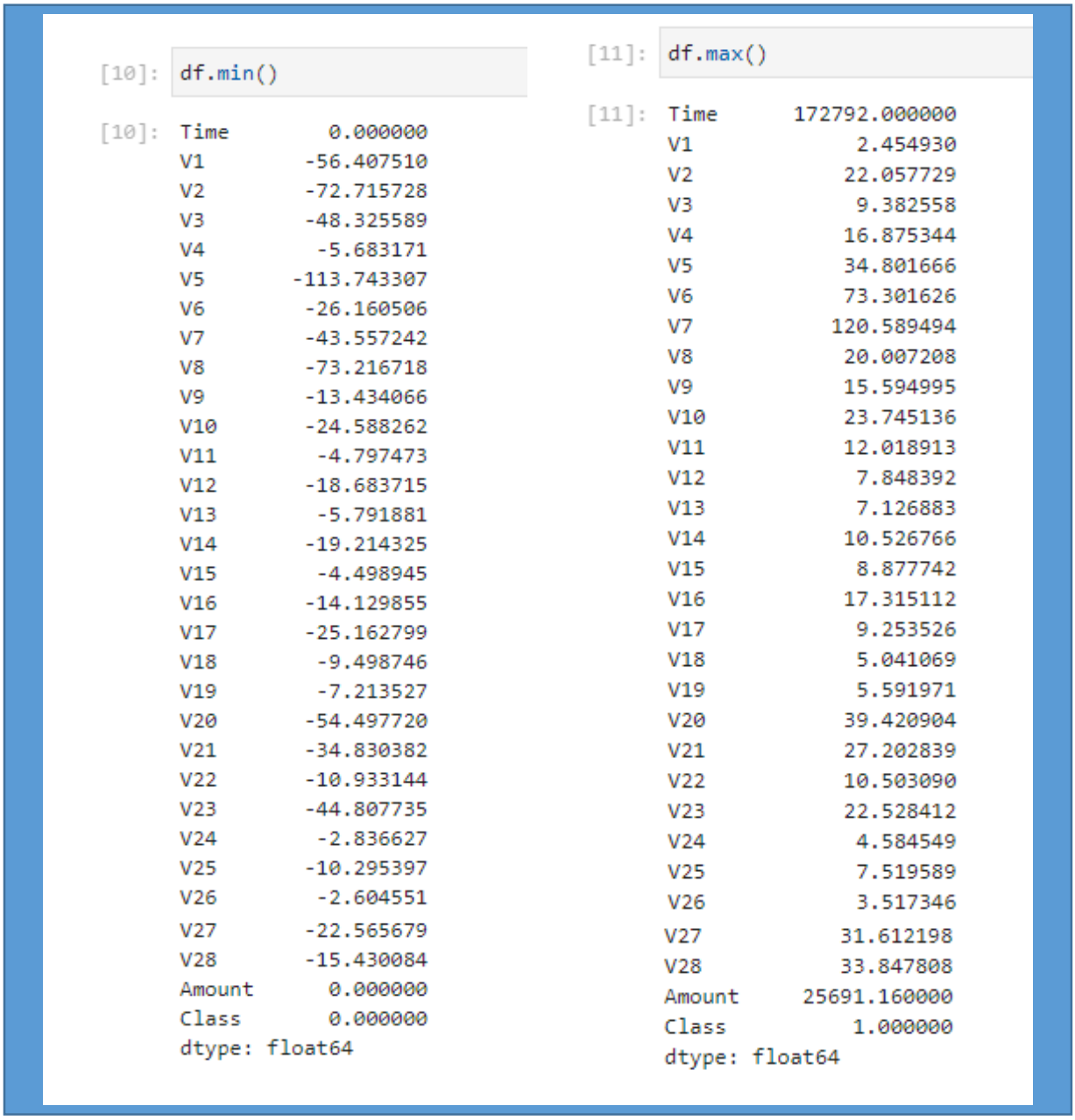
Normalization/Feature Scaling

Normalization is used in a variety of ways in statistics. It is also known as **feature scaling**. The meaning we are going after, is the normalization of the range of the data to a standard scale.

- **Case 1:** Say we have four lengths; all of them are in centimeters and only one of them is inches. So, in those cases , we need to follow one standard and generalize all length units.

- **Case 2:** We have two features like Age and Salary and we can easily say that both of the features will have a different range. Hence, we will use normalization/scaling techniques to bring them on the same scale.

As seen in the previous section, that all the values had a different range. Let's consolidate those values and see their ranges for all the features.



```
[10]: df.min()

[10]: Time         0.000000
      V1         -56.407510
      V2         -72.715728
      V3         -48.325589
      V4          -5.683171
      V5        -113.743307
      V6         -26.160506
      V7         -43.557242
      V8         -73.216718
      V9         -13.434066
      V10        -24.588262
      V11         -4.797473
      V12        -18.683715
      V13         -5.791881
      V14        -19.214325
      V15         -4.498945
      V16        -14.129855
      V17        -25.162799
      V18         -9.498746
      V19         -7.213527
      V20        -54.497720
      V21        -34.830382
      V22        -10.933144
      V23        -44.807735
      V24         -2.836627
      V25        -10.295397
      V26         -2.604551
      V27        -22.565679
      V28        -15.430084
      Amount       0.000000
      Class        0.000000
      dtype: float64
```

```
[11]: df.max()

[11]: Time      172792.000000
      V1             2.454930
      V2            22.057729
      V3             9.382558
      V4            16.875344
      V5            34.801666
      V6            73.301626
      V7           120.589494
      V8            20.007208
      V9            15.594995
      V10           23.745136
      V11           12.018913
      V12            7.848392
      V13            7.126883
      V14           10.526766
      V15            8.877742
      V16           17.315112
      V17            9.253526
      V18            5.041069
      V19            5.591971
      V20           39.420904
      V21           27.202839
      V22           10.503090
      V23           22.528412
      V24            4.584549
      V25            7.519589
      V26            3.517346
      V27           31.612198
      V28           33.847808
      Amount     25691.160000
      Class          1.000000
      dtype: float64
```

Fig: Range for all columns

From the above image, we can the max and min of the feature side by side. We need to apply some scaling concepts to standardize/normalize the dataset.

Seeing the feature ranges, we can see ‘’Time’’ and ‘’Amount’’ has not been ended. To gain more understanding about the dataset , we need to understand different types of scaling techniques that will help us to deal with the dataset.

Data Analysis

We need to work on the data and analyze all the features and later choose the best features for predicting the class.

Let us start by seeing some distribution of the features and analyze them.

Previously we have seen ‘’Time ‘’ was not scaled and the range for that feature was not scaled and the range for that feature was [0,172792]. We can check the distribution of the feature and see how it looks.

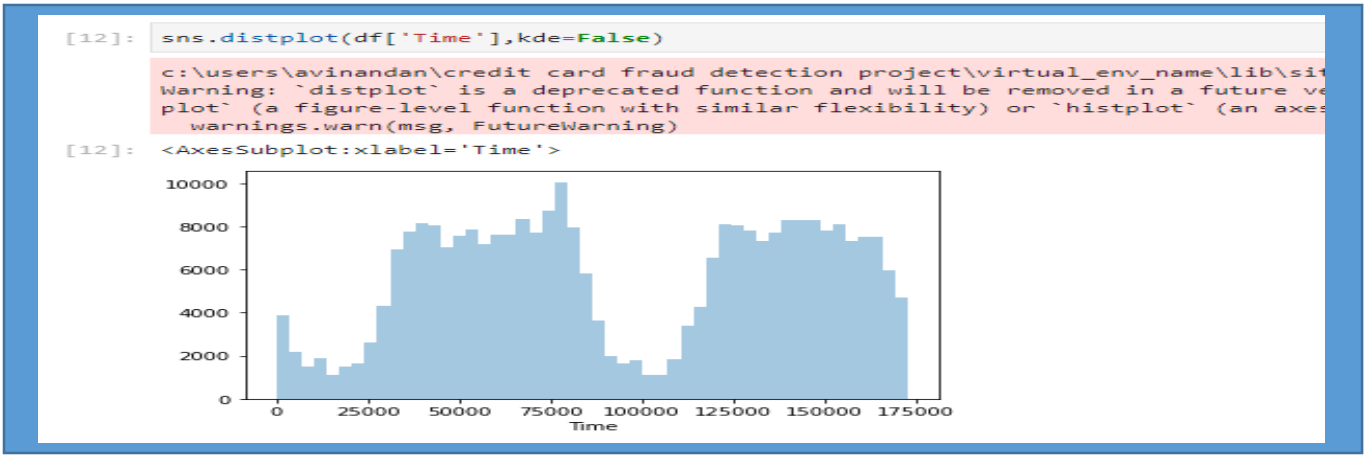


Fig: Bi-modal ‘’Time’’ distribution plot.

Seeing the above distribution, we can see two spikes in transaction one at 5000 sec and another at the 13500-sec range. That means we can see there are two similar modes in the distribution.

That is a Bimodal Distribution and it is a continuous probability distribution with two different modes.

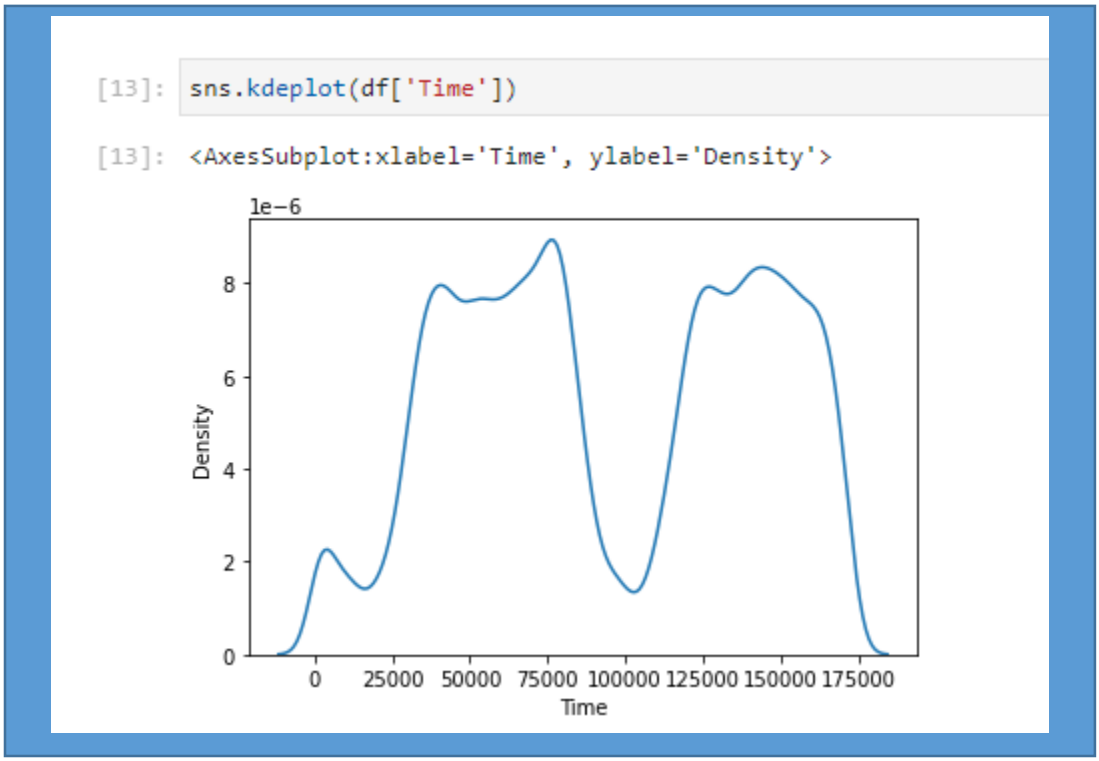


Fig: KDE for ''Time''.

KDE clearly shows that there are two similar modes to this continuous distribution.

Note: There can be more than two modes in a continuous probability distribution. and those are known as Multimodal distribution.

We have seen the distribution for “Time,” and now we need to see the distribution for the “Amount”.

We are expecting a skewed distribution as we know it cannot be uniform because the extremely high-value credit card transactions are rare compared to the average amount transaction.

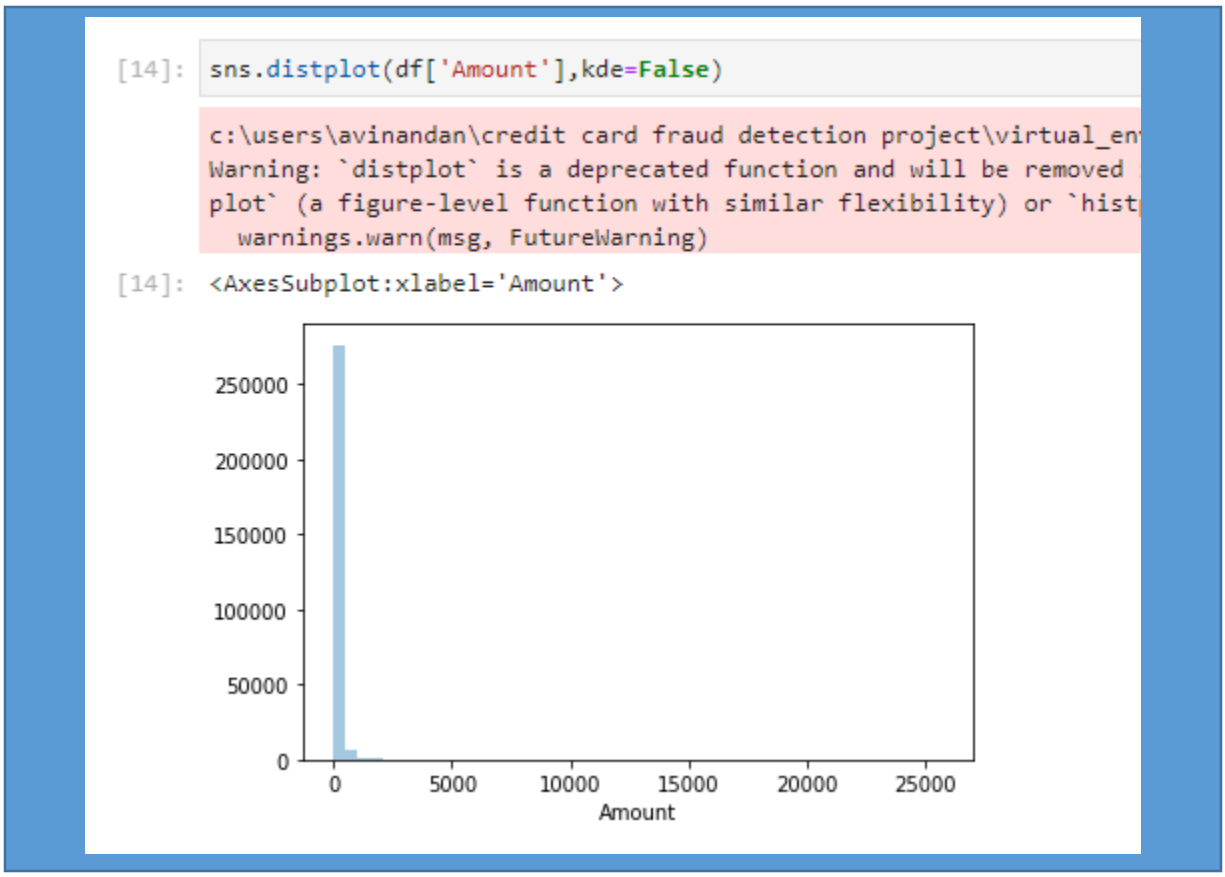


Fig: “Amount” Frequency Distribution.

Seeing the distribution, we can confirm the hypothesis about the skewness. We see the mode distribution lies at far left of the distribution, may be the mode will range from [1,1000] for

this distribution. But this kind of distribution is not odd because it is an expected distribution for any transaction dataset.

We can see the KDE plot for the same as well.

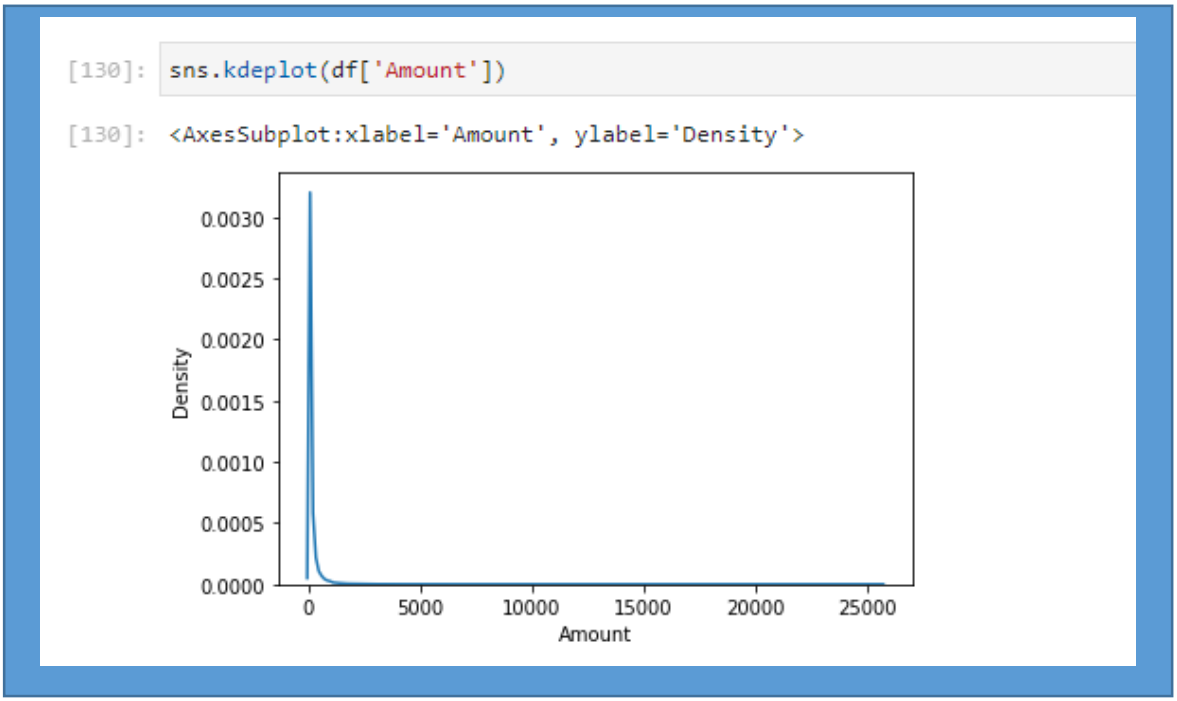


Fig: KDE for "Amount"

KDE plot also shows some density at the right end, and those are outliers. And for this business statement, the outliers are extremely important because there can fraudulent transaction.

We should try to see if there is any visual link/similarity among distributions.

```
[131]: fig,ax= plt.subplots(1,2, figsize=(14,4))
       sns.distplot(df['Amount'],ax=ax[0],color='r')
       ax[0].set_title('Distribution of Transaction Amount', fontsize=14)
       sns.distplot(df['Time'],ax=ax[1],color='b')
       ax[1].set_title('Distribution of Transaction Time', fontsize=14)
       plt.show()
```

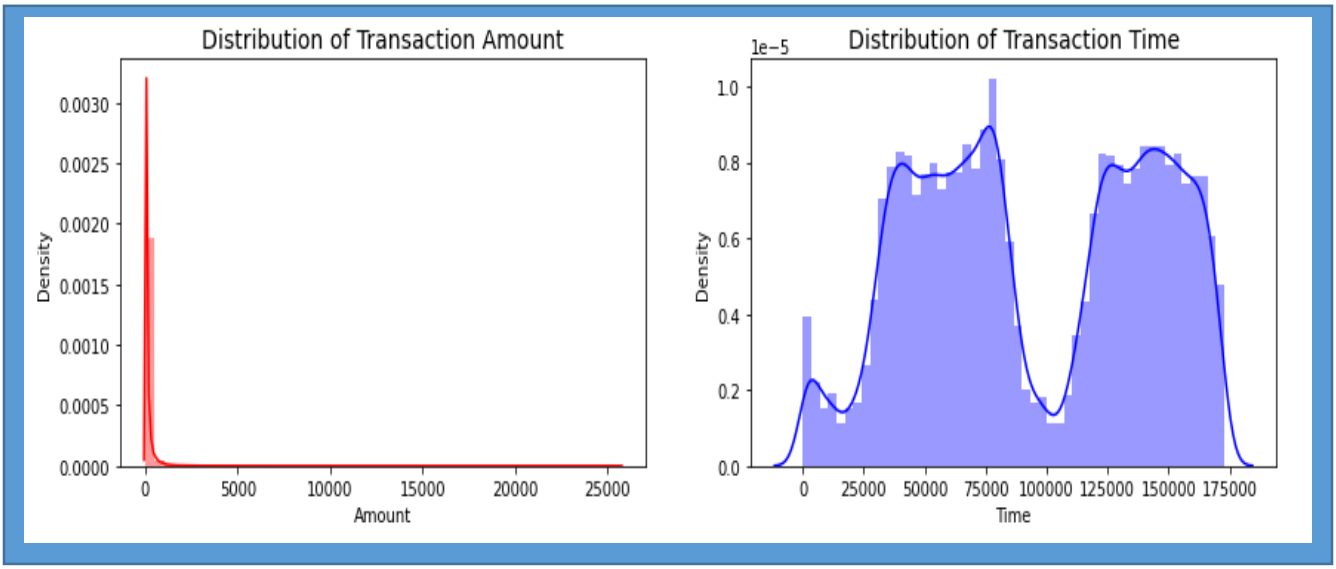


Fig: Comparison for an "Amount" and "Time" KDEs.

Unfortunately, from the distribution, we cannot make any sense. So we need to drop this idea to compare "Time" and "Amount" distribution separately.

It will make sense if we plot a chart between **"Time" vs. "Amount"**. We should know that for every "time" what will be the "amounts" like it is shown above.

```
[132]: df[['Time','Amount']].groupby(['Time']).head()
```

[132]:

	Time	Amount
0	0.0	149.62
1	0.0	2.69
2	1.0	378.66
3	1.0	123.50
4	2.0	69.99
...	...	...
284802	172786.0	0.77
284803	172787.0	24.79
284804	172788.0	67.88
284805	172788.0	10.00
284806	172792.0	217.00

279146 rows × 2 columns

Fig: Summed "Amount" grouped by "Time."

For any plot, especially the bar plot, we cannot plot multiple values for the same time interval, so it's better to plot the sum of" Amount" for each of the time intervals.

We are expecting some intervals, which will have a spike, in the sum of their total transaction amount.

Let us go ahead and plot such kind of bar plot.

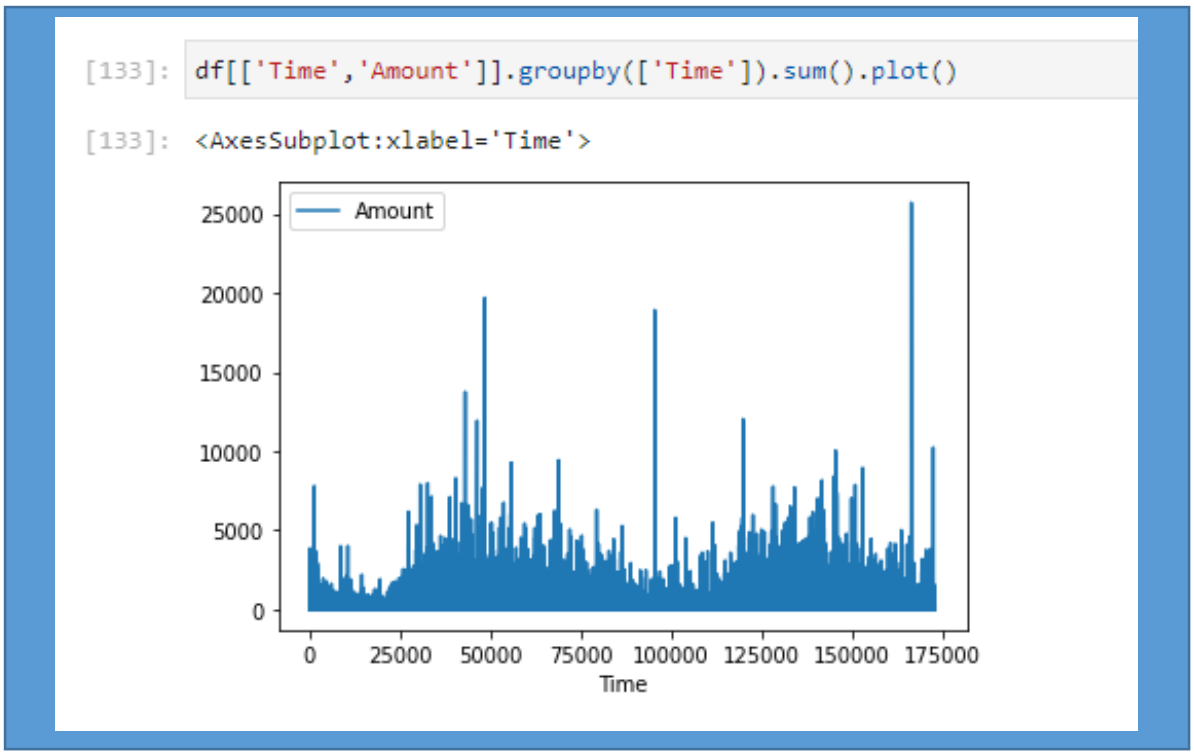


Fig: Summed "Amount" Bar Plot.

This shows what we exactly, thought, so; this gives us a rough idea about the sum of the amount distributed.

Here, we can only see the frequency of the amounts for the given time. But, need a plot that will help us to see the fraudulent transactions and the range for amounts.

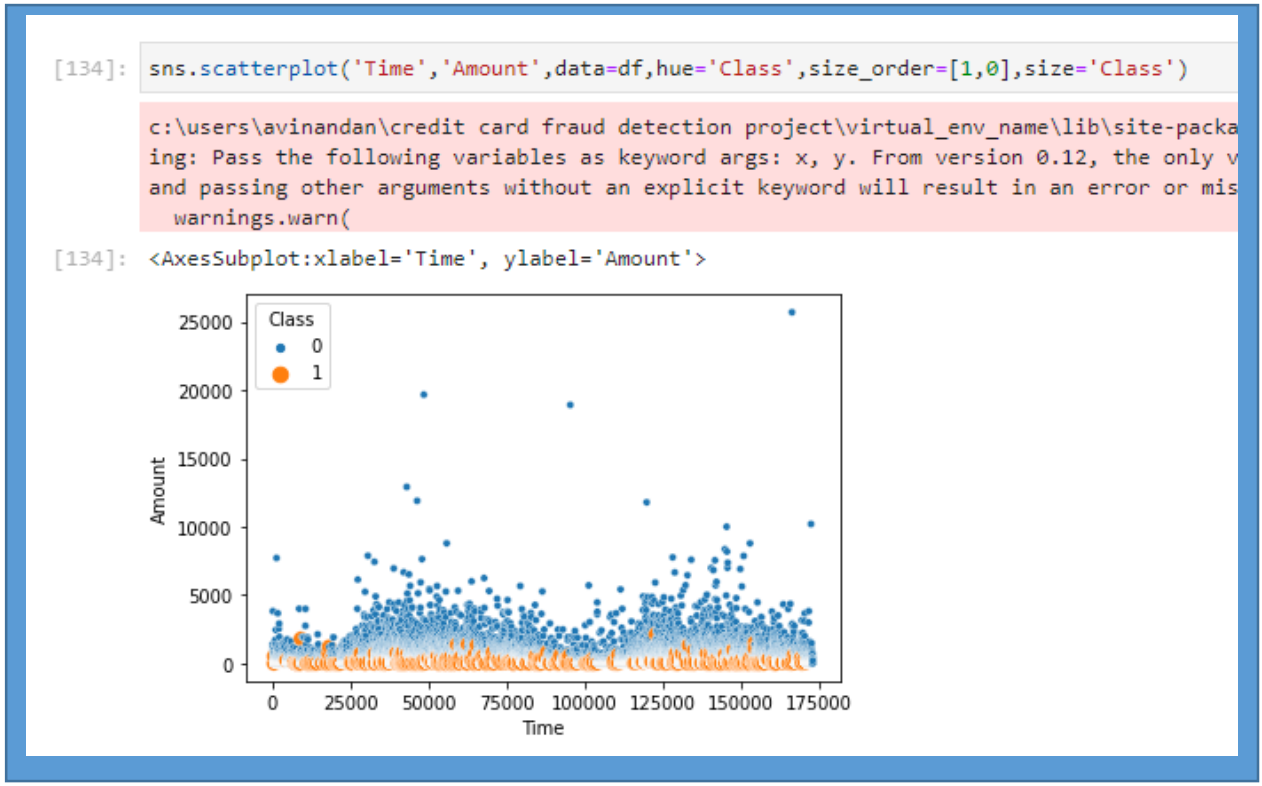


Fig: Class-wise "Amount" vs "Time" Scatter Plot.

The above figure shows the range for the fraudulent transaction to be around[0,2500]. We can test this out and the range of the fraudulent transaction amount.

```
[135]: df.Amount.min(), df.Amount.max()

[135]: (0.0, 25691.16)
```

Fig: Range for Fraudulent Class.

We can confirm the hypothesis seeing the range above. One more thing we can check and think about is, what are the total number of records whose amount value is Zero.

Does it make sense to keep zero-valued transactions in the dataset? How can there be a fraud when the transaction value is zero?

```
[41]: df[df["Amount"]==0].shape

[41]: (1825, 31)
```

Fig: Count of records where "Amount" is zero.

We have a small chunk of records where "Amount" equals to zero. We need to see the breakdown of "Class" as well and decide whether to keep it or ignore it.

```
[42]: df[df["Amount"]==0]["Class"].value_counts()

[42]: 0    1798
      1      27
      Name: Class, dtype: int64
```

Fig: Class-wise Zero "Amount" Transaction.

Seeing the breakdown, we can clearly say that it is not significant enough to impact the decision or prediction. But the question we raised about the zero amount transactions does not have a proper logical solution as we cannot back it up with any proof. The dataset description doesnot say much about this type of transaction, so the chances of removing this kind of transaction are pretty high.

We will see more distributions and later decide to keep it or not.

Lets think more about the dataset and see what can be done with it.

So, now we should try to plot distributions for all the class only for “Time” and “Amount”. Before that, we need to split the dataset into two different data frames based on “Class”.

```
[43]: f_df = df[df['Class']==1]
      n_df = df[df['Class']==0]
      (f_df.shape, n_df.shape)

[43]: ((492, 31), (284315, 31))
```

Fig: Shape of Fraudulent and Non-Fraudulent Class.

For some time, we will ignore those transaction zero records and consider the entire dataset. Seeing the above code, we now have two data frames, one containing all the fraudulent transactions and another containing all the non-fraudulent transactions.

With those, we can now individually plot the distributions. We can start with the “Amount” distribution.

```
[44]: fig,ax= plt.subplots(1,2, figsize=(14,4))
sns.distplot(n_df['Amount'], kde=False,ax=ax[0],color='r')
ax[0].set_title('Non-Fraudulent: Transaction Amount', fontsize=14)
sns.distplot(f_df['Amount'],kde=False,ax=ax[1],color='b')
ax[1].set_title('Fraudulent: Transaction Amount', fontsize=14)
plt.show()
```

```
c:\users\avinandan\credit card fraud detection project\virtual_env_name\lib\site-packages\seaborn\distributions.py:261
Warning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use ei
plot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

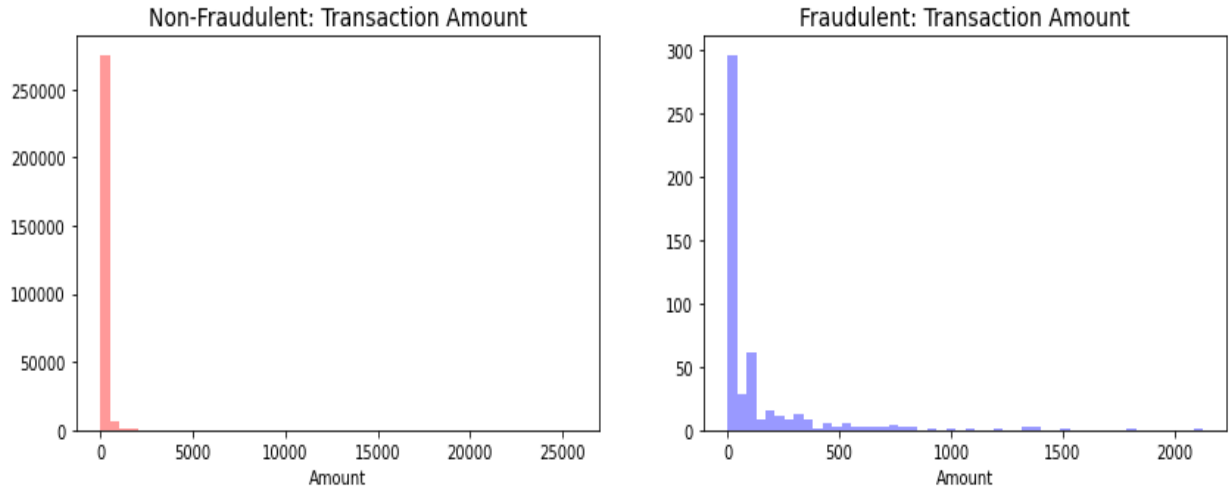


Fig: "Amount" Frequency Distribution for Fraudulent and Non-Fraudulent Class.

Seeing the above image, we can see the distribution is same or nearly identical if we compare them. KDE will be same as well, which we can confirm by plotting it.

```
[45]: fig,ax= plt.subplots(1,2, figsize=(14,4))
sns.kdeplot(n_df['Amount'],ax=ax[0],color='r')
ax[0].set_title('Non-Fraudulent: Transaction Amount(KDE)', fontsize=14)
sns.kdeplot(f_df['Amount'],ax=ax[1],color='b')
ax[1].set_title('Fraudulent: Transaction Amount(KDE)', fontsize=14)
plt.show()
```

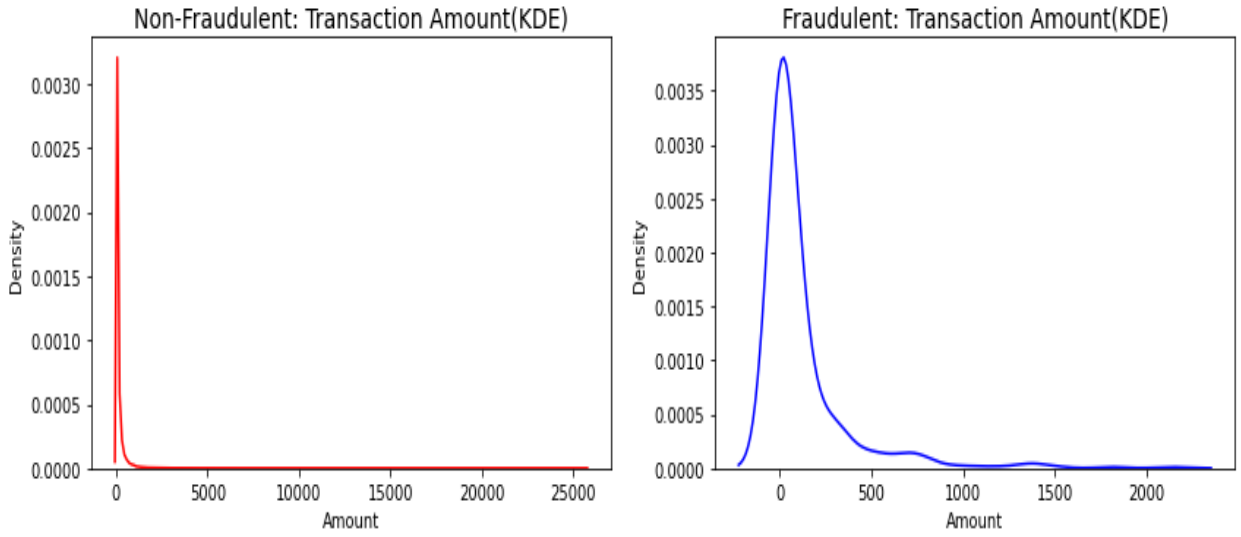


Fig: "Amount"KDE for Fraudulent and Non-fraudulent Class.

The KDE is similar and the distribution looks similar. The fraudulent transaction KDE is a bit spread compared to the non-fraudulent transaction but, it won't be of much use because the size of the dataset is small.

We need to perform something similar for "Time" as well and see the distribution.

```
[46]: fig,ax= plt.subplots(1,2, figsize=(14,4))
sns.distplot(n_df['Time'], kde=False,ax=ax[0],color='r')
ax[0].set_title('Non-Fraudulent: Transaction Time', fontsize=14)
sns.distplot(f_df['Time'],kde=False,ax=ax[1],color='b')
ax[1].set_title('Fraudulent: Transaction Time', fontsize=14)
plt.show()
```

```
c:\users\avinandan\credit card fraud detection project\virtual_env_name\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

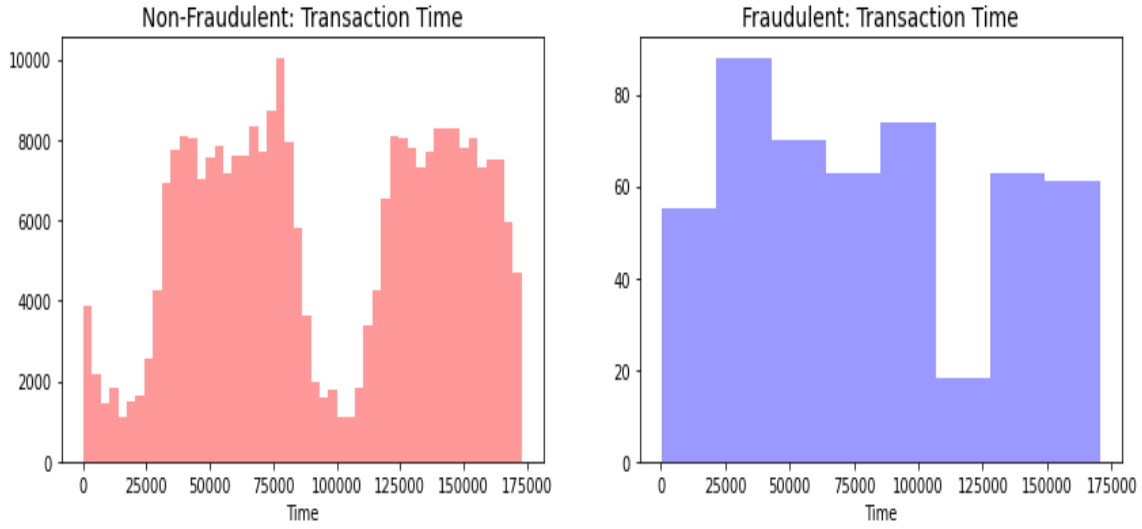


Fig: "Time" frequency. Distribution for Fraudulent and Non-Fraudulent Class.

Interestingly, the "Time" distribution for Fraudulent transaction is different. We can confirm and analyze after plotting the KDE.

```
[47]: fig,ax= plt.subplots(1,2, figsize=(14,4))
sns.kdeplot(n_df['Time'],ax=ax[0],color='r')
ax[0].set_title('Non-Fraudulent: Transaction Time(KDE)', fontsize=14)
sns.kdeplot(f_df['Time'],ax=ax[1],color='b')
ax[1].set_title('Fraudulent: Transaction Time(KDE)', fontsize=14)
plt.show()
```

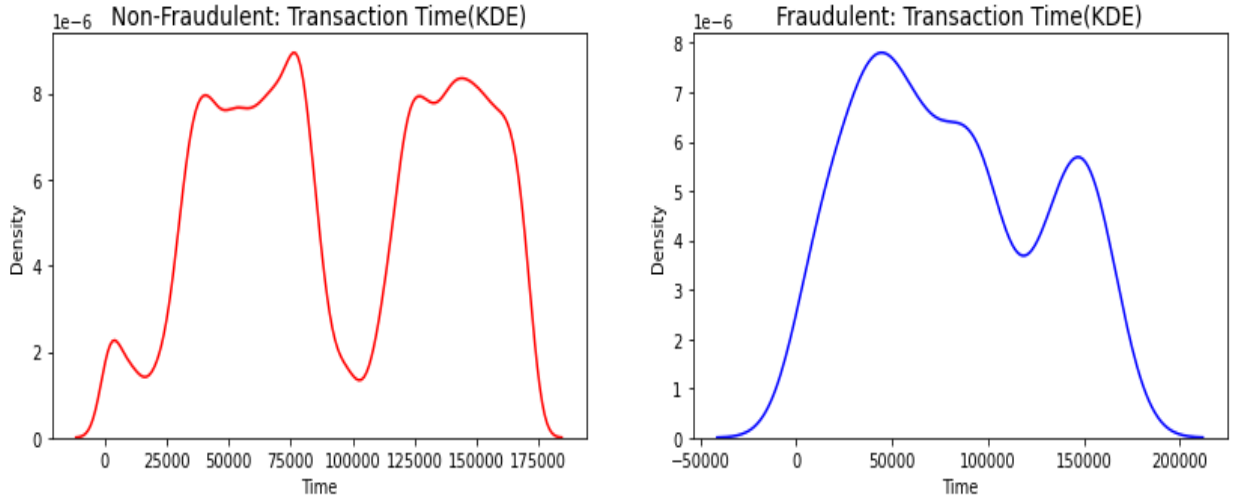


Fig: "Time" KDE for Fraudulent and Non-Fraudulent Class.

Seeing the above distribution, we can draw two conclusions:

1. Non-fraudulent transaction for both "Time" and "Amount" looks the same when compared with the entire population. That is because the size of the Fraudulent dataset is so less the impact is not significant enough to change the distribution.

2. Fraudulent transaction for "Time" looks different, and that has some significance, i.e., the density of the distribution is higher at the beginning of time, and there is only one statistical model to the distribution. We need to keep this

in mind, when scale the data and make sure this kind of patter is not lost.

Let us go ahead and see the distribution for the non-fraudulent dataset.

```
[48]: n_df[['Time','Amount']].groupby(['Time']).sum().plot()

[48]: <AxesSubplot:xlabel='Time'>
```

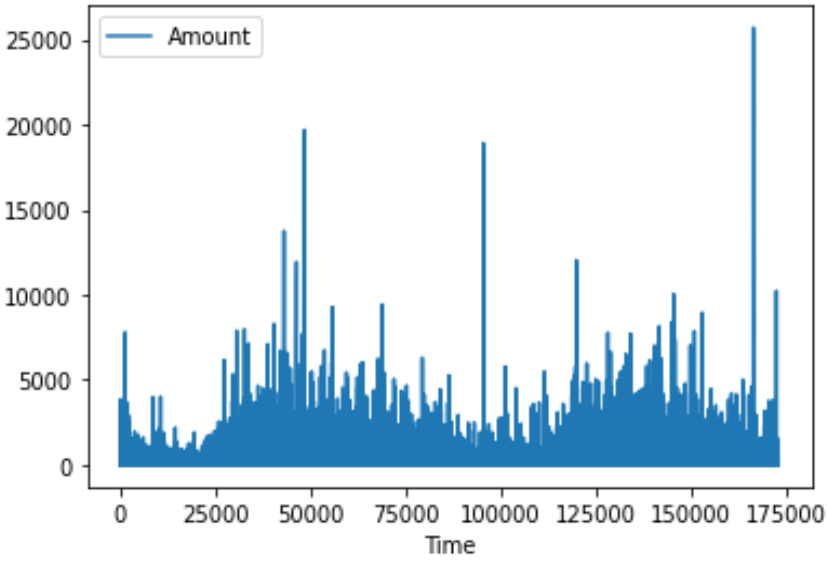


Fig: Non-Fraudulent Summed “Amount” vs “Time”.

As with the previous plots for this class this is highly similar to the plot for the entire population. But we are expecting something different from the Fraudulent dataset as the distribution for the “Time” feature was a bit different.

```
[49]: f_df[['Time','Amount']].groupby(['Time']).sum().plot()
```

```
[49]: <AxesSubplot:xlabel='Time'>
```

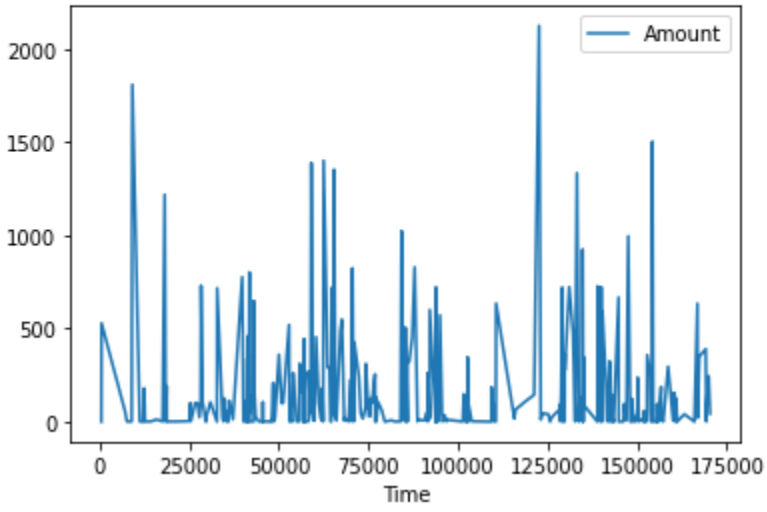


Fig: Fraudulent Summed "Amount" vs. "Time"

As we thought, the plot will be different from the hypothesis, and we find the hypothesis to be true. We can see that the sum of the amount is not that high compared to Non-Fraudulent one, and that is obvious but, we can say the transaction amount is not of high variance. We can confirm the total amount of different classes.

```
[50]: print("Entire Dataset:" +str(df.Amount.sum()))
      print("Non-Fraudulent Dataset:" +str(n_df.Amount.sum()))
      print("Fraudulent Dataset:" +str(f_df.Amount.sum()))

      Entire Dataset:25162590.009999998
      Non-Fraudulent Dataset:25102462.04
      Fraudulent Dataset:60127.97
```

Fig: Total summed amount for different class.

We can see the total amount for each group and removing the zero-amount transaction would not affect the “Amount” distribution, and it won’t change the “Time” distribution as the size of the expectation is very low. So, it will be safe to remove those records.

```
[51]: df.drop(df[df.Amount==0].index, inplace=True)
      n_df = df[df["Class"]==0]
      f_df = df[df["Class"]==1]
      (df.shape,n_df.shape,f_df.shape)

[51]: ((282982, 31), (282517, 31), (465, 31))
```

Fig: Removing Zero “Amount” Transaction.

With the above code snippet, we have removed those transactions. Moving forward, we will use this dataset for all analyses and modelling.

Scaling

Scaling is a method used to normalize the range of independent variables or features of data. In data processing, it is also known as data normalization and is generally performed during the data preprocessing step.

Standard Scaler

Standard scalar from the sklearn implements standardization/Z-score normalization.

$$x' = \frac{x_i - \bar{\mathbf{x}}}{\sigma}$$

Where , $\bar{\mathbf{x}}$ is the mean and σ is the standard deviation.

PCA

Principal Component Analysis(PCA) is a statistical method to explain variance and covariance structure of a set of variables through linear combination. It uses the concept of orthogonal transformation to convert the set of variables(mostly correlated variables) into a set of linearly uncorrelated values, i.e. also known as Principal component.

We will see briefly in the form of an algorithm of how it works:

- **Step1:** Get the data in the form of $m \times n$ matrix.
- **Step2:** Standardize the matrix.
- **Step3:** Calculate the Covariance Matrix.
- **Step 4:** Calculate Eigenvectors and Eigenvalues of the Covariance Matrix, i.e. Eigen decomposition of the Covariance Matrix.

- **Step 5:** Choose the Principal Components and form a feature vector.

- **Step 6**: Deriving the new dataset and form the clusters.

PCA is one of the popular methods to perform dimension reduction on a large dataset, and it helps in multiple ways like visualization, handling a smaller number of columns/feature for modelling.

We know from the data description that all the features from V1,…,V28 are scaled as those are the output result of a Dimension Reduction Algorithm i.e., PCA.

So we will only implement the standard scaling for "Time" and "Amount".

[52]:
```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
df['scaled_amount']= std_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time']= std_scaler.fit_transform(df['Time'].values.reshape(-1,1))
print("Scaled Amount")
print(df['scaled_amount'])
print("")
print("Time of Scaled Amount")
print(df['scaled_time'])
```

```
Scaled Amount
0          0.242005
1         -0.343785
2          1.155155
3          0.137868
4         -0.075469
              ...
284802    -0.351439
284803    -0.255675
284804    -0.083881
284805    -0.314641
284806     0.510639
Name: scaled_amount, Length: 282982, dtype: float64

Time of Scaled Amount
0         -1.997561
1         -1.997561
2         -1.997540
3         -1.997540
4         -1.997519
              ...
284802     1.641389
284803     1.641410
284804     1.641431
284805     1.641431
284806     1.641515
Name: scaled_time, Length: 282982, dtype: float64
```

Fig: Standard Scaling the features.

Here, we are using different features to store the scaled values because we need to compare them with the original values and choose to keep the best feature.

We need to visualize the change so that we can decide whether to keep or discard it.

The below function will help to plot the comparison with the original dataset.

```
[53]: def compare_kde(col1,col2,name):
          fig,ax = plt.subplots(2,2,figsize=(16,7))
          sns.kdeplot(df[df['Class']==0]['Amount'],ax=ax[0][0],color='r')
          ax[0][0].set_title('Non-Fraudulent: Transaction Amount(Original)',fontsize=12)
          sns.kdeplot(df[df['Class']==1]['Time'],ax=ax[0][1],color='b')
          ax[0][1].set_title('Fraudulent: Transaction Time(Original)',fontsize=12)
          sns.kdeplot(df[df['Class']==0][col1],ax=ax[1][0],color='r')
          ax[1][0].set_title('Non-Fraudulent: Transaction Amount('+name+')',fontsize=12)
          sns.kdeplot(df[df['Class']==1][col2],ax=ax[1][1],color='b')
          ax[1][1].set_title('Fraudulent: Transaction Time('+name+')',fontsize=12)
          plt.show()

[54]: compare_kde("scaled_amount","scaled_time","Standard Scaler")
```

Fig: Function to plot KDE.

From the above code snippet, we are expecting the same distribution. The original distribution will be on top row, and the scaled distribution will be on the bottom row.

We can see the plot below and analyze it further.

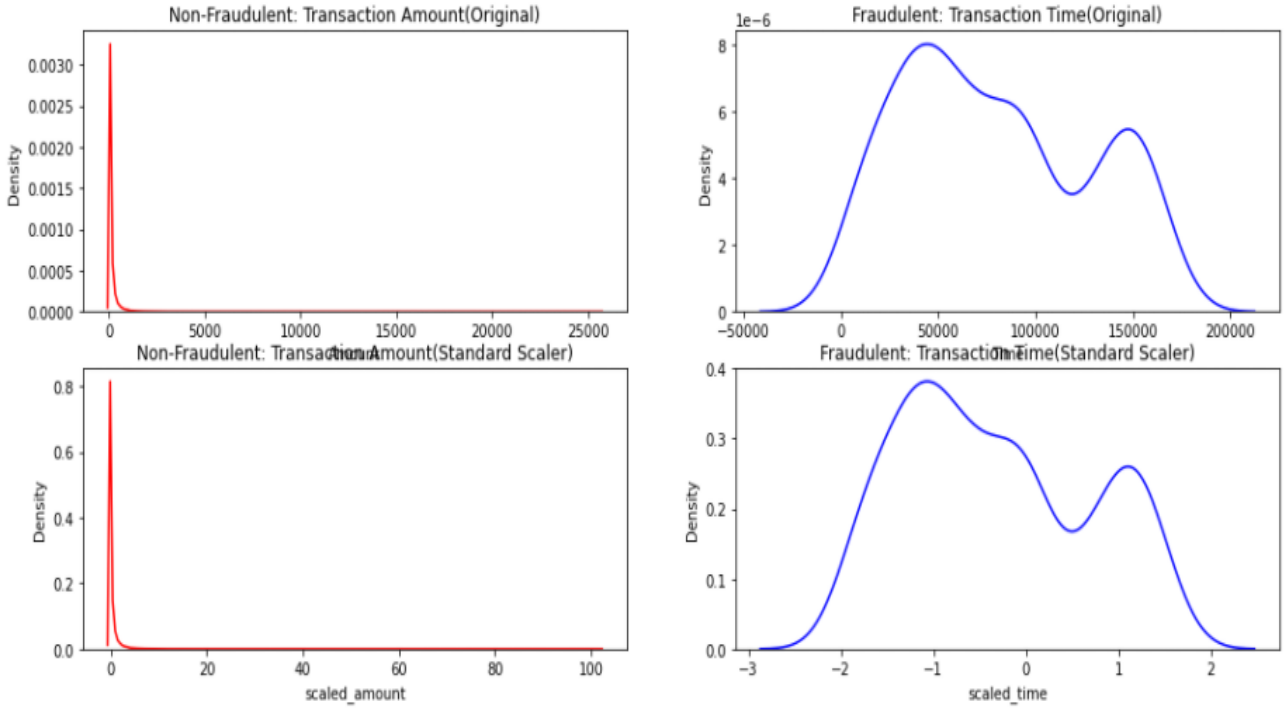


Fig: Comparing Standard Scaled Features with Original Features.

We can see from the above-scaled distribution that the values of the axis have been changed and scaled-down a lot, keeping the distribution same. If we now see the difference between the “Time” and “Amount” scale, it has significantly reduced from the original one.

Let us see the same in correlation matrix and see if there are any changes or not.

```
[55]: plt.figure(figsize=(12,8))
      sns.heatmap(df.corr(),cmap='coolwarm_r',annot=False)
```

[55]: <AxesSubplot:>

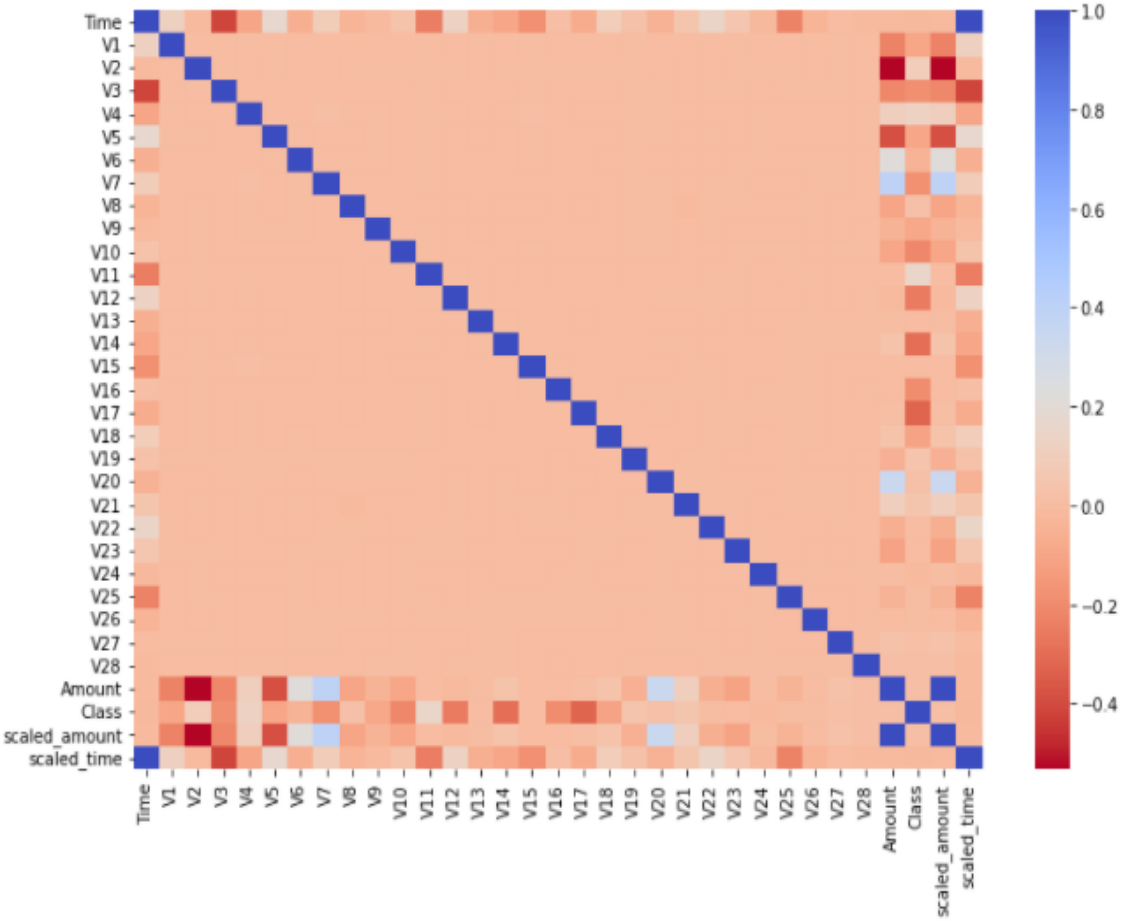


Fig: Correlation heat map after scaling.

Interestingly, after scaling, it did not make any visual change, and that is only expected because, “Amount” and “scaled_amount” features correlation equals to one.

We will see a few more scaling algorithms and see whether we can use it or not.

From the next algorithm onwards, we won't be plotting the correlation heat map. At the end of the scaling section, we will see a single correlation heat map and analyze it.

Robust Scaling

Robust Scaling is similar to standard scaling, but it removes the median and scales the data points according to the IQR (Interquartile Range).

[56]:
```
from sklearn.preprocessing import RobustScaler
rob_scaler = RobustScaler()
df['robust_amount']=rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['robust_time']=rob_scaler.fit_transform(df['Time'].values.reshape(-1,1))
print("Robust Amount")
print(df['robust_amount'])
print("")
print("Time of Robust Amount")
print(df['robust_time'])
```

```
Robust Amount
0         1.765449
1        -0.274962
2         4.946119
3         1.402722
4         0.659631
            ...
284802   -0.301625
284803    0.031940
284804    0.630329
284805   -0.173448
284806    2.701153
Name: robust_amount, Length: 282982, dtype: float64

Time of Robust Amount
0        -0.995242
1        -0.995242
2        -0.995230
3        -0.995230
4        -0.995218
            ...
284802    1.034848
284803    1.034860
284804    1.034871
284805    1.034871
284806    1.034918
Name: robust_time, Length: 282982, dtype: float64
```

Fig: Robust scaling.

Robust scaling helps to handle the outliers better than standard scaling. We will also check the distribution and compare it with Standard Scalar.

```
[57]: compare_kde("robust_amount","robust_time","Robust Scaler")
```

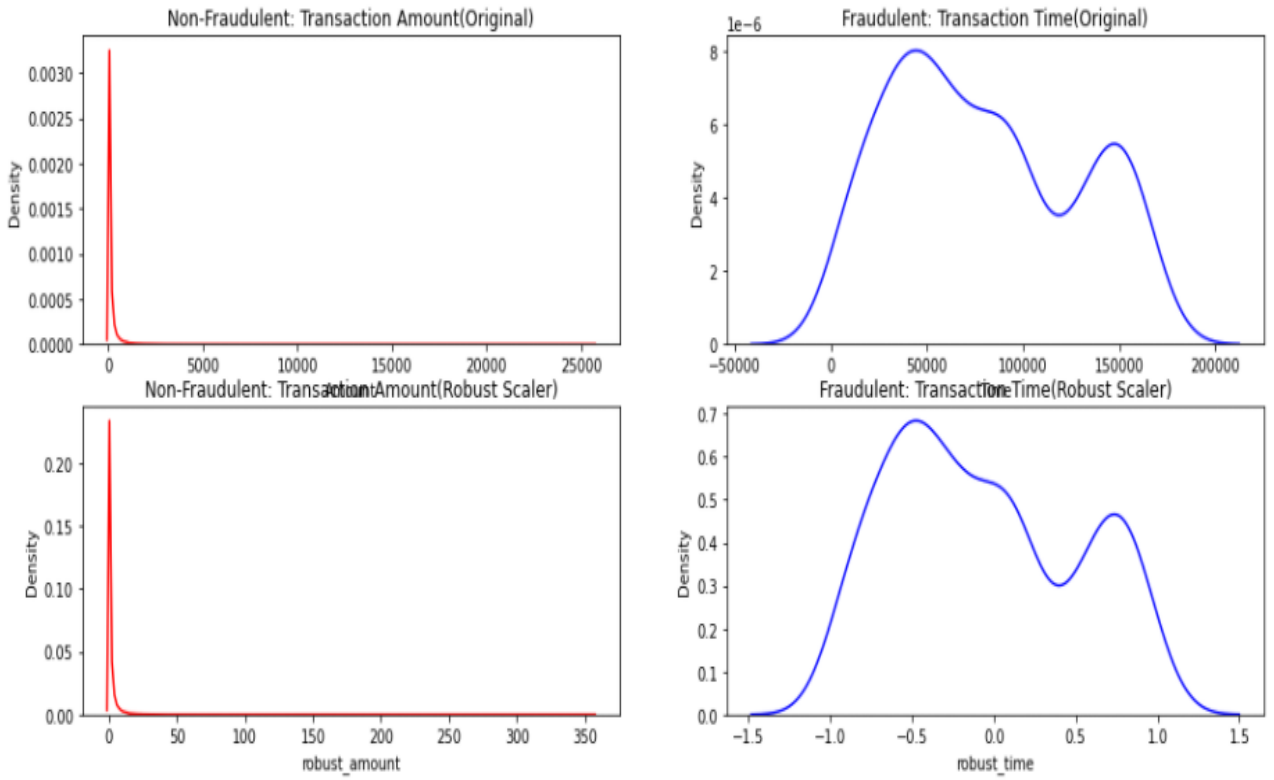


Fig: Comparing Robust scaling.

It didn't change that distribution, but the range and the scale are changed. Compared with both the Original and Standard Scalar, it quite similar.

Power Transformer

Power transformer belongs to a family of parametric, monotonic transformations that target to map/project data points from any distribution. This process helps to stabilize the variance of the data and minimize skewness. This method has the highest effects on skewed data.

As a power transformer is based on monotonic transformation, so it preserves the rank of values among the features.

```
[58]: from sklearn.preprocessing import PowerTransformer
      p_trans = PowerTransformer()
      df['ptrans_amount']=p_trans.fit_transform(df['Amount'].values.reshape(-1,1))
      df['ptrans_time']=p_trans.fit_transform(df['Time'].values.reshape(-1,1))
      print("PTrans Amount")
      print(df['ptrans_amount'])
      print("")
      print("Time of PTrans Amount")
      print(df['ptrans_time'])
```

Fig: Code for power transformer.

```
PTrans Amount
0         1.116503
1        -1.154352
2         1.620710
3         1.009746
4         0.688945
             ...
284802   -1.655577
284803    0.087773
284804    0.671461
284805   -0.441934
284806    1.320911
Name: ptrans_amount, Length: 282982, dtype: float64

Time of PTrans Amount
0        -2.436283
1        -2.436283
2        -2.436024
3        -2.436024
4        -2.435794
             ...
284802    1.534841
284803    1.534859
284804    1.534876
284805    1.534876
284806    1.534947
Name: ptrans_time, Length: 282982, dtype: float64
```

Fig: Output data of power transformer.

Similarly , like before we will use a different feature to store the scaled values. It will later help us to compare all the scaled features and come to some conclusion.

Let us see the KDE for the scaled feature and can assume that this plot will be interesting as “Amount” was highly skewed data. But for “Time,” the change won’t be significant.

Power transformer uses the algorithm Yeo-Johnson transform to scale the values with the below law:

$$y_i^{(\lambda)} = \begin{cases} ((y_i+1)^{\lambda}-1)/\lambda, & if \lambda \neq 0, y \geq 0 \\ \log(y_i+1), & if\ \lambda = 0, y \geq 0 \\ -\dfrac{[(-y+1)^{(2-\lambda)}-1]}{2-\lambda}, & if \lambda \neq 2, y < 0 \\ -\log(-y_i+1), & if\ \lambda = 2, y < 0 \end{cases}$$

Where the Yeo-Johnson transformation allows zero and negative values of y. λ can be any real number, and $\lambda = 1$ produces the identity transformation.

Now let us apply the above concept with the dataset and see it how it looks.

```
[59]: compare_kde("ptrans_amount","ptrans_time","Power Transformer")
```

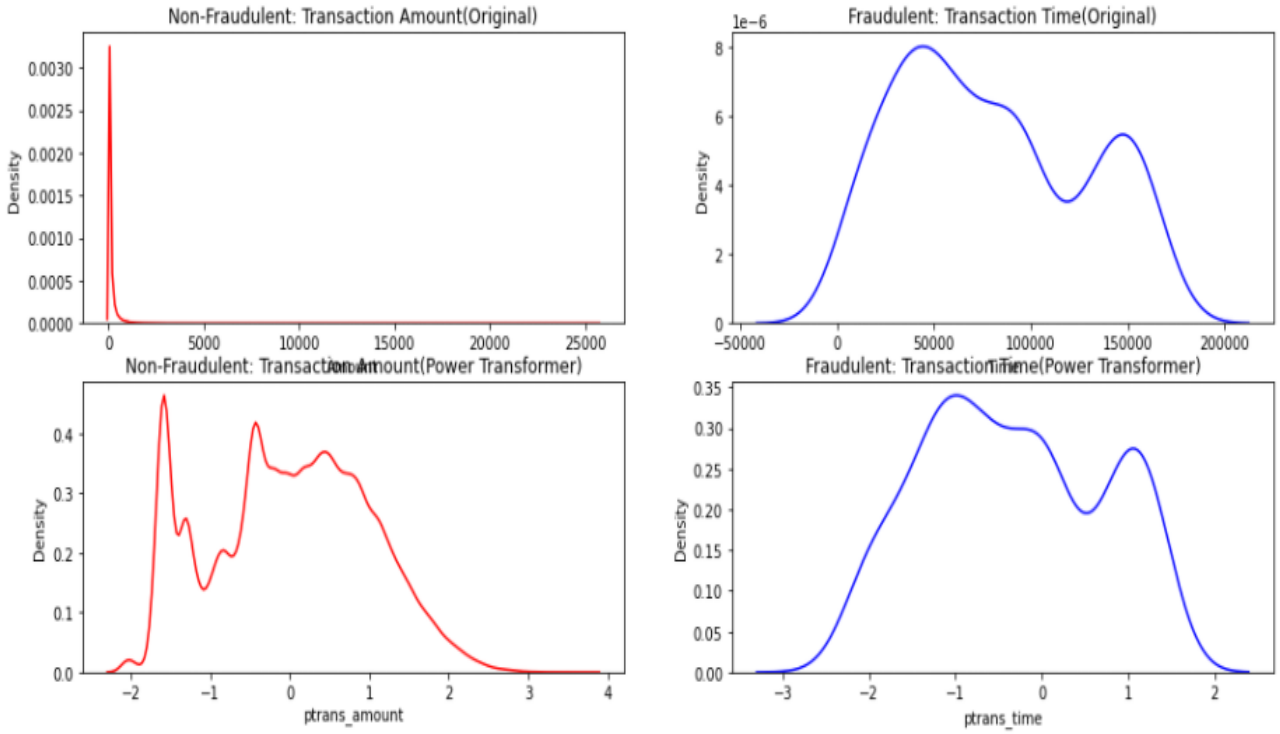


Fig: Comparing power transformer.

“Amount” feature had quite a lot of change, and it is not similar to a normal distribution but, when compared to “Time” distribution, the change is a lot.

If we again scale the scaled feature , then this will transform into a normal distribution. But it won’t make sense to scale twice.

Quantile Transformer

Quantile transformer is a non-parametric method to transform the features such that it follows a uniform or a normal distribution. Therefore, for a given feature, this transformation trends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is , therefore, a robust pre-processing scheme.

Even quantile transformer is based on monotonic transformation, so it preserves the rank of values among the feature.

[60]:
```
from sklearn.preprocessing import QuantileTransformer
q_trans = QuantileTransformer(output_distribution="normal")
df['qtransn_amount']=q_trans.fit_transform(df['Amount'].values.reshape(-1,1))
df['qtransn_time']=q_trans.fit_transform(df['Time'].values.reshape(-1,1))
print("QTransN Amount")
print(df['qtransn_amount'])
print("")
print("Time of QTransN Amount")
print(df['qtransn_time'])
q_trans=QuantileTransformer(output_distribution="uniform")
df['qtransu_amount']=q_trans.fit_transform(df['Amount'].values.reshape(-1,1))
df['qtransu_time']=q_trans.fit_transform(df['Time'].values.reshape(-1,1))
print("")
print("QTransU Amount")
print(df['qtransu_amount'])
print("")
print("Time of QTransU Amount")
print(df['qtransu_time'])
```

```
QTransN Amount
0          1.065950
1         -0.918498
2          1.654945
3          0.955155
4          0.607463
             ...
284802    -2.023292
284803     0.035525
284804     0.587370
284805    -0.410169
284806     1.312065
Name: qtransn_amount, Length: 282982, dtype: float64

Time of QTransN Amount
0         -5.199338
1         -5.199338
2         -4.408345
3         -4.408345
4         -4.255774
             ...
284802     4.334274
284803     4.484395
284804     5.199338
284805     5.199338
284806     5.199338
Name: qtransn_time, Length: 282982, dtype: float64


QTransU Amount
0            0.857955
1            0.179680
2            0.951898
3            0.831469
4            0.729229
                ...
284802       0.022022
284803       0.515092
284804       0.722580
284805       0.343343
284806       0.904816
Name: qtransu_amount, Length: 282982, dtype: float64

Time of QTransU Amount
0            0.000000
1            0.000000
2            0.000000
3            0.000000
4            0.000005
                ...
284802       1.000000
284803       1.000000
284804       1.000000
284805       1.000000
284806       1.000000
Name: qtransu_time, Length: 282982, dtype: float64
```

Fig: Quantile transformer.

Here we will be performing for both uniform and normal distribution for both the features “Time” and “Amount”.

We will start with transforming to a normal distribution and see how it works out.

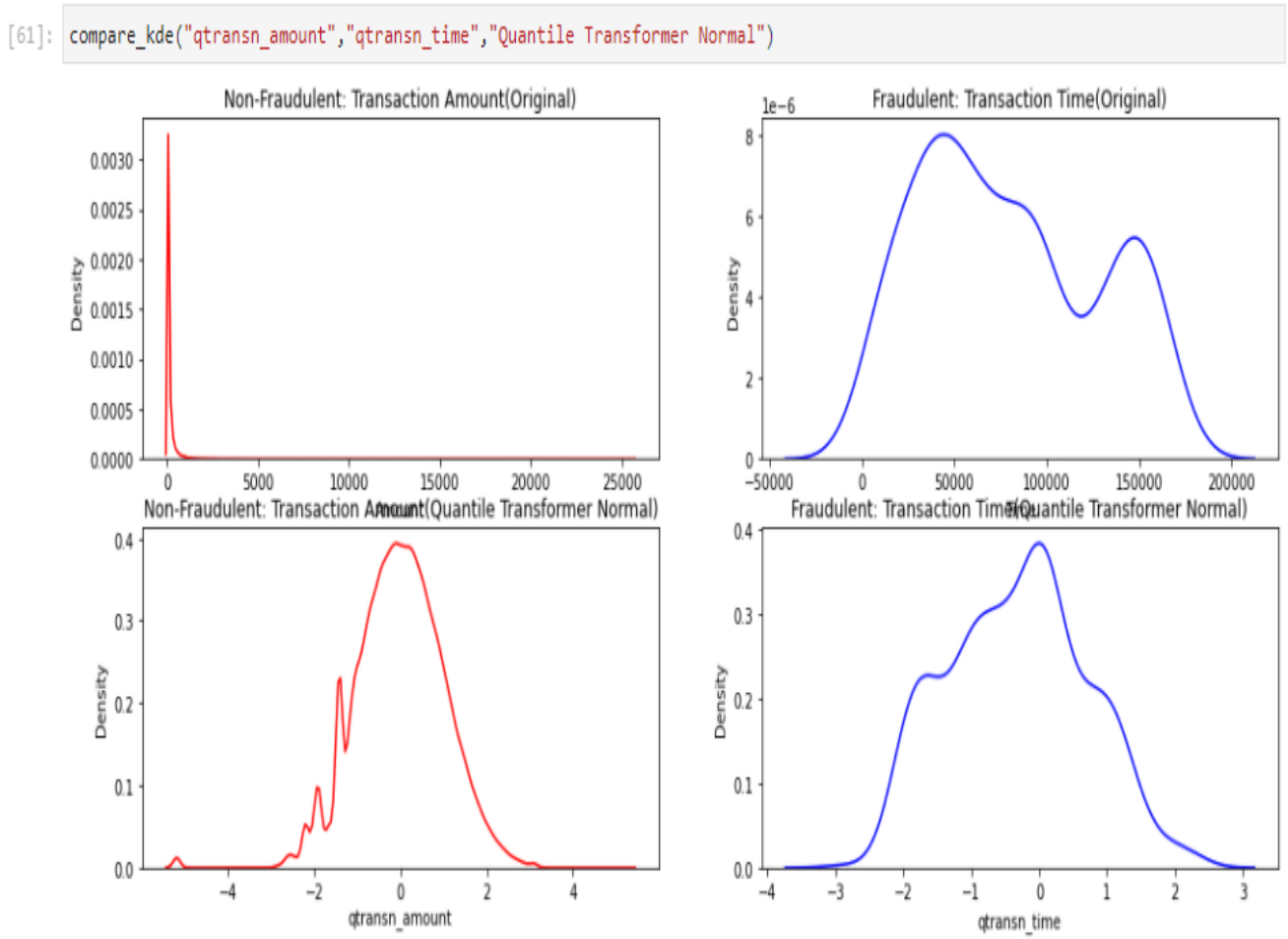


Fig: Comparing Quantile Transformer- Normal.

The above distribution have a massive change from the skewed distribution, and now the “Amount” looks similar to a Gaussian distribution, it is the same case for “Time” as well.

Similarly, we will see for uniform distribution that the scaled value will resemble a uniform distribution.

Let us plot and confirm the above hypothesis.

```
[62]: compare_kde("qtransu_amount","qtransu_time","Quantile Transformer Uniform")
```

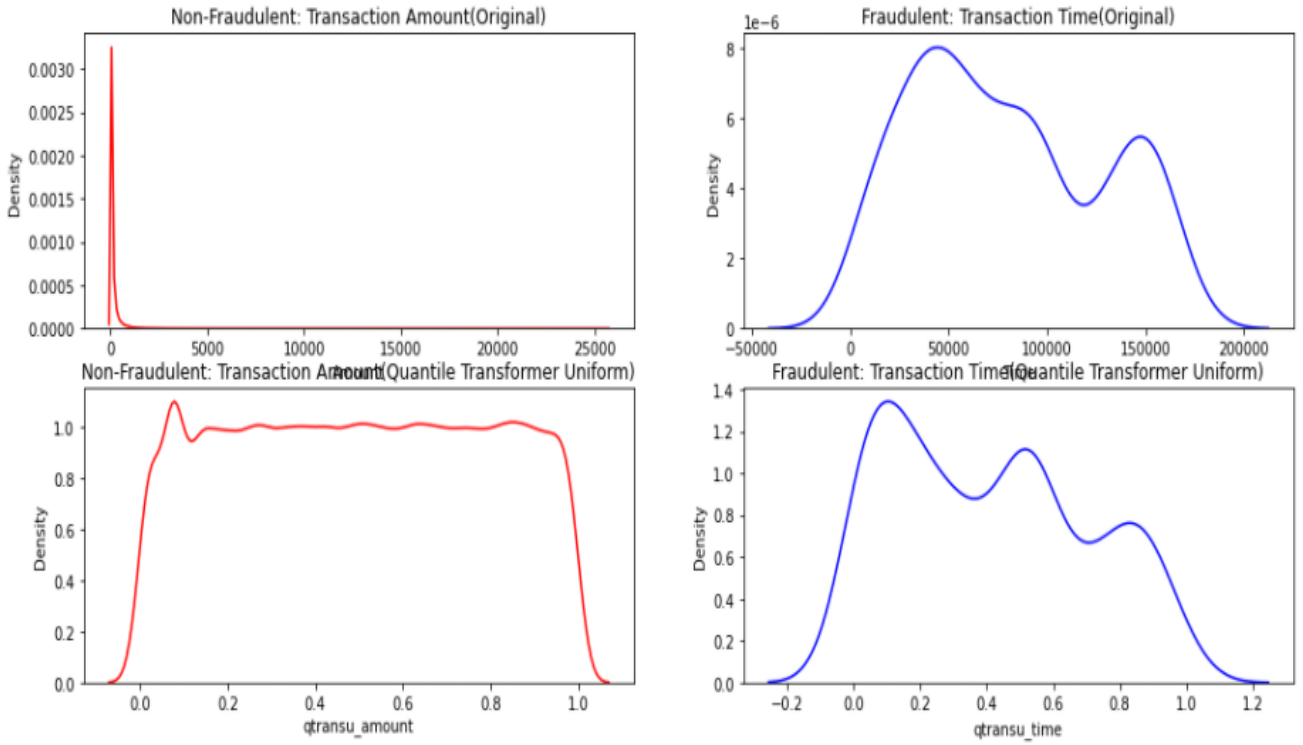


Fig: Comparing Quantile Transformer-Uniform.

We can see that “Amount” is now a uniform distribution and “Time” is close to a uniform distribution.

Seeing the distributions alone won’t make any sense until and unless we see the effect. We can plot the correlation heat map for all the scaled features which we have done so far and see which type of scaling serves the best.

```
[63]: plt.figure(figsize=(12,8))
      sns.heatmap(df.corr(),cmap='coolwarm_r',annot=False)
```

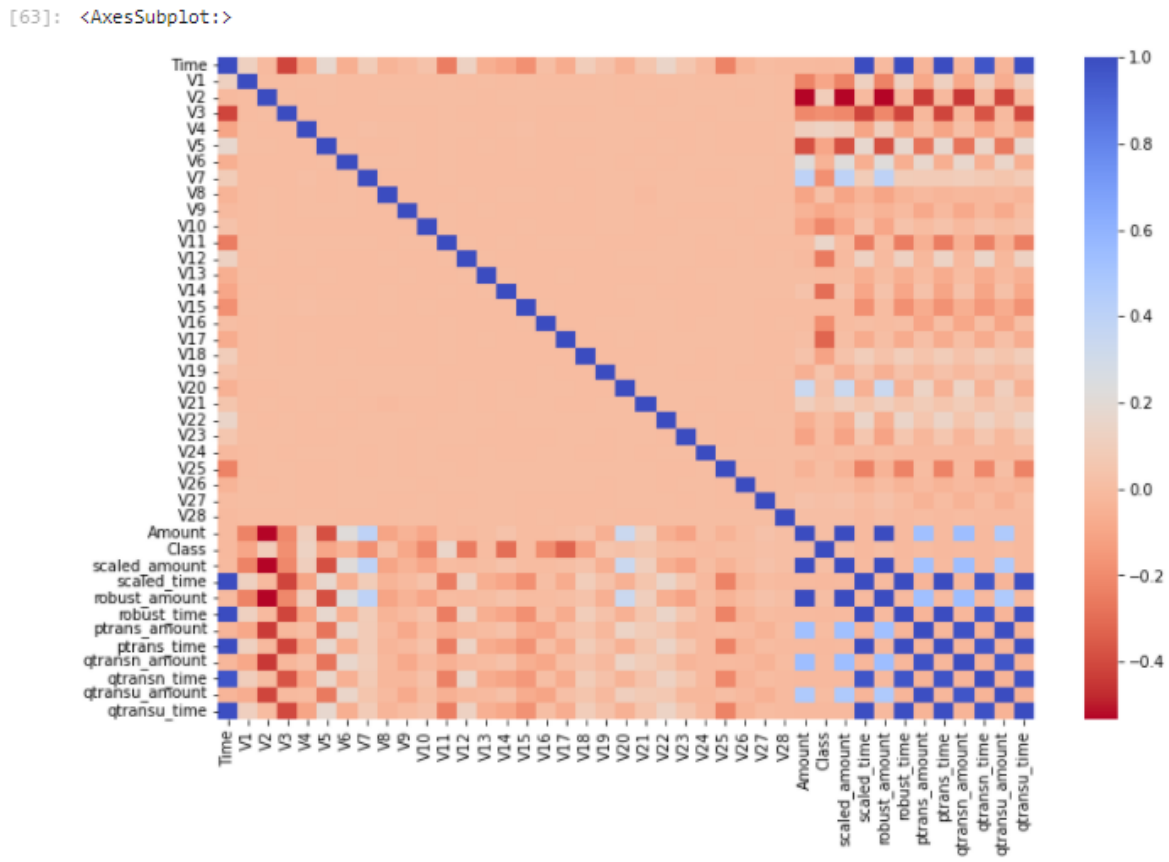


Fig: Correlation heat map after applying all scaling algorithms.

Now, we need to analyze this correlation matrix and figure out the available scaled feature.

It is 100% possible that we choose one scaling algorithm/technique for one feature and another scaling algorithm/technique for a different feature.

We have one observation that none of the dimensionally reduced features changed its correlation with respect to “Time” and “Amount”. There can be one reason why this is happening

i.e. due to imbalance. Next section, we will see how to tackle the imbalance nature of the dataset and again visualize the correlation. Then we can see there will be a significant change in the correlation, and it will be easier to choose the best features that will have a contribution to the decision we make.

Analyzing this correlation heat map is very tough as we are not V1…, V28 into consideration as of now. So, it is better to plot all the scaled amounts and time separately to analyze them.

```
[64]: plt.figure(figsize=(12,10))
      sns.heatmap(df[["ptrans_amount","qtransn_amount","qtransu_amount","robust_amount",
                      "scaled_amount","Amount","Class","Time","scaled_time","robust_time",
                     "qtransu_time","qtransn_time","ptrans_time"]].corr(),cmap='coolwarm_r',annot=True)
```

Fig: Code for plotting only scaled features.

The above code snippet will generate the below image. We will the simplified and reduced version of the correlation matrix like the below image.

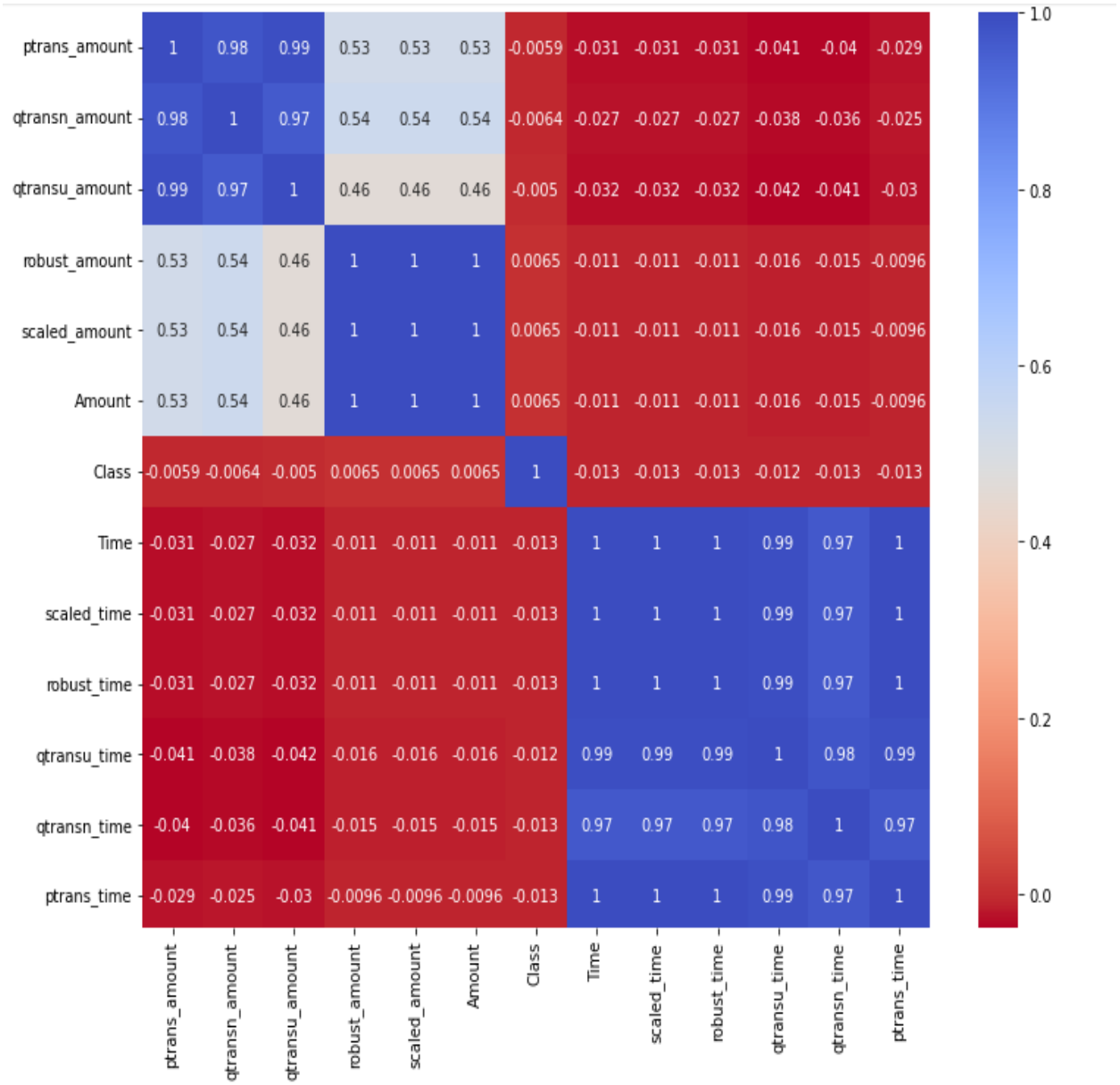


Fig: Correlation for all scaled features.

We need to compare “Time,” “Class,” and “Amount” and see an interesting result that has nearly zero change in correlation concerning “Class.” But some changes can be seen for statistical mean, standard deviation, and other features, as shown in the below diagram.

```
[65]: df[["Amount","scaled_amount","robust_amount","qtransu_amount","qtransn_amount","ptrans_amount"]].describe().T
```

[65]:

	count	mean	std	min	25%	50%	75%	max
Amount	282982.0	8.891940e+01	250.824374	0.010000	5.990000	22.490000	78.000000	25691.160000
scaled_amount	282982.0	2.892570e-17	1.000002	-0.354469	-0.330628	-0.264845	-0.043534	102.072560
robust_amount	282982.0	9.225024e-01	3.483188	-0.312179	-0.229135	0.000000	0.770865	356.459797
qtransu_amount	282982.0	4.998445e-01	0.288814	0.000000	0.251251	0.500424	0.749750	1.000000
qtransn_amount	282982.0	-9.911602e-03	1.020451	-5.199338	-0.676854	-0.003992	0.672779	5.199338
ptrans_amount	282982.0	2.466719e-16	1.000002	-2.050775	-0.733013	0.030784	0.750683	3.649005

Fig: Description of scaled "Amount".

The above diagram shows all the scaled features for "Amount". We can see some changes like

"**Standard Deviation**" : $(\sigma) = \sqrt{\frac{1}{N}\sum(x_i - \bar{\mathbf{x}})^2}$

"**Mean**":$(\bar{\mathbf{x}}) = \frac{\sum f(x)}{N}$

"Quartile": $\boldsymbol{Q_1 < Q_2 < Q_3; Q_2 = median;}$

"Inter Quartile Range(IQR)" $= \boldsymbol{Q_3 - Q_1}$;

"Quartile Deviation": $\frac{\boldsymbol{Q_3 - Q_1}}{\mathbf{2}}$

And min-max scaling.

Min-Max scaling(Rescaling)

This is one of the most common, important and simple scaling algorithms.

Here we scale the entire column with any given range into a usable given range [a,b].

$$x' = a + \frac{(x_i - \min(x))(b - a)}{\max(x) - \min(x)}$$

The above equation will iterate through all the elements and scale with the columns/list's min, max values.

Generally, we scale the list/series into the range[0,1].

Therefore, a=0 and b=1, replacing them will give us the below equation.

$$x' = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

When the min-max feature is generally used, the above equation is referred to with range [0,1].

But there can be some cases where we need to scale down to a given range.

Note: Scaling down using min-max feature scaling, won't change the distribution of the feature. It only changes the range and keeping the distribution the same for the feature.

Mean normalization

We have another variety of scaling i.e. **Mean normalization,** and the mathematical formula for it is:

$$x' = \frac{x_i - \bar{\mathrm{x}}}{\max(x) - \min(x)}$$

There can be situations where this can be used but, generally, it is used less compared to the other scaling techniques.

Next, we will see similar changes to the feature “Time.”

```
[66]: df[["Time","scaled_time","robust_time",
          "qtransu_time","qtransn_time","ptrans_time"]].describe().T
```

[66]:

	count	mean	std	min	25%	50%	75%	max
Time	282982.0	9.484896e+04	47482.459589	0.000000	54251.250000	84707.500000	139363.750000	172792.000000
scaled_time	282982.0	-1.349866e-16	1.000002	-1.997561	-0.855006	-0.213584	0.937501	1.641515
robust_time	282982.0	1.191536e-01	0.557879	-0.995242	-0.357835	0.000000	0.642165	1.034918
qtransu_time	282982.0	4.997672e-01	0.288296	0.000000	0.250490	0.499717	0.747851	1.000000
qtransn_time	282982.0	-6.691771e-04	1.001008	-5.199338	-0.677714	-0.000041	0.678906	5.199338
ptrans_time	282982.0	-1.928380e-16	1.000002	-2.436283	-0.809483	-0.143207	0.928790	1.534947

Fig: Description of scaled “Time”.

“Time” also reflects the same characteristics as “Amount,” but the change is drastic for some algorithm for the future “Amount.”

Splitting dataset

Before moving forward with handling the imbalance, we need to split the data for many reasons. One of the major reasons is for testing the model. We do not want to test the model with oversampled data because that may lead to the wrong result.

We will use train_test_split function to split the dataset into a test and train split dataset.

```
[67]: from sklearn.model_selection import train_test_split
import numpy as np
print('No Frauds',round(df['Class'].value_counts()[0]/len(df)*100,2),'% of the dataset')
print('No Frauds',round(df['Class'].value_counts()[1]/len(df)*100,2),'% of the dataset')
X= df.drop('Class',axis=1)
y= df['Class']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=2)

#See if both the train and test label distribution are similarly distributed
train_unique_label, train_counts_label = np.unique(y_train,return_counts=True)
test_unique_label, test_counts_label = np.unique(y_test,return_counts=True)

print("\n Label Distribution \n")
print(train_counts_label/len(y_train))
print(test_counts_label/len(y_test))
print("\n Train:")
print('No Frauds',round(len(y_train[y_train==0])/len(X_train)*100,2),'% of the dataset')
print('Frauds',round(len(y_train[y_train==1])/len(X_train)*100,2),'% of the dataset')
print("\n Test:")
print('No Frauds',round(len(y_test[y_test==0])/len(X_test)*100,2),'% of the dataset')
print('Frauds',round(len(y_test[y_test==1])/len(X_test)*100,2),'% of the dataset')
```

Fig: Splitting dataset.

We are trying to split the dataset then printing the percentage of fraudulent and non-fraudulent transactions for both train and test datasets.

```
No Frauds 99.84 % of the dataset
No Frauds 0.16 % of the dataset

 Label Distribution

[0.99830377 0.00169623]
[0.99856883 0.00143117]

 Train:
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset

 Test:
No Frauds 99.86 % of the dataset
Frauds 0.14 % of the dataset
```

Fig: Splitting Data Statistics.

We can see a few things that are the percentage of fraud training data is 0.17%, and the percentage of fraud test data is 0.14% but, the original dataset has only 0.16% of data.

We need to make the fraud percentage the same or similar to the original fraud percentage. As we are dealing with an extremely small percentage of fraud data, our target should be the same.

This will help us to make the dataset distribution similar to the original one. We have to always keep in mind that the total number of fraud class is too less.

There is a solution to this problem that is stratified K-fold, it is a similar concept like a cross-validation K-fold technique. It gives us the same distribution as the original distribution.

Cross-validation

Cross-validation is a technique by which we can assess how it will perform in practice, i.e., in real time data. Cross-validation is a model validation technique to ensure its performance, but it only works best when the training dataset distribution is similar to real-time data distribution.

There are different types of cross-validation techniques:

- Exhaustive cross-validation
  - Leave-p-out cross-validation
  - Leave-one-out cross-validation

- Non-exhaustive cross-validation
  - K-fold cross-validation
  - Holdout method
  - Repeated random sub-sampling validation.

We will look at the k-fold cross-validation technique as that is one of the popular validation techniques.

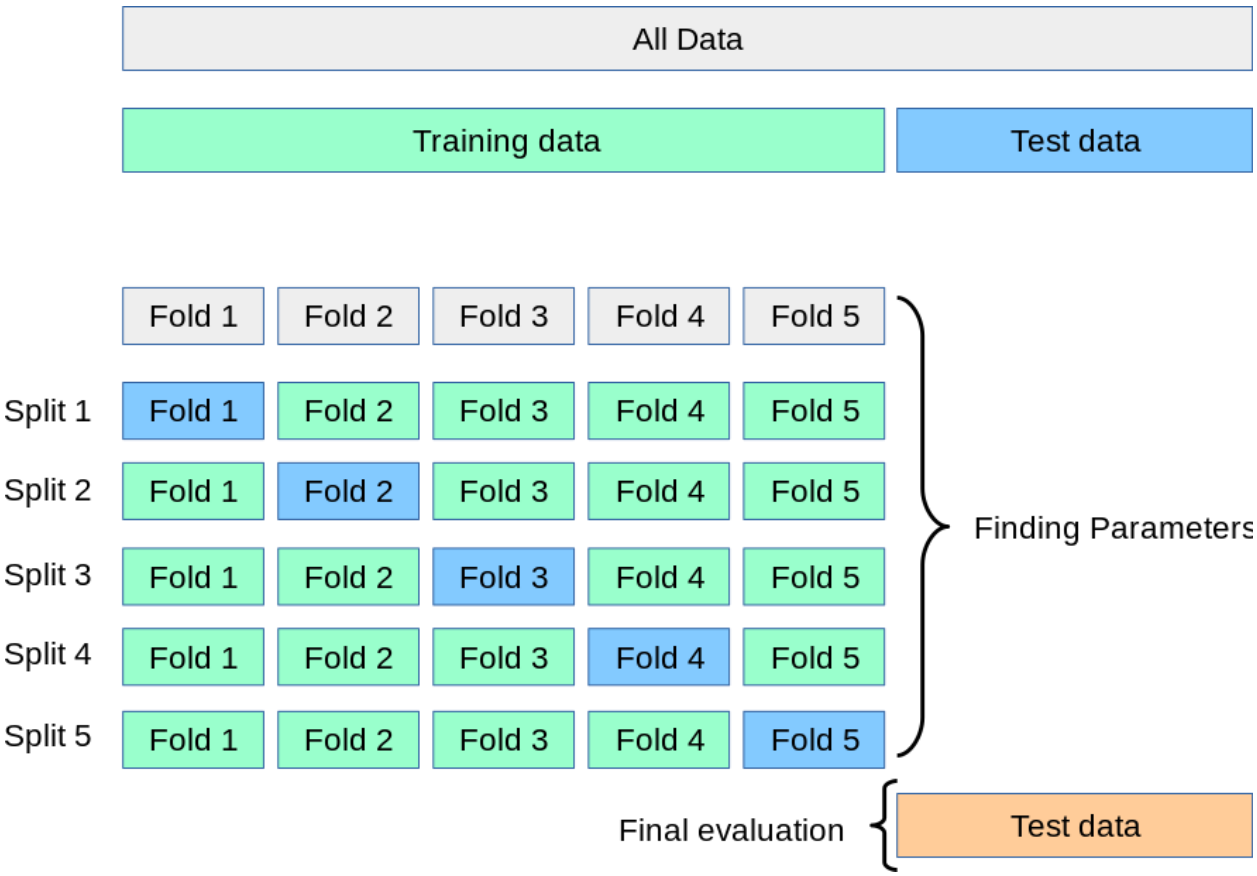


Fig: K-fold cross-validation.

The above diagram shows the image for k-fold cross validation where k = 5.

We can see from the above image that we divide the training data into k folds or k sections. Then each section is selected once as a validation dataset, and rest sections are used for training. Due to the folding concepts , the models are less prone to overfitting.

The goal of cross- validation techniques help the model to solve problems like overfitting or selection bias and gives an insight

into how the model will generalize to an independent/unknown dataset.

```
[68]: from sklearn.model_selection import StratifiedKFold
      import numpy as np
      print('No Frauds',round(df['Class'].value_counts()[0]/len(df)*100,2),'% of the dataset')
      print('No Frauds',round(df['Class'].value_counts()[1]/len(df)*100,2),'% of the dataset')
      X= df.drop('Class',axis=1)
      y= df['Class']

      skf = StratifiedKFold(n_splits=10, random_state=None, shuffle=False)

      for train_index, test_index in skf.split(X,y):
          X_train, X_test = X.iloc[train_index],X.iloc[test_index]
          y_train, y_test = y.iloc[train_index],y.iloc[test_index]

      #See if both the train and test label distribution are similarly distributed
      train_unique_label, train_counts_label = np.unique(y_train,return_counts=True)
      test_unique_label, test_counts_label = np.unique(y_test,return_counts=True)

      print("\n Label Distribution \n")
      print(train_counts_label/len(y_train))
      print(test_counts_label/len(y_test))
      print("\n Train:")
      print('No Frauds',round(len(y_train[y_train==0])/len(X_train)*100,2),'% of the dataset')
      print('Frauds',round(len(y_train[y_train==1])/len(X_train)*100,2),'% of the dataset')
      print("\n Test:")
      print('No Frauds',round(len(y_test[y_test==0])/len(X_test)*100,2),'% of the dataset')
      print('Frauds',round(len(y_test[y_test==1])/len(X_test)*100,2),'% of the dataset')
```

Fig: Splitting Dataset(StratifiedKFold).

From the above code snippet, we are expecting the training and testing dataset with the class distribution like the original one.

```
No Frauds 99.84 % of the dataset
No Frauds 0.16 % of the dataset

 Label Distribution

[0.99835875 0.00164125]
[0.99833911 0.00166089]

 Train:
No Frauds 99.84 % of the dataset
Frauds 0.16 % of the dataset

 Test:
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
```

Fig: Split Data Statistics(StratifiedKFold).

We got the expected output i.e. the percentage of the class distribution is the same for the train, test and original dataset.

Handling imbalance

Tackling imbalance dataset can be done in a few ways:

1. By oversampling the smaller dataset.
2. By under-sampling the larger dataset.
3. Mix of oversampling and under-sampling.

In oversampling, we will generally create synthetic data points for the class, which has low counts. So, for oversampling, we have to use the test data from the original data, because after performing oversampling, if we split the data that won't be correct and we will get a wrong result.

But for under-sampling, we can use the original data frame to make a sub sample out of it. But in this case we will only use the train test dataset.

What is a sub-sample?

Subsample just means a part of the original dataset, but in this scenario, the subsample will be 50-50. We will take the entire Fraudulent class and under-sample the non-fraudulent class so that the ratio is 50-50.

Advantages of sub-sample are:

1. **Overfitting** is a common problem when there is an imbalance because it assumes that in most cases there are non-fraudulent transactions. But for a subsample, this problem will be less prone to overfitting.

2. Correlation with the class will improve drastically as our subsample will have no imbalance problem. Although we don't know what the "V" features stand for(but, my assumption is "Vector"), it will be useful to understand how each of these features influences the result with the class(Fraud or No Fraud) by having an imbalance data frame we are not able to see the true correlations between the class and features.

We have seen some of the importance of sub-sampling and how it can affect some of the other aspects of the data like

analysis and model output. Now, we will take a look at a different type of resampling techniques.

What is Resampling?

Resampling is a technique to handle imbalance data by creating a sub-sampled dataset from the original data. Resampling can be done in two ways one removing samples from the majority class, i.e., also known as under sampling and adding synthetic samples to the minority class, which is also known as over-sampling.

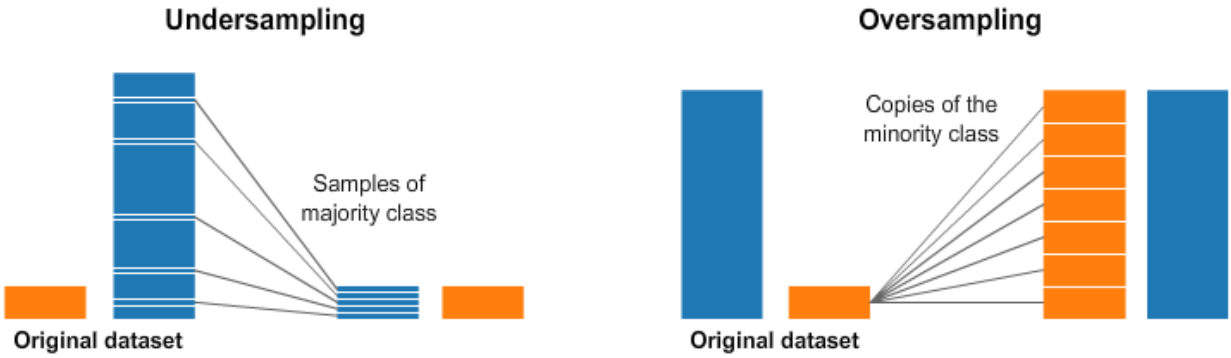


Fig: Oversampling and Under sampling.

We can see from the above illustration that how over and under sampling works. Going forward, we will see some of the classic and modern techniques to oversample and under sample the dataset.

Before that, we need to make sure that we are using only the train dataset and not using the test dataset at all. The test

dataset will only be used to validate the model and see how good it is.

```
[69]: train_df = X_train.copy()
      train_df['Class'] = y_train
      train_df.shape

[69]: (254684, 41)
```

Fig: Train data frame

We can see that; we will only use 2,54,684 records. Within that training data frame. There will be a mixture of fraud and non-fraud datasets in the ratio of the Original dataset as we have used the Stratified K-fold technique. Let's see the count of different classes.

```
[70]: train_df['Class'].value_counts()

[70]: 0    254266
      1       418
      Name: Class, dtype: int64
```

Fig: Train Dataframe Class Distribution.

We can see only have 419 records for Class 1 i.e. Fraud class. We will perform resampling on this dataset and analyze the dataset.

Before going ahead, we need to find a way to visualize the 41 column features. We can use a dimensionality reduction algorithm like Principal Component Analysis to reduce to 2 principal components and plot in a 2D plot.

Below a function will help us to plot a 2D graph.

```
[71]: def plot_2d_space(X,y,label='Class'):
          colors = ['#1F77B4' , '#FF7F0E']
          markers = ['o' , 's']
          for l,c,m in zip(np.unique(y),colors,markers):
              plt.scatter(
                  X[y==1,0],
                  X[y==1,1],
                  c=c, label=l, marker=m
              )
          plt.title(label)
          plt.legend(loc='upper right')
          plt.show()
```

Fig: Function to plot 2 component PCA

As we will plot a 2D graph, again and again, we are using a function for that. Then a simple function call is sufficient to plot it.

So, let us plot for the original train dataframe and see how it looks.

```
[56]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X = pca.fit_transform(X_train)

plot_2d_space(X, y_train, 'Imbalanced dataset (2 PCA components)')
```

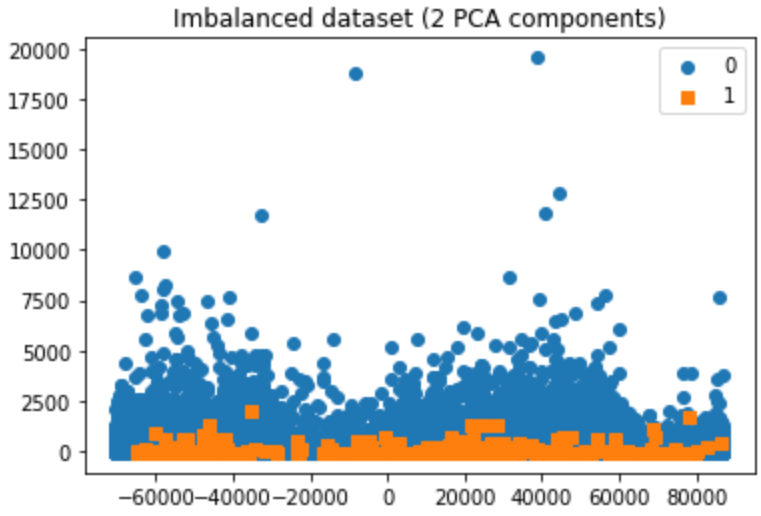


Fig: PCA plot for Original dataframe.

We can see a certain pattern of different classes of the transaction from the above image. Going forward, we will see two varieties of under-sampling and two varieties of over sampling.

Random under-sampling

Random under-sampling will remove the majority class data points such that it is equivalent/equal/proportional to the minority class. This process creates a balanced dataset, thus preventing common problems like overfitting:

In the below code snippet, we will divide the train dataframe into two classes.

```
[73]: #class count
      count_class_0 , count_class_1 = train_df.Class.value_counts()

      #Divide by class
      train_df_0 = train_df[train_df['Class']==0]
      train_df_1 = train_df[train_df['Class']==1]
```

Fig: Dividing the training dataset.

As we have divided the training dataset as per different classes, now we can perform random under-sampling or reduce the record counts for the majority class. Under-sampling can be achieved in multiple ways, but here we will only see random sampling, the next section, we will focus on another method.

```
train_df_0_under = train_df_0.sample(count_class_1)
train_df_under = pd.concat([train_df_0_under , train_df_1], axis=0)

print('Random under_sampling:')
print(train_df_under.Class.value_counts())
sns.countplot('Class', data=train_df_under)
```

```
Random under_sampling:
0    418
1    418
Name: Class, dtype: int64
```

```
c:\users\avinandan\credit card fraud detection project\virtual_env_na
ing: Pass the following variable as a keyword arg: x. From version 0.
passing other arguments without an explicit keyword will result in ar
  warnings.warn(
```

[73]: `<AxesSubplot:xlabel='Class', ylabel='count'>`

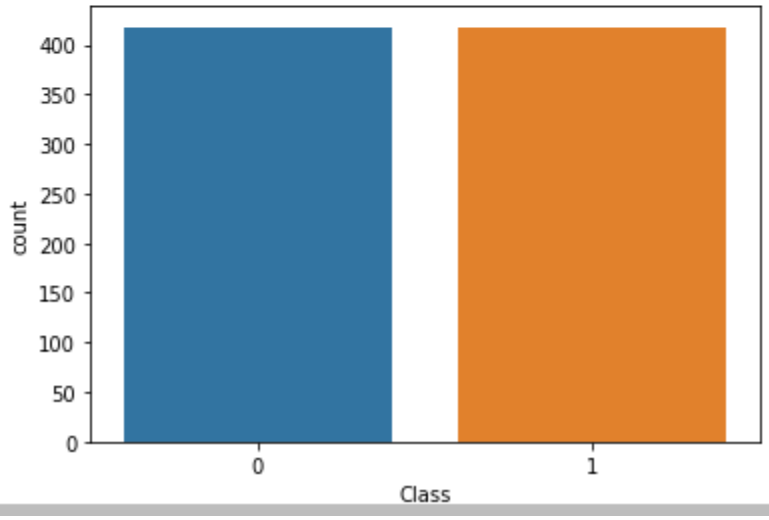


Fig: Random under sampling.

From the above illustration, we can see Class 0 , i.e. the non-fraudulent class, has the same records as a fraudulent class. Now, this dataset is equi-distributed, and it is likely to give a better correlation and model results, as we have hypothesized before.

But reducing/resampling the number of records drastically can arise multiple problems due to information loss as we are bringing 419 non-fraud transactions from 284,315 non-fraud transactions.

After completing random under sampling, we need to visualize dataset with 41 features.

With the original dataset, we have dimensionally reduced using PCA into two principal components; similarly we have to perform the same operation on this dataset such that we will be able compare them.

```
[58]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X = pca.fit_transform(train_df_under.drop(columns=["Class"]))

plot_2d_space(X, train_df_under["Class"], 'Undersampled dataset (2 PCA components)')
```

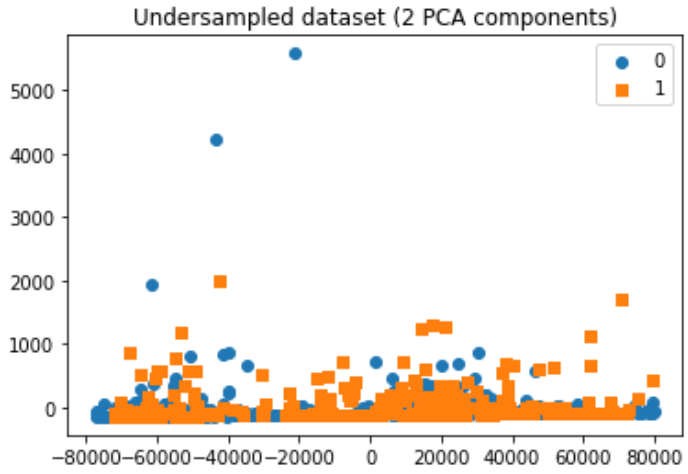


Fig: PCA plot for random under sampling.

From the above diagram, we can see homogeneity in the data points of different classes.

With this new homogeneous dataset, we are expecting a good correlation between all the features and primarily with the feature “class.” Lets us plot the correlation matrix heat map and analyze it.

```
[75]: plt.figure(figsize=(12,8))
      sns.heatmap(train_df_under.corr(),cmap='coolwarm_r',annot=False)

[75]: <AxesSubplot:>
```

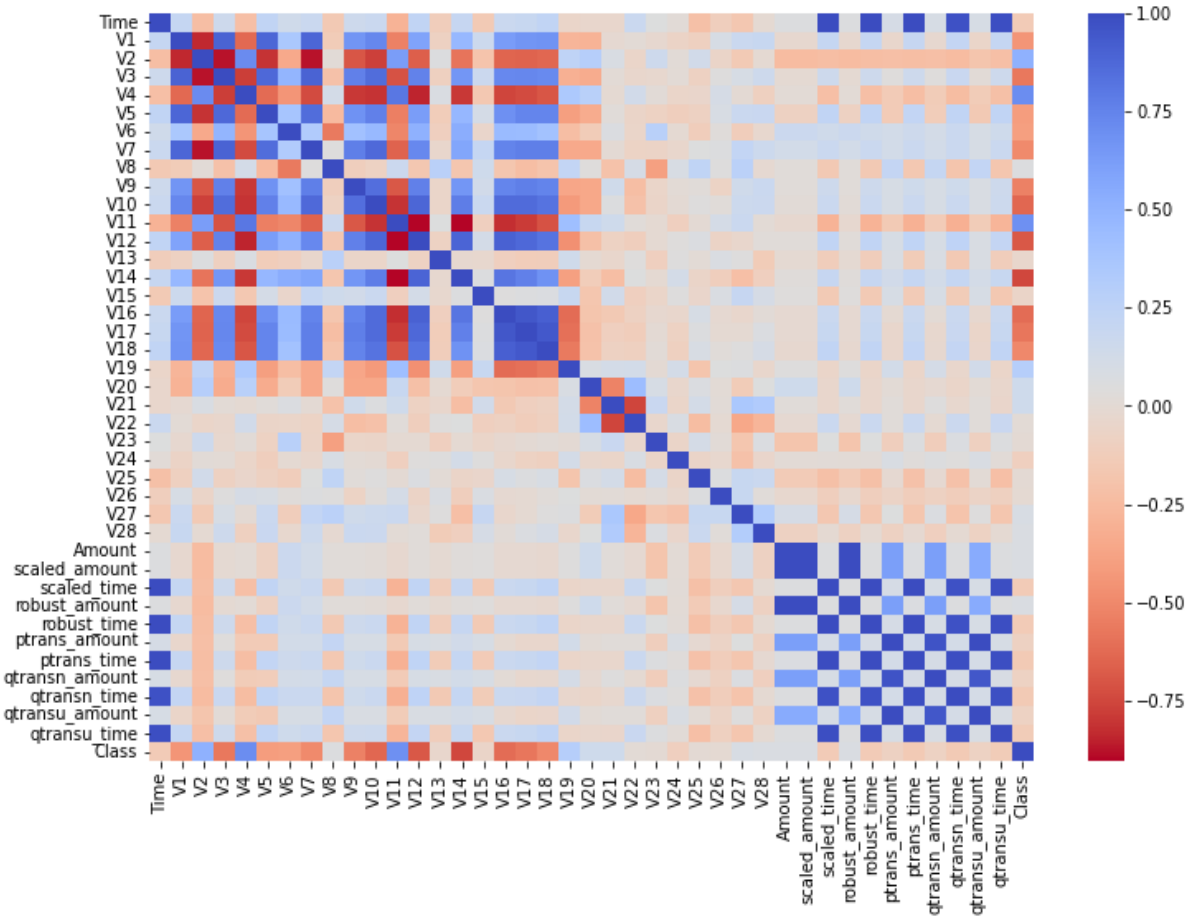


Fig: Correlation for Random Under Sampled data.

We can see a drastic with the correlation matrix for mostly all the features. Some features are now positively and negatively correlated, but due to imbalanced nature of the original dataset, the correlation matrix nearly shows zero correlation for the features V1…., V28.

If we individually see the features V1…,V28 , then there are some positively and negatively correlated features as listed below:

- Positive correlations- V2,V4,V11 and V19 are positively correlated.

- Negatively correlations- V17, V14, V12, V16 and V10 are negatively correlated.

Later we will analyze the above nine features individually:

Now, we have a list of features that will affect the output feature, i.e., “Class.” There is one more interesting observation that after random under-sampling, all the scaled features and their respective original features have the same or similar correlation.

Random over-sampling

Similar to random under-sampling, we have random over-sampling where we will oversample the minor class, i.e., in this case, it will be the fraudulent transaction dataset.

Let us see how the new dataset will look after it is oversampled.

```
[76]: train_df_1_over = train_df_1.sample(count_class_0,replace=True)
      train_df_over = pd.concat([train_df_0 , train_df_1_over], axis=0)

      print('Random over-sampling:')
      print(train_df_over.Class.value_counts())
      sns.countplot('Class', data=train_df_over)
```

```
Random over-sampling:
0    254266
1    254266
Name: Class, dtype: int64
```

```
[76]: <AxesSubplot:xlabel='Class', ylabel='count'>
```

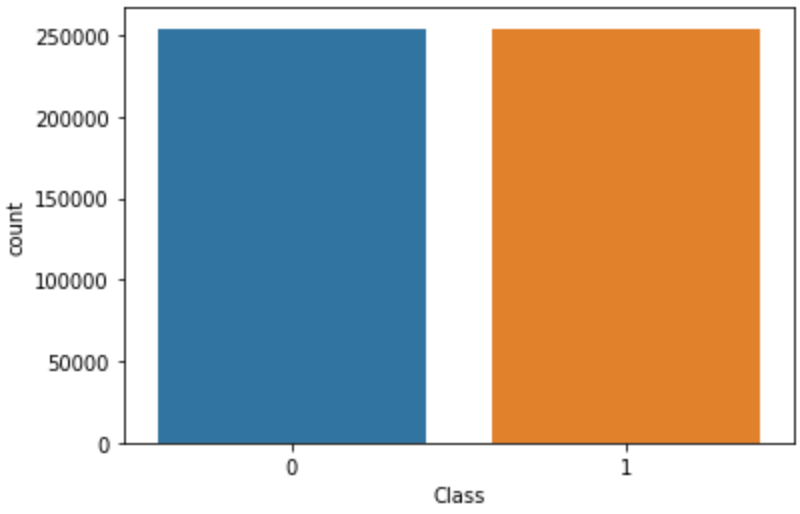


Fig: Random over-sampling.

We can see both the classes have 2,54,266 records and now the problem of imbalance data is not there. This doesn't mean the

dataset is good to use; we need to analyze further to come to this conclusion.

```
[61]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X = pca.fit_transform(train_df_over.drop(columns=["Class"]))

plot_2d_space(X, train_df_over["Class"], 'Oversampled dataset (2 PCA components)')
```

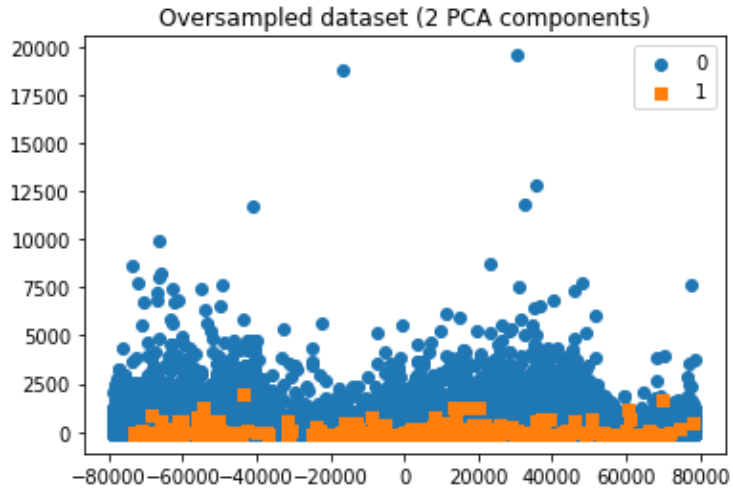


Fig: PCA plot for random over-sampling.

The Oversampled data set shows the same plot as the original dataset plot. So that it, means the correlation heat map will be similar to the original one. But unfortunately, no, it will be extremely different because this oversampled dataset is equidistributed so it will have a better correlation and will be similar to the randomly under-sampled data set.

Let us plot the correlation heat map and match it with the original correlation heat map and the under-sampled correlation heat map.

```
[78]: plt.figure(figsize=(12,8))
      sns.heatmap(train_df_over.corr(),cmap='coolwarm_r',annot=False)
```

```
[78]: <AxesSubplot:>
```

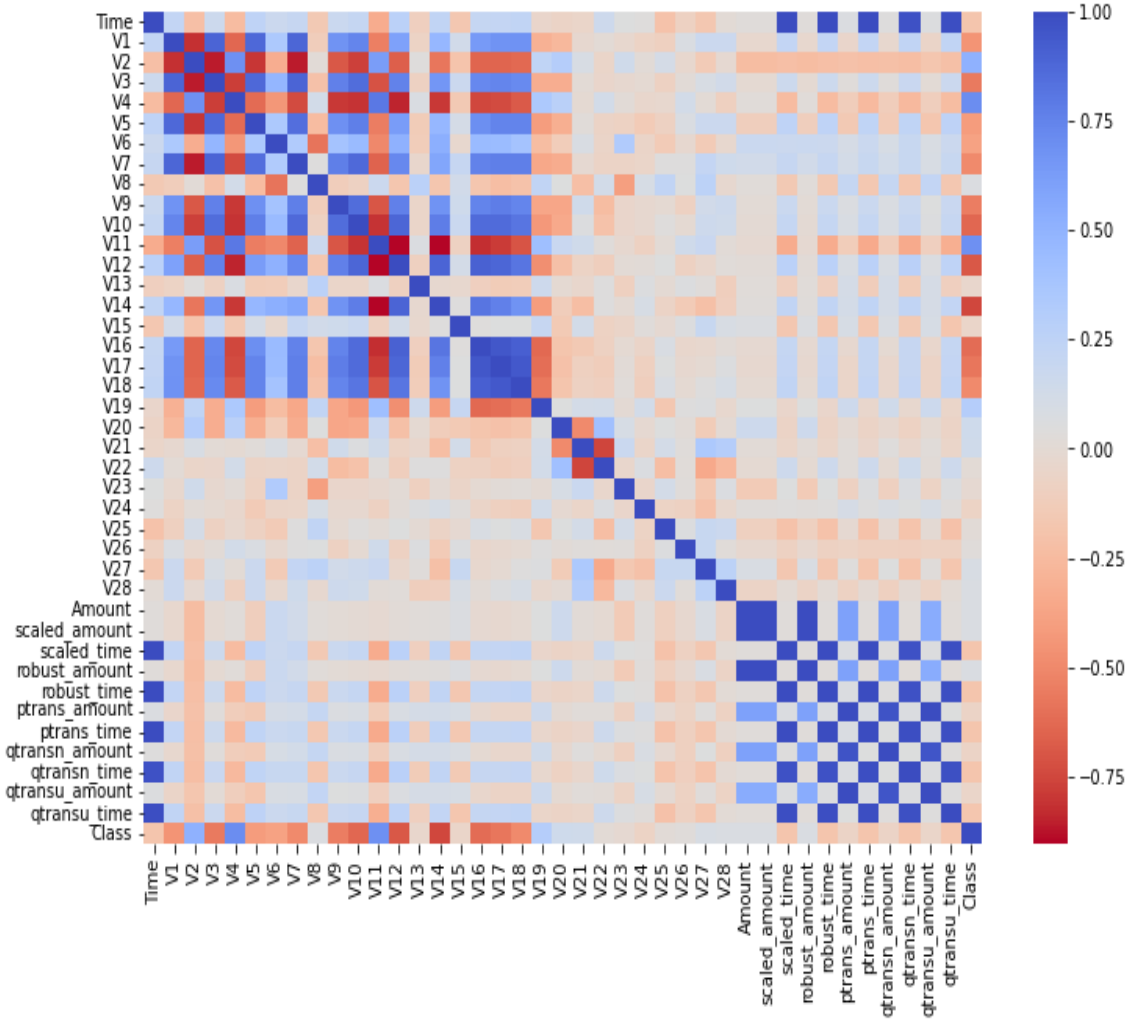


Fig: Correlation heat map for random oversampled data.

Seeing the above correlation, we can say it has no resemblance with the original heat map, but in turn, it is similar to a random under-sampled data frame. Let us see some other algorithms and compare them along the way.

Tomek's Links under sampling

This is another type of under-sampling where we generally remove Tomek's Links from the dataset.

The detailed explanation is out of scope for this book.

But we can explain/visualize the concept, in brief, using some visualization.

Note: All the steps below are not detailed out. It is just a brief workflow.

Step 1: The data points are plotted in n-dimensions(n-features) in this case for simplicity, we are using 2-axis and differentiate them with respect to class (here, there are 2 classes only).

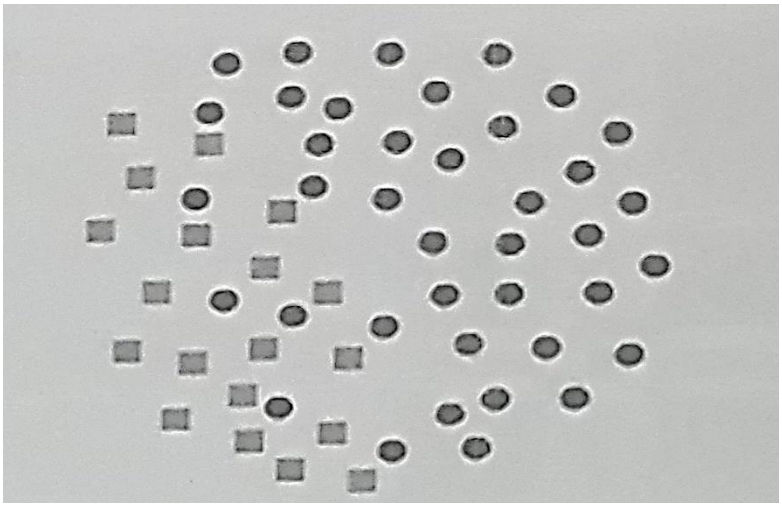


Fig: Plot Data

Step 2: We find the nearest neighbor where k=1 where the minor and the major classes are coupled.

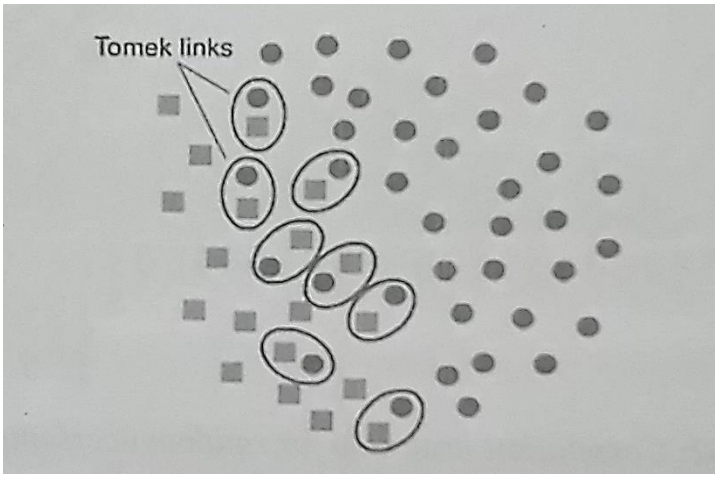


Fig: Find Tomek Links

Step 3: After we find the coupled data, we will remove the major class for under-sampling.

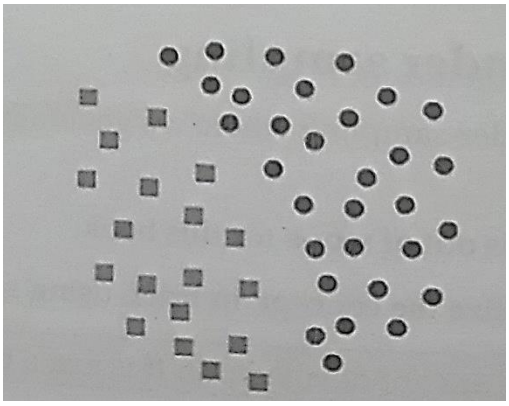


Fig: Remove the Major Class

The above illustrations give us a simple working of Tomek's Link. This helps us to define a good decision boundary and remove some of the data points from the majority class. So, this can be termed as under-sampling.

```
[63]: from imblearn.under_sampling import TomekLinks

tkl = TomekLinks(sampling_strategy='auto', n_jobs=-1)
X_tl, y_tl = tkl.fit_resample(X_train, y_train)

train_df_tkl = X_tl
train_df_tkl['Class'] = y_tl

print('Tomek links under-sampling:')
print(train_df_tkl.Class.value_counts())
sns.countplot('Class', data=train_df_tkl)
```

```
Tomek links under-sampling:
0    254204
1       418
Name: Class, dtype: int64
```

```
[63]: <AxesSubplot:xlabel='Class', ylabel='count'>
```

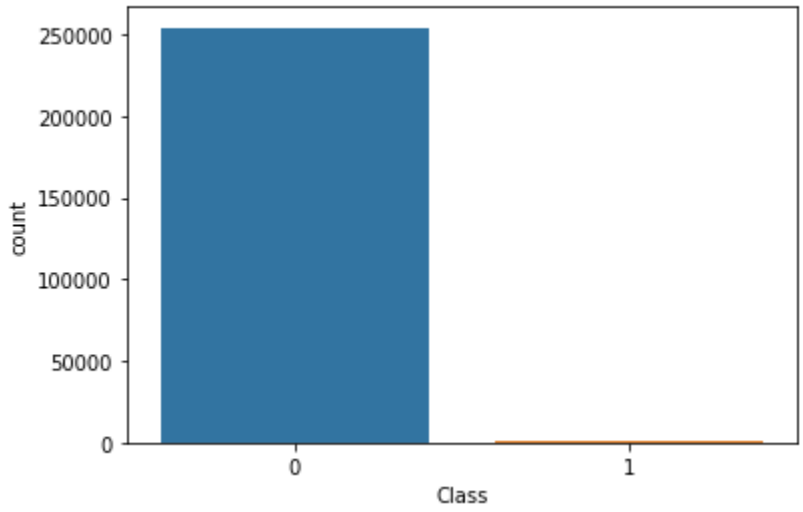


Fig: Tomek Links.

Seeing the image and the counts, we can see there is not much change. We can clearly say there were extremely a smaller number of Tomek's link.

```
train_df.shape[0] - train_df_tkl.shape[0]
```

```
62
```

Fig: Total data points removed.

We only have 62 records like that which can be removed from the dataset. We clearly state that there won't be any change from the original data as the change is not significant enough.

Lets plot the principal components and see how it looks.

```
[64]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X = pca.fit_transform(train_df_tkl.drop(columns=["Class"]))

plot_2d_space(X, train_df_tkl["Class"], 'Tomek links under-sampling (2 PCA components)')
```

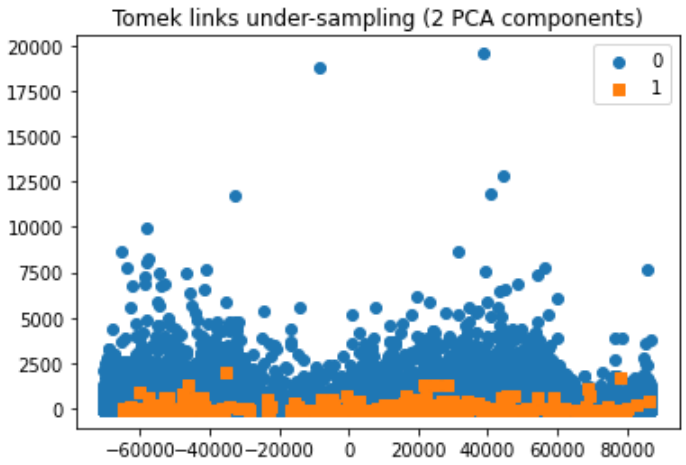


Fig: PCA plot for Tomek's Link.

We can see it is same distribution as the original data frame. So, we can say that correlation will be the same, and the distribution did not change at all.

```
[65]: plt.figure(figsize=(12, 8))
      sns.heatmap(train_df_tkl.corr(),  cmap='coolwarm_r', annot=False)
```

```
[65]: <AxesSubplot:>
```

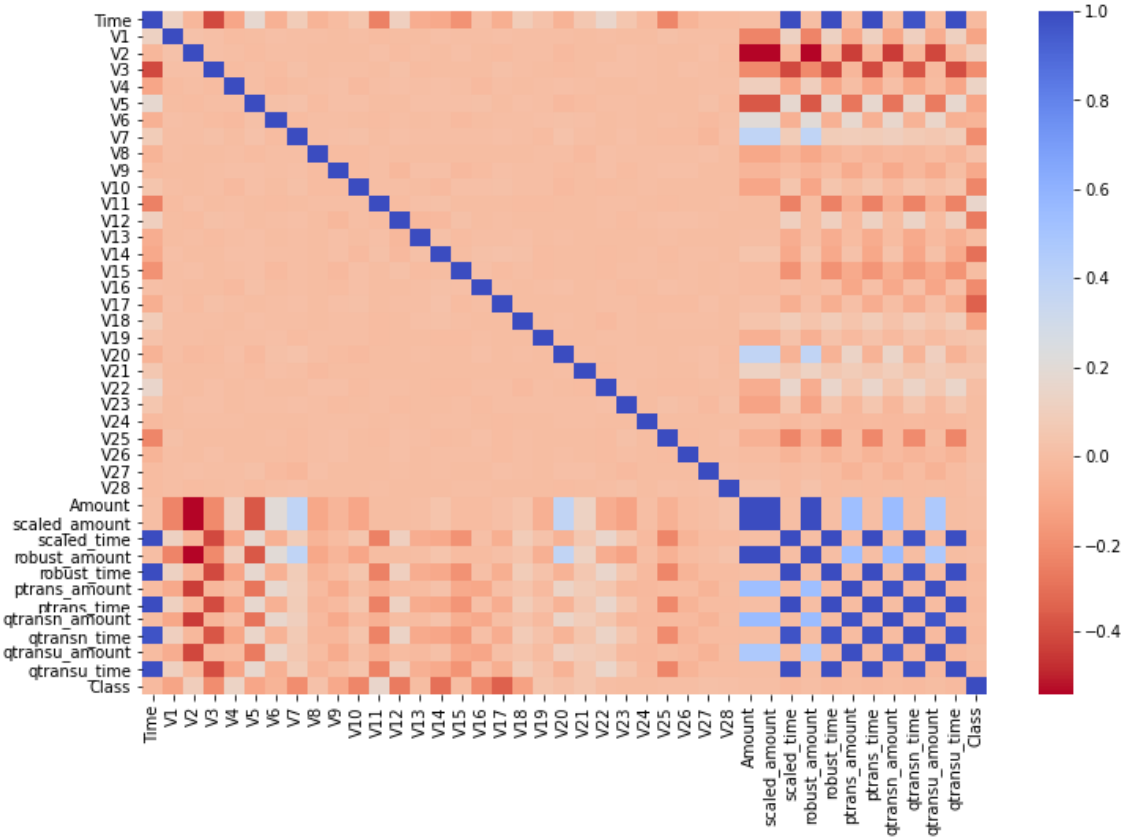


Fig: Correlation Heatmap for Tomek Links.

We can confirm that the heat map did not change at all, so this technique cannot be used for under-sampling.

In real-time, this work is highly tedious and time taking, to go through many hypotheses, and later it turns out to be wrong. This gives us experience and more insight into the dataset.

As this is a small number of Tomek's link, we can remove them so that other algorithms can work better if their algorithms have some effect due to those links.

Synthetic minority over-sampling technique

We will see one more oversampling technique **Synthetic Minority Over-Sampling(SMOTE)**. This technique oversamples the minor class using the pattern of the dataset.

To understand SMOTE, we will see how it works, and to know how this algorithm oversamples the minority class.

Step1: Let us plot the data points n-dimensional space(n-features) and classify them as per the “Class.”

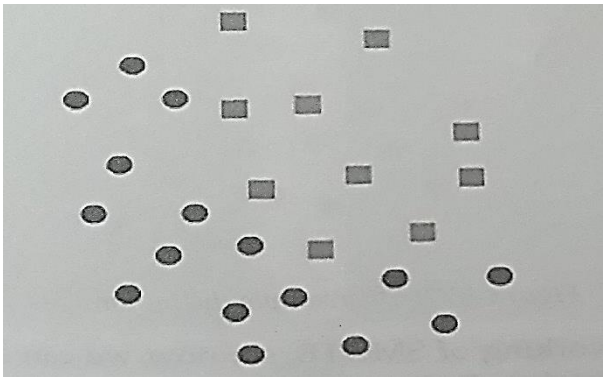


Fig: Plot Data Points.

Step2: Create a pattern/boundary of the minority dataset with the data points. We can create a space using the boundary of the minority data and add synthetic values within the boundary where the nearest neighbor is also of the same class.

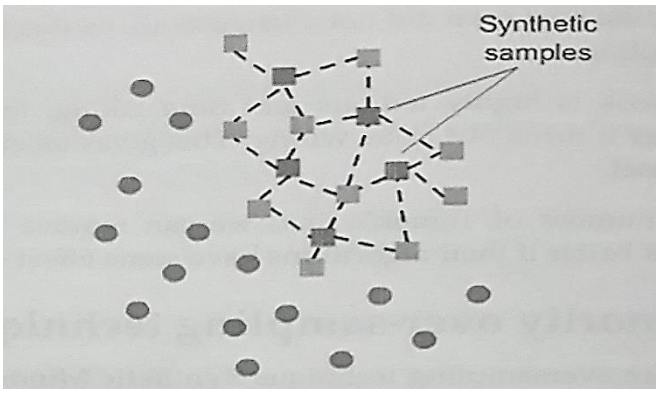


Fig: Find the relation between the minor data points.

Step 3: Plot those points and which meets step 2 conditions and assign them the minor class.

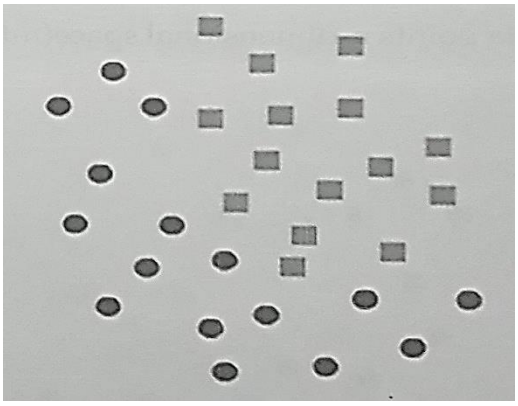


Fig: Oversample the minor class.

Implementing SMOTE, which is an oversampling algorithm, the minority class will have the same count of data as the majority class. Let us execute the below code to see how it look like.

```
[66]: from imblearn.over_sampling import SMOTE

sm = SMOTE()
X_sm, y_sm = sm.fit_resample(X_train, y_train)

train_df_sm = X_sm
train_df_sm['Class'] = y_sm

print('SMOTE over-sampling:')
print(train_df_sm.Class.value_counts())
sns.countplot('Class', data=train_df_sm)
```

```
SMOTE over-sampling:
0    254266
1    254266
Name: Class, dtype: int64
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: Fut
rg: x. From version 0.12, the only valid positional argument will be `dat
yword will result in an error or misinterpretation.
  warnings.warn(
```

```
[66]: <AxesSubplot:xlabel='Class', ylabel='count'>
```

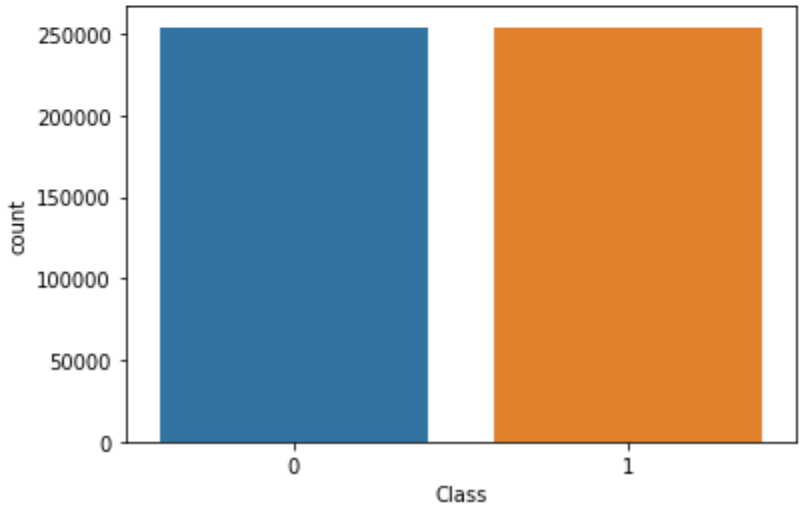


Fig: SMOTE.

We can see that it equidistributed and we have the same number of records for both the classes. But we can think that increasing the minor class with this amount can affect the

quality of the dataset. The answer is mostly; it won't affect as the synthetic data points follow the minor class pattern.

We can take a look at the plot of the data and compare it with the random oversampling data set.

```
[67]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X = pca.fit_transform(train_df_sm.drop(columns=["Class"]))

plot_2d_space(X, train_df_sm["Class"], 'SMOTE over-sampling (2 PCA components)')
```

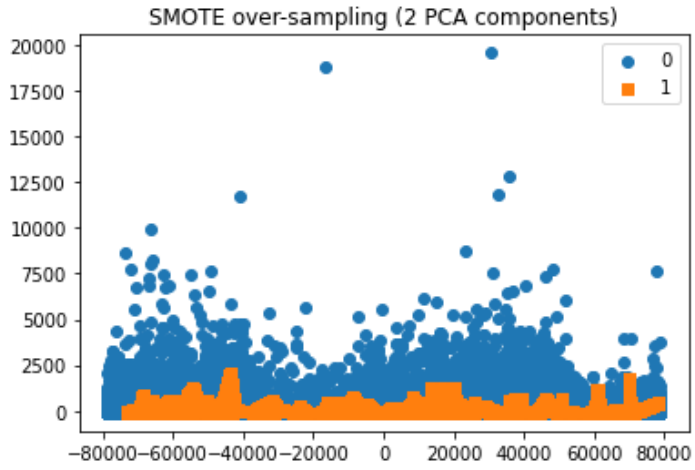


Fig: PCA plot for SMOTE.

We can see there is a difference in the distribution. But it is not significant enough from my perspective. We can test these hypotheses in the latter part of the chapter.

```
[68]: plt.figure(figsize=(12, 8))
      sns.heatmap(train_df_sm.corr(),  cmap='coolwarm_r', annot=False)
```

```
[68]: <AxesSubplot:>
```

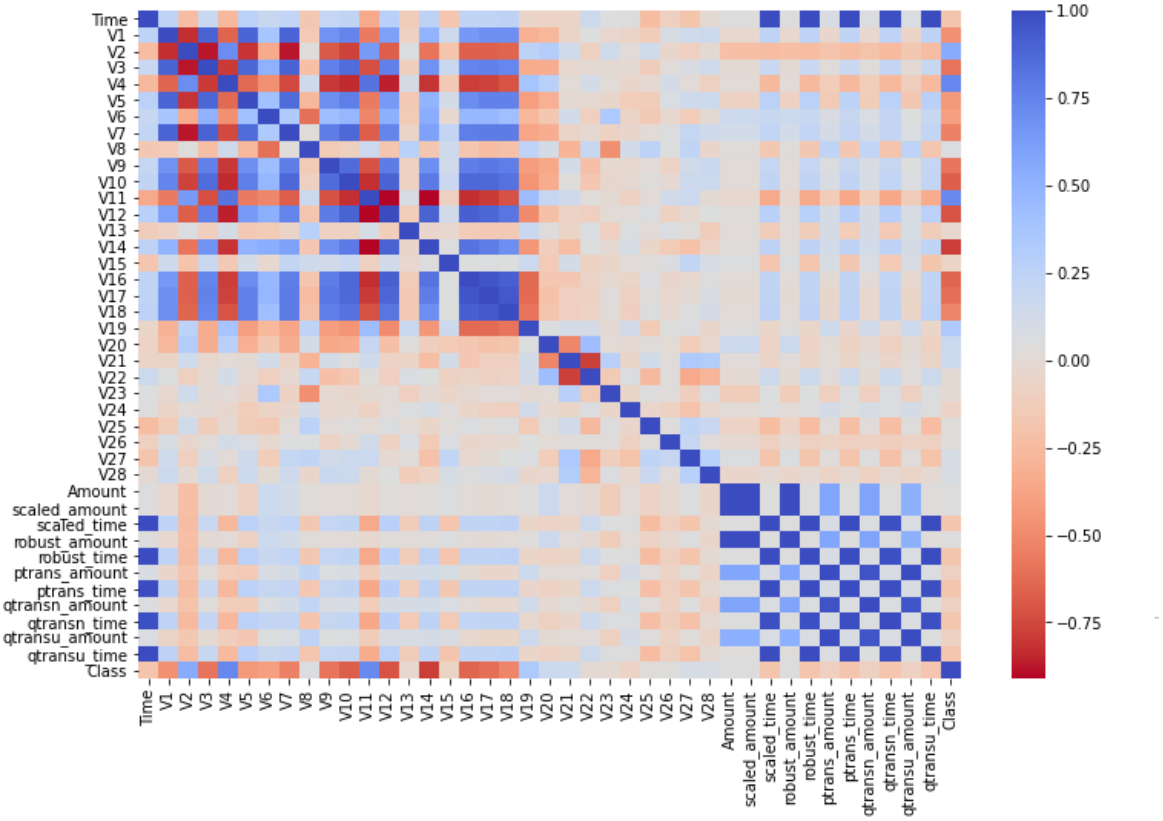


Fig: Correlation Heat map for SMOTE dataset.

Interestingly, we can see a similar heat map for Random Oversampling and SMOTE.

So, in the next section, we will see some analysis of the positive and Negative correlations that we have inferred from this section.

Data re-analysis

In this section, we will focus on analyzing the positive and negative correlation features, especially V1…, V28.

Here we will take a look at the positive and negative correlated features with different sampling techniques and finally choose one or two of the techniques for modelling only.

Positive feature (V1…, V28)

From the last section, we have seen the features V2, V4, V11 and V19 are positively correlated, so going forward, we will see these features only.

[180]:
```
def plot_pos_box(df):
    f,axes = plt.subplots(ncols=4,figsize=(20,7))
    colors = ['#0101DF' , '#DF0101']

    sns.boxplot(x="Class" , y="V11" , data=df,palette=colors,ax=axes[0])
    axes[0].set_title('V11 vs Class Positive Correlation' )

    sns.boxplot(x='Class', y='V4',data=df,palette=colors,ax=axes[1])
    axes[1].set_title('V4 vs Class Positive Correlation' )

    sns.boxplot(x='Class', y='V2',data=df,palette=colors,ax=axes[2])
    axes[2].set_title('V2 vs Class Positive Correlation' )

    sns.boxplot(x='Class', y='V19',data=df,palette=colors,ax=axes[3])
    axes[3].set_title('V19 vs Class Positive Correlation' )
    plt.show()
```

Fig: Function for plot Box-and-whiskers plot for Positive Correlated Data.

Original training data

Below we will see for the original data frame and all the positively correlated features.

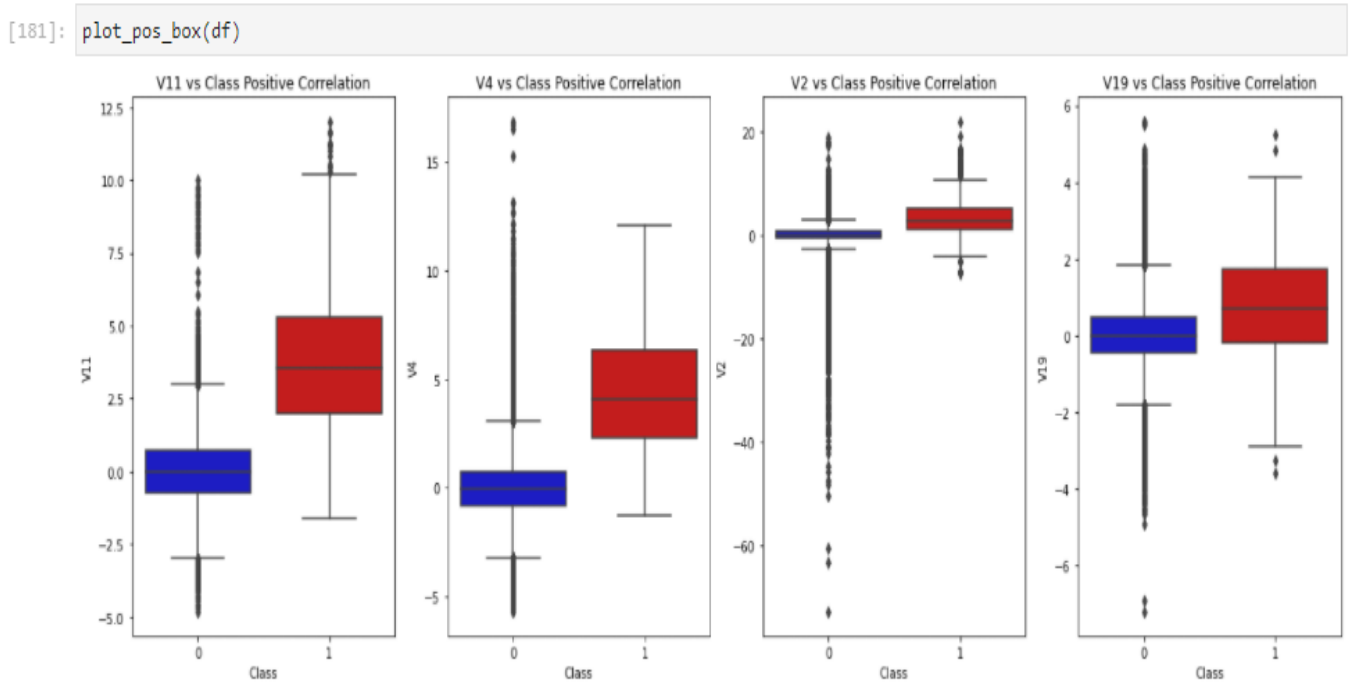


Fig: Original data frame box plot.

We can see that the original training data frame has an extremely large number of outliers for class 0, and that is very bad for modeling, so we went for resampling the dataset.

Random under-sampling

Now let us see the same visualization for Random under sample and compare them.

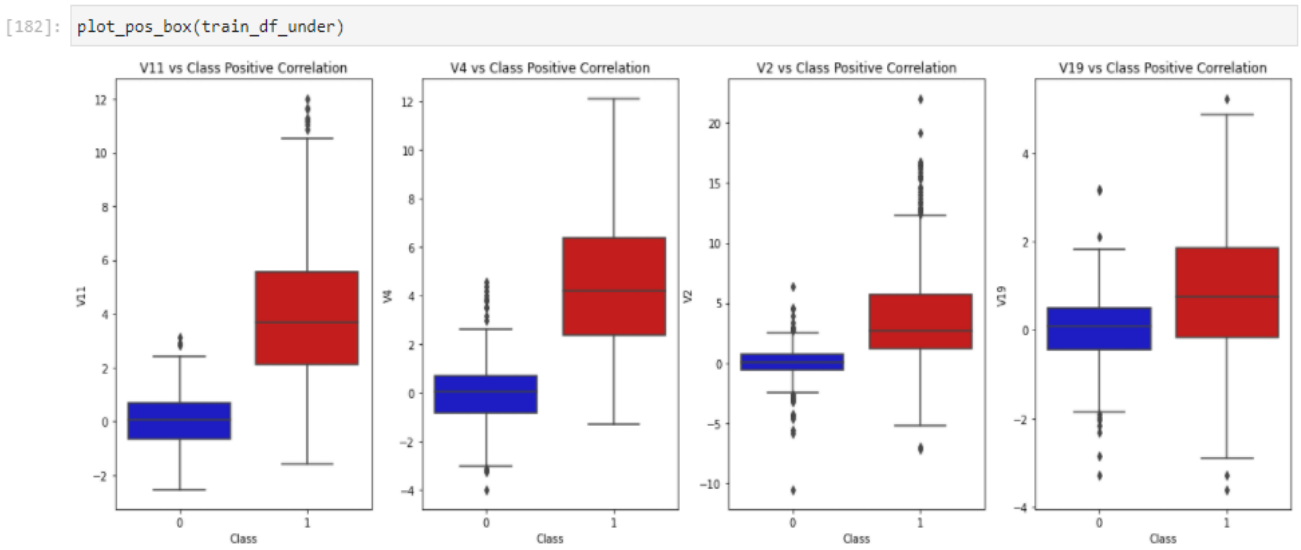


Fig: Random over-sampling box plot.

As this shows so many outliers, we cannot use this dataset for modelling. SO we will be ignoring this dataset.

Tomek's Link under-sampling

Tomek's Link under-sampling was not good in the first place as there were extremely small number of records that were removed due to Tomek's Link.

```
[72]: plot_pos_box(train_df_tkl)
```

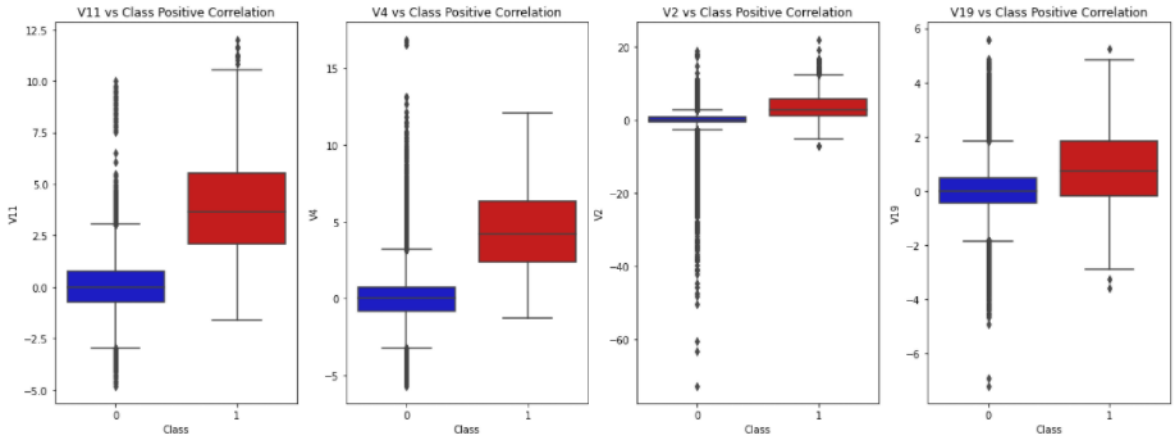


Fig: Tomek's Link under-sampling box plot.

SMOTE

SMOTE is also oversampling technique so that it will look similar to Random Oversampling.

```
[73]: plot_pos_box(train_df_sm)
```

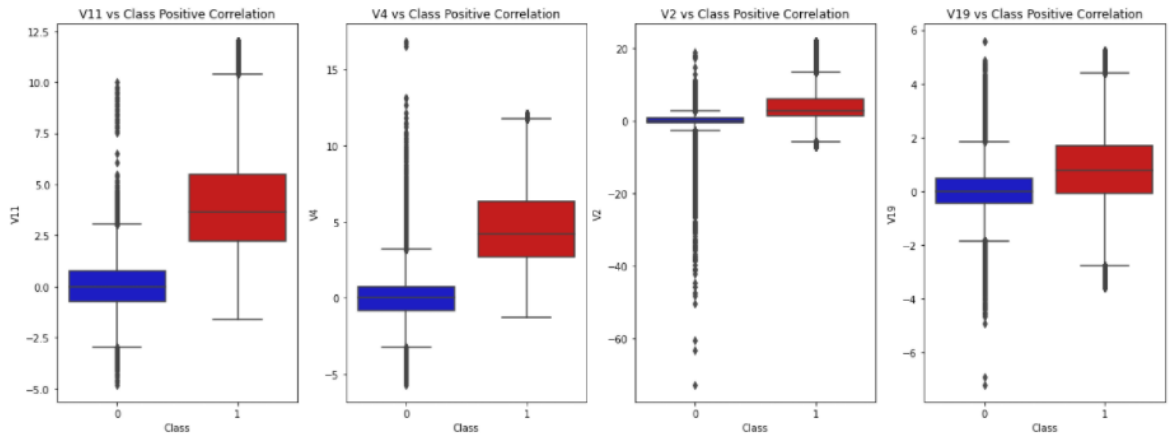


Fig: SMOTE box plot.

Seeing the plot, we can confirm the hypothesis, but SMOTE is an extremely popular technique that is used in case of handling an imbalanced dataset.

We can train the models with one oversampled and one under sampled data and see how it will look.

Negative feature (V1…, V28)

Similarly, we will write a function for negatively correlated values to plot box-and-whiskers and find the optimal resampling technique.

[185]:
```
def plot_neg_box(df):
    f,axes = plt.subplots(ncols=5,figsize=(28,7))
    colors = ['#0101DF' , '#DF0101']

    sns.boxplot(x="Class" , y="V17" , data=df,palette=colors,ax=axes[0])
    axes[0].set_title('V17 vs Class Negative Correlation' )

    sns.boxplot(x='Class', y='V14',data=df,palette=colors,ax=axes[1])
    axes[1].set_title('V14 vs Class Negative Correlation' )

    sns.boxplot(x='Class', y='V12',data=df,palette=colors,ax=axes[2])
    axes[2].set_title('V12 vs Class Negative Correlation' )

    sns.boxplot(x='Class', y='V16',data=df,palette=colors,ax=axes[3])
    axes[3].set_title('V16 vs Class Negative Correlation' )

    sns.boxplot(x='Class', y='V10',data=df,palette=colors,ax=axes[4])
    axes[4].set_title('V10 vs Class Negative Correlation' )
    plt.show()
```

Fig: Function for plot Box-and-whiskers plot for Negative Correlated Data.

```
[186]: plot_neg_box(df)
```

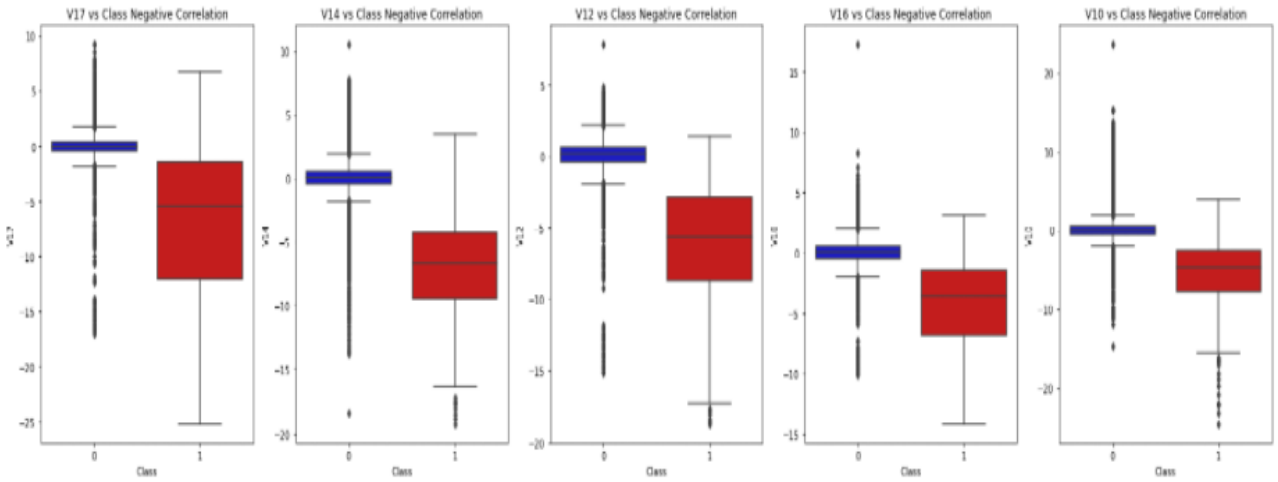


Fig: Original Data.

```
[187]: plot_neg_box(train_df_under)
```

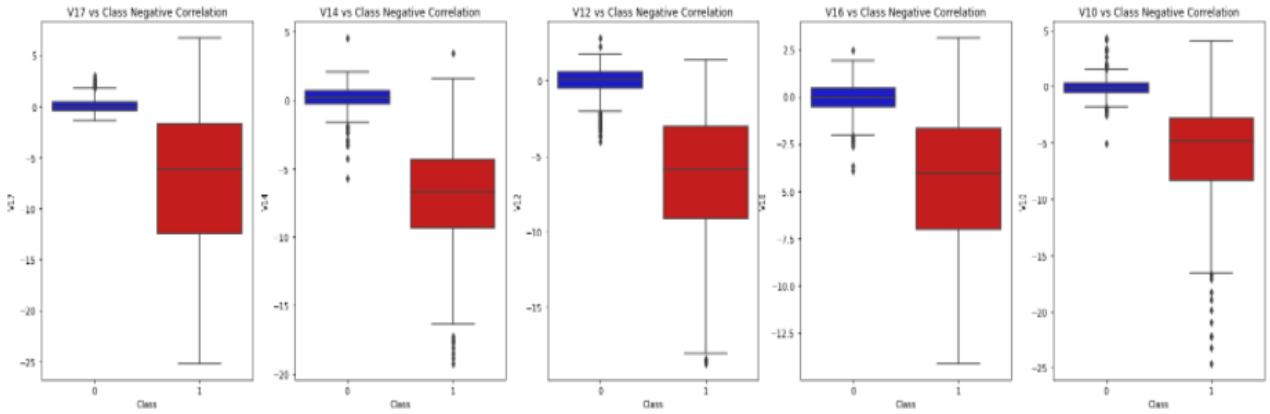


Fig: Random under-sampling box plot.

```
[188]: plot_neg_box(train_df_over)
```

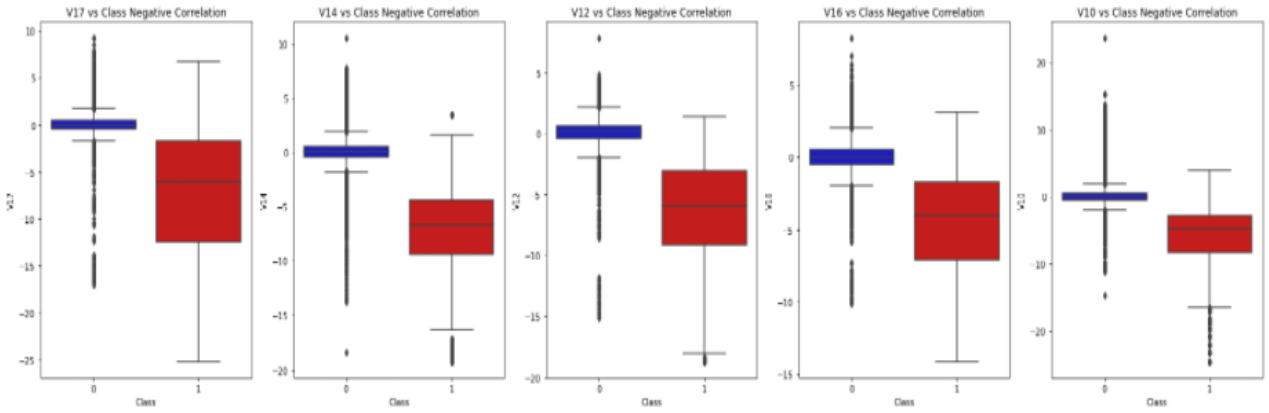


Fig: Random over-sampling box plot.

```
[78]: plot_neg_box(train_df_tkl)
```

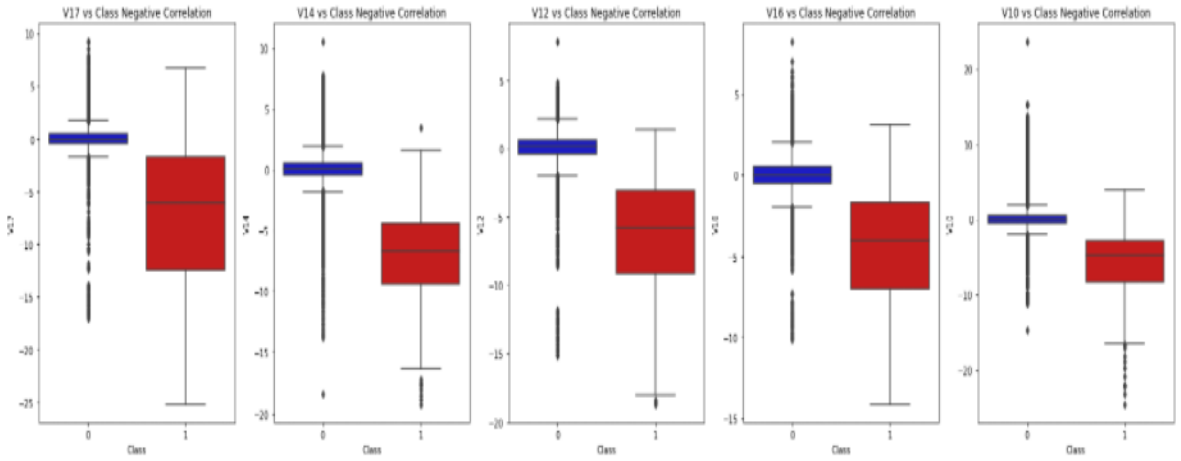


Fig: Tomek's Link under-sampling box plot.

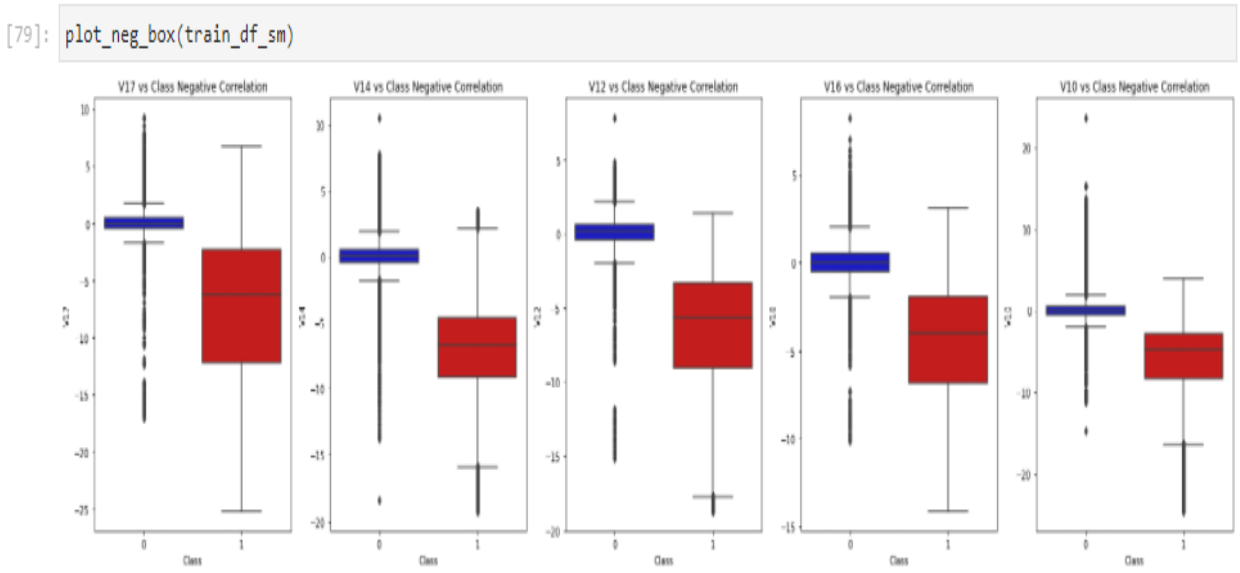


Fig: SMOTE box plot.

Scaled features

Now we have to select which scaling technique to use for the modelling purpose. We have selected only two resampling techniques from the previous section. With that data frame, we will be looking at the scaled data and select the optimal algorithm for the scaled feature(“Time” and “Amount”).

Like before we have to write functions, so we don’t have to write the same code again and again.

The first function is ‘Amount’

Amount

```
[191]: def plot_scaled_amt(df):
           f,axes = plt.subplots(ncols=5,figsize=(28,7))
           colors = ['#0101DF' , '#DF0101']

           sns.boxplot(x="Class" , y='scaled_amount', data=df,palette=colors,ax=axes[0])
           axes[0].set_title('Standard Scaling(Amount)' )

           sns.boxplot(x='Class', y='robust_amount',data=df,palette=colors,ax=axes[1])
           axes[1].set_title('Robust Scaling(Amount)' )

           sns.boxplot(x='Class', y='ptrans_amount',data=df,palette=colors,ax=axes[2])
           axes[2].set_title('Power Transformer(Amount)' )

           sns.boxplot(x='Class', y='qtransn_amount',data=df,palette=colors,ax=axes[3])
           axes[3].set_title('Quartile Transformer -Normal(Amount)' )

           sns.boxplot(x='Class', y='qtransn_amount',data=df,palette=colors,ax=axes[4])
           axes[4].set_title('Quartile Transformer -Uniform(Amount)' )
           plt.show()
```

Fig: Function for plot Box and whiskers plot for Scaled Amount.

Original training data distribution

```
[192]: plot_scaled_amt(df)
```

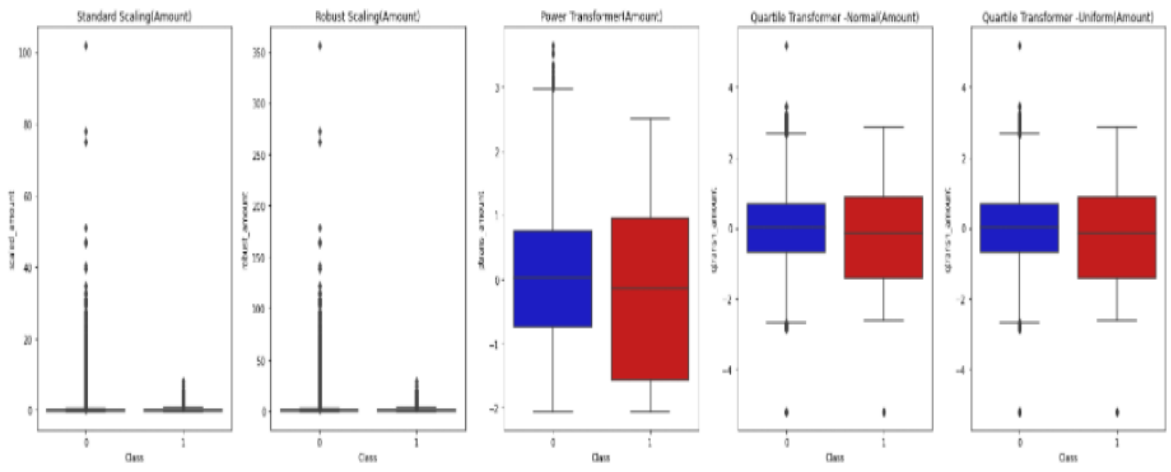


Fig: Original Training Data for Box plot.

Here for amount , we cannot ignore or eliminate the outliers as they will give us a lot of information. We will try to stick with the original distribution with some minor changes.

Let us see for the random under-sampled dataset as we know it reduced the major class, so the density of the outliers will be reduced.

```
[193]: plot_scaled_amt(train_df_under)
```

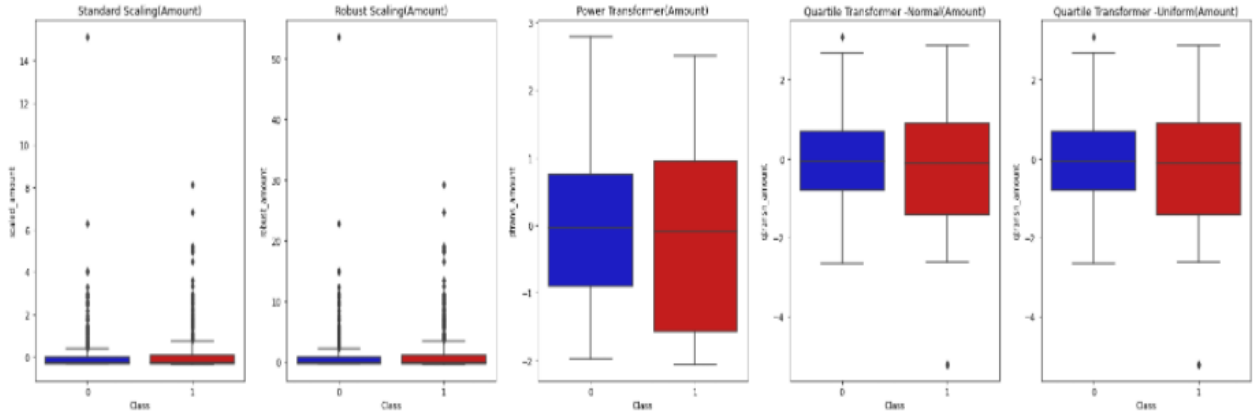


Fig: Random under sampling for Box plot.

We can see outliers have reduced, and Power Transformer looks good in terms of distribution. But we will also look at Robust Scaler as it more or less gives us the original distribution. “Quantile Transformer-Normal “looks good, but it has already performed badly for the future ‘Time’, so we will ignore it.

```
[194]: plot_scaled_amt(train_df_over)
```

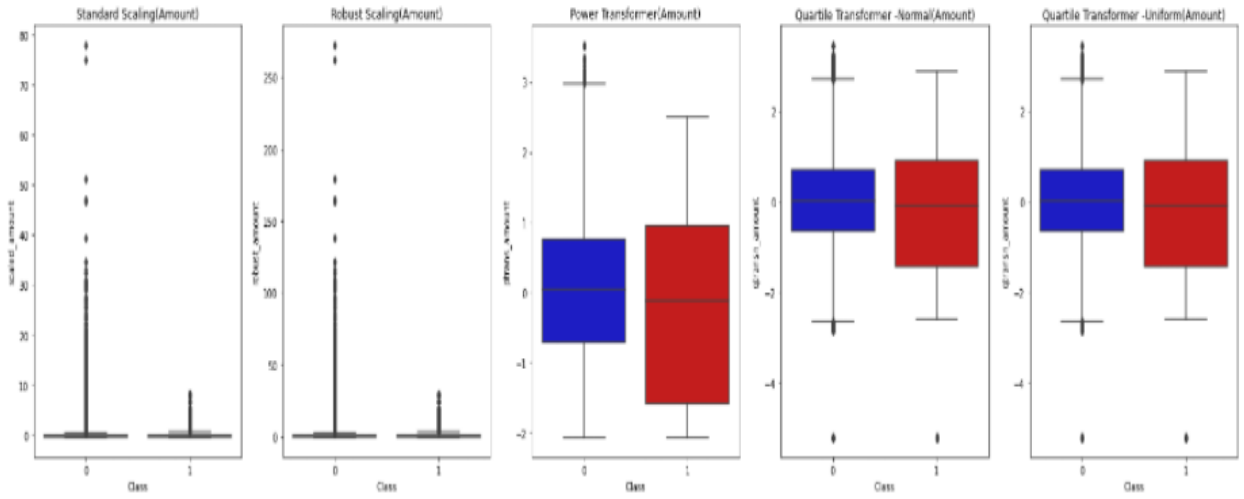


Fig: Random over-sampling box plot.

```
[84]: plot_scaled_amt(train_df_tkl)
```

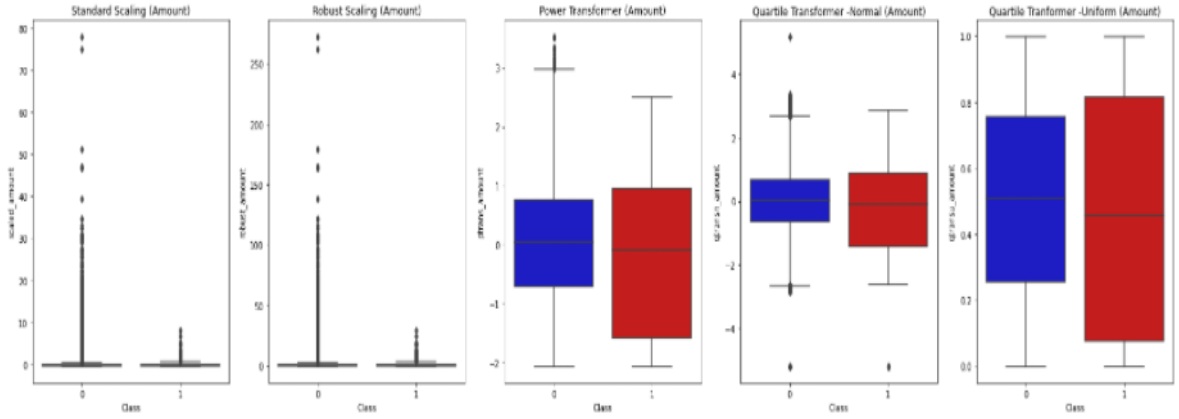


Fig: Tomek Link's box plot.

We will also take a look at the SMOTE dataset and see its plot.

```
[85]: plot_scaled_amt(train_df_sm)
```

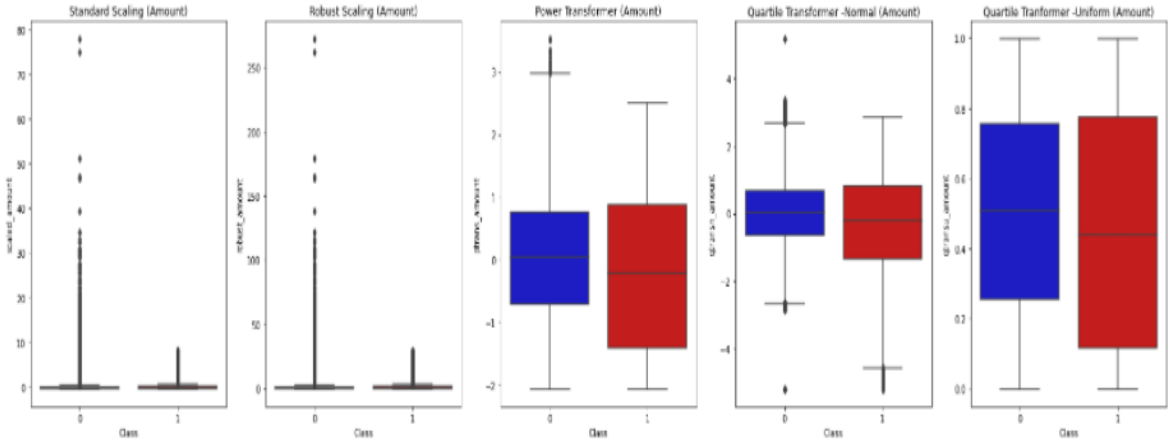


Fig: SMOTE for box plot.

SMOTE plot shows a similar thing like the original training dataset.

2nd Function is “Time”

```
[197]: def plot_scaled_time(df):
           f,axes = plt.subplots(ncols=5,figsize=(28,7))
           colors = ['#0101DF' , '#DF0101']

           sns.boxplot(x="Class" , y='scaled_time', data=df,palette=colors,ax=axes[0])
           axes[0].set_title('Standard Scaling(time)' )

           sns.boxplot(x='Class', y='robust_time',data=df,palette=colors,ax=axes[1])
           axes[1].set_title('Robust Scaling(time)' )

           sns.boxplot(x='Class', y='ptrans_time',data=df,palette=colors,ax=axes[2])
           axes[2].set_title('Power Transformer(time)' )

           sns.boxplot(x='Class', y='qtransn_time',data=df,palette=colors,ax=axes[3])
           axes[3].set_title('Quartile Transformer -Normal(time)' )

           sns.boxplot(x='Class', y='qtransn_time',data=df,palette=colors,ax=axes[4])
           axes[4].set_title('Quartile Transformer -Uniform(time)' )
           plt.show()
```

Fig: Function for plot Box-and-whiskers plot for Scaled Time.

Time

Let us first make a yardstick to compare the results with. We will plot first for the original data frame.

```
[198]: plot_scaled_time(df)
```

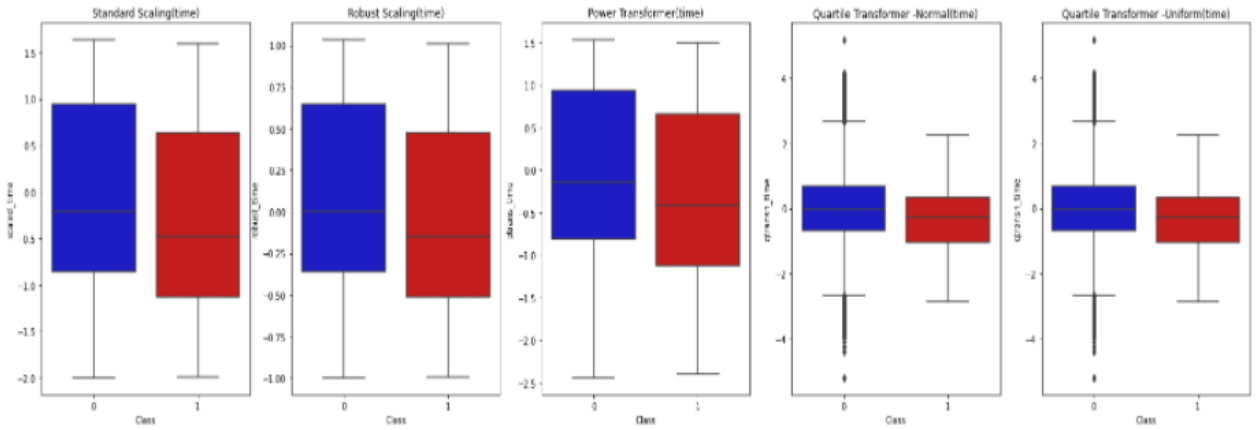


Fig: Original Training Data for Box plot.

We can see for “Quantile Transformer-Normal” there is skewness and outliers in the plot. We now have to compare this entire plot with Random Under sampling and SMOTE. We will go ahead with the under-sampling dataset and see how it look.

```
[199]: plot_scaled_time(train_df_under)
```

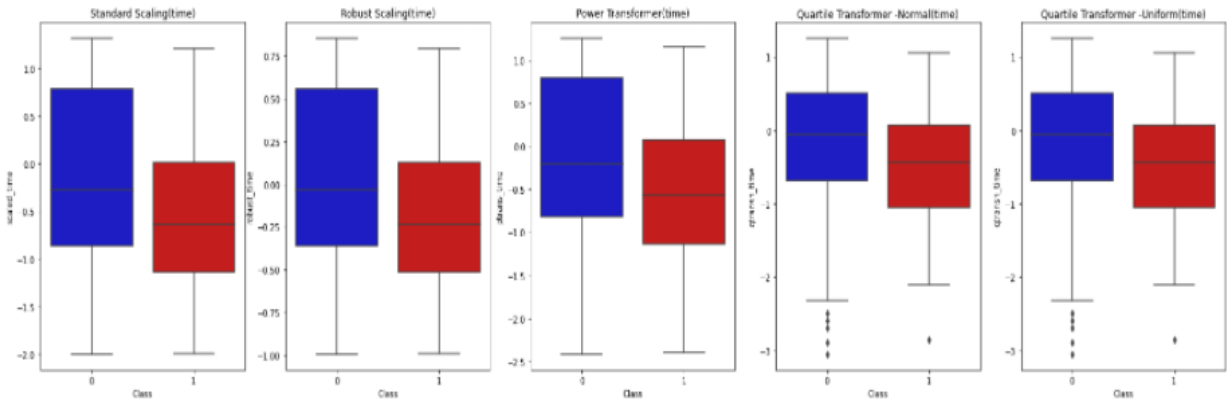


Fig: Random under-sampling for Box plot.

For this dataset, we can see the same thing that for “Quantile Transformer-Normal,” there are outliers in the plot. Apart from that all looks good for use. Previously, we have seen that “Quantile Transformer-Uniform” actually changed the distribution so that can be ignored as well.

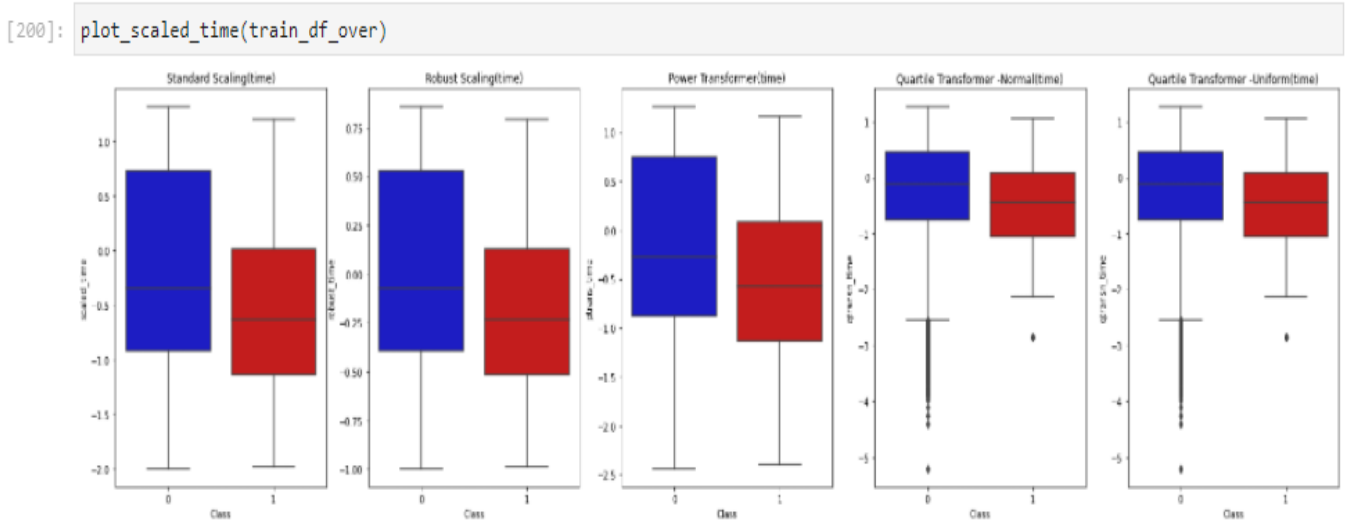


Fig: Random over-sampling for Box plot.

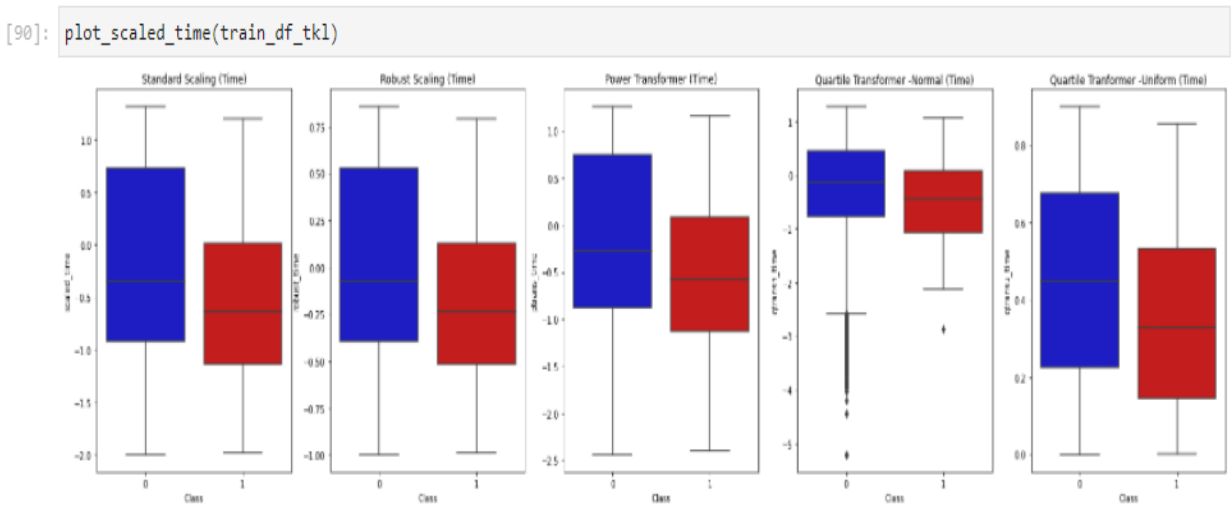


Fig: Tomek’s Link for Box plot.

Let us see for SMOTE as well , and conclude with some optimal results

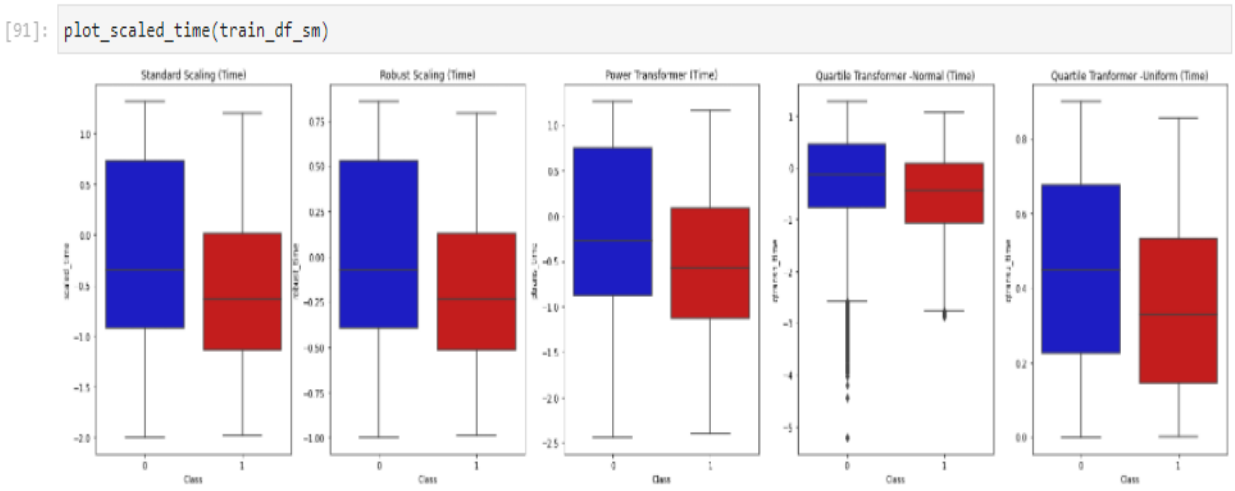


Fig: SMOTE for Box plot.

For SMOTE dataframe , we see the same result , so, for the feature “Time,” we can choose any algorithm from “Quantile Transformer.”

We have to be in sync with the feature “Amount” as well. It is not mandatory to select the same scaling algorithm for all features, but whatever we do, we need to have some justification for that action.

Syncing we with ‘Amount’ we find that we will go ahead with both Robust Scaling and Power Transformer for both “Time” and “Amount”. Finally we will have four dataframe to run all the models.

SMOTE Dataset 1 - Robust Scaled Time, Robust Scaled Amount , V2, V4,V11, and V19(Positive), V17,V14,V12, V16 and V10(Negative).

SMOTE Dataset 2 - Power Transformer Time, Power Transformer Amount , V2, V4,V11, and V19(Positive), V17,V14,V12, V16 and V10(Negative).

Random Under-Sample Dataset 1 - Robust Scaled Time, Robust Scaled Amount , V2, V4,V11, and V19(Positive), V17,V14,V12, V16 and V10(Negative).

Random Under-Sample Dataset 2 - Power Transformer Time, Power Transformer Amount , V2, V4,V11, and V19(Positive), V17,V14,V12, V16 and V10(Negative).

Modelling

We always need to know that data preparation is 70% of the entire pipeline, so we have to give importance to the data. If the data is well prepared, then modelling will be ease, and we can achieve optimal accuracy for the model. Then we need to tune the model iteratively to improve it further.

Machine Learning Algorithms

Here we will see some basic algorithm to make the foundation.

Logistic Regression

Logistic regression is also known as a logit regression, is a statistical model and is somewhat similar to the concepts of Linear regression. The basic version of Logistic Regression utilizes logistic function to model a binary dependent variable.

Logit function

$$Logit(p) = log\left(\frac{p}{1-p}\right)$$

Where,

$$p = probability,$$

$$\frac{p}{1-p} = corresponding\ odds.$$

Hence, it is known as $log\ of\ odds\ or\ log(odds)$

$$logit^{-1}(\alpha) = logistic(\alpha) = \frac{1}{1+e^{-\alpha}} = \frac{\exp^{\alpha}}{\exp^{\alpha}+1} = \sigma(\alpha)$$

Where,

$$0 < \sigma(\alpha) < 1\ i.e. Sigmoid(\sigma) only\ ranges\ from\ [0,1]$$

Decision Tree

A **decision tree** is a popular classification algorithm, but it can be used as a regression algorithm as well.

Structure of a decision tree

The decision tree resembles a binary tree4 which we generally study under Data Structures. This tree is upside down of an original tree as “roots” starts from the top, and as we go down, we have “leaves.”

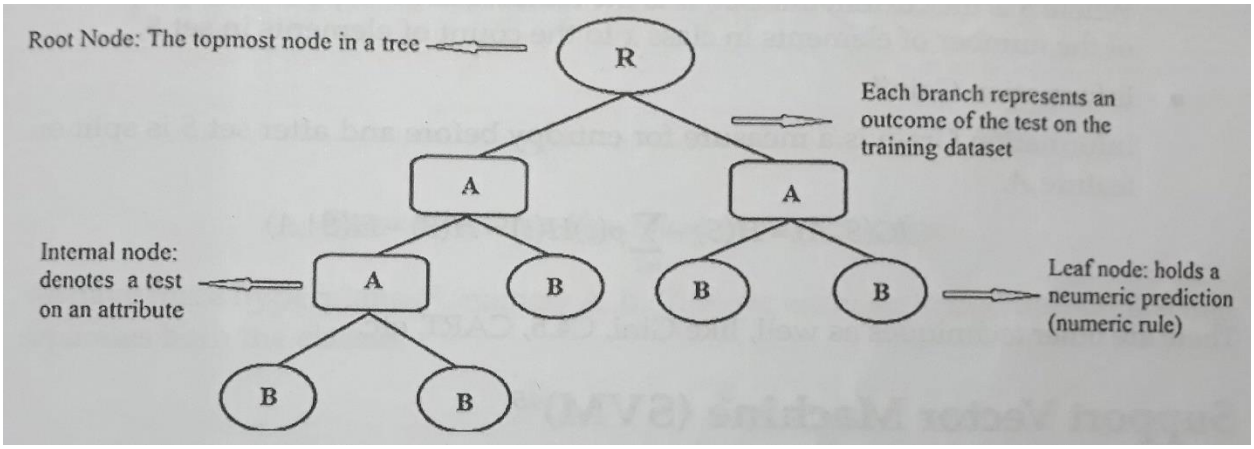


Fig: Decision tree structure.

This is an important structure to remember, and the internal node is a decision node, and it splits the data.

Algorithm

- **Step 1:** Select the best decision node using **Attribute Selection Methods (ASM).**

- **Step 2:** Now using the selected Decision Node break the dataset into the smaller subsets.

- **Step 3:** Repeat Step 1 and Step 2 for all the child nodes until and unless the below conditions are met.

    - There are no more remaining features to deal with.
    - There are no more records.
    - All records belong to the same feature.

Mathematics behind ASM

Iterative Dichotomiser 3(ID3)

In ID3, information gain is calculated for each remaining attribute. The attribute with the largest information gain as a decision node that splits the set on this iteration.

- Entropy:
  Entropy is a measure of the uncertainty in the dataset S.

$$H(S) = \sum_{x \epsilon X} -p(x) \log_2 p(x)$$

  Where S is the current dataset, X is the features in set S, p(x) is the proportion of the number of elements in class x to the count of elements in set S.

- Information Gain:
  Information gain is a measure for entropy before and after set S is split on feature A.

$$IG(S, A) = H(S) - \sum_{t \varepsilon T} p(t)H(t) = H(S) - H(S|A)$$

Support Vector Machine (SVM)

Support Vector Machine(SVM) is a model generally used for classification and regression problems. It can solve linear and nonlinear problems that help the algorithm to tackle real-time projects.

The concept of Linear SVM is simple. The algorithm creates a line or a hyperplane which separates the data into classes.

How does it work?

Let us see the working of Linear SVM visually, as it will be simpler to understand.

First we will plot all the data points with class, and then we draw multiple hyperplanes, here we will try to separate the two classes.

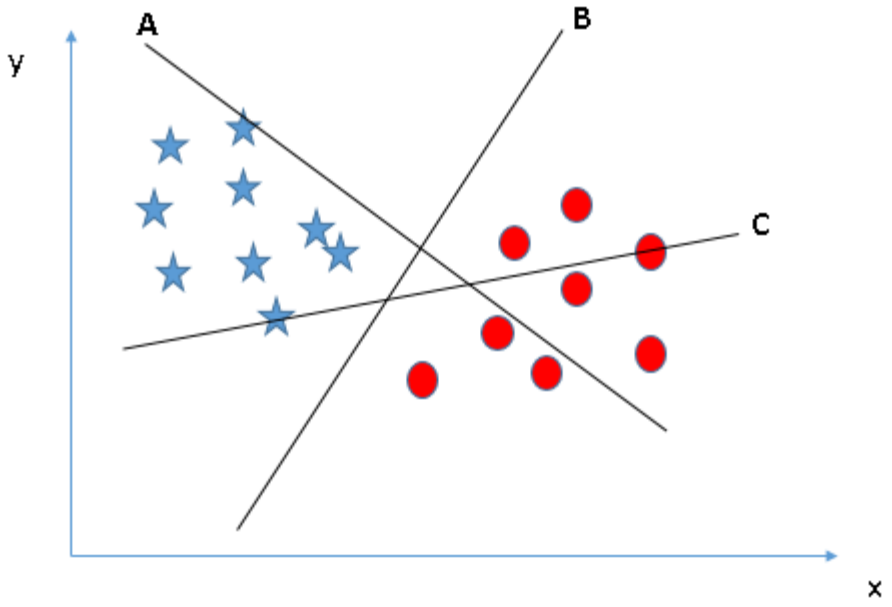


Fig: Draw Hyperplane.

We have three hyperplanes, namely A,B,C. Now, we need to find the best line that separates both the classes.

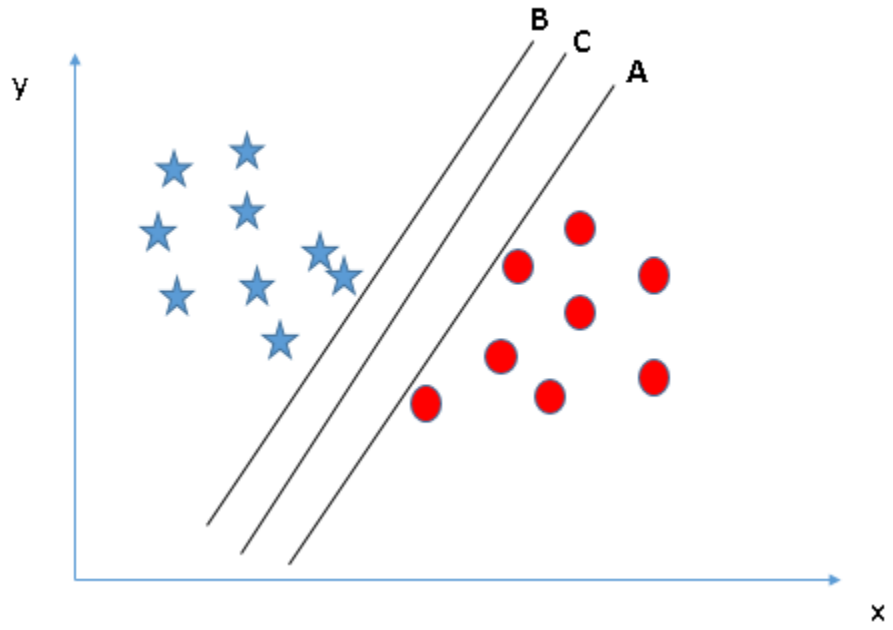


Fig: Find the Hyperplane separating them.

Say we have three hyperplanes that separate both the classes like the above image. Now we need to choose one optimal hyperplane.

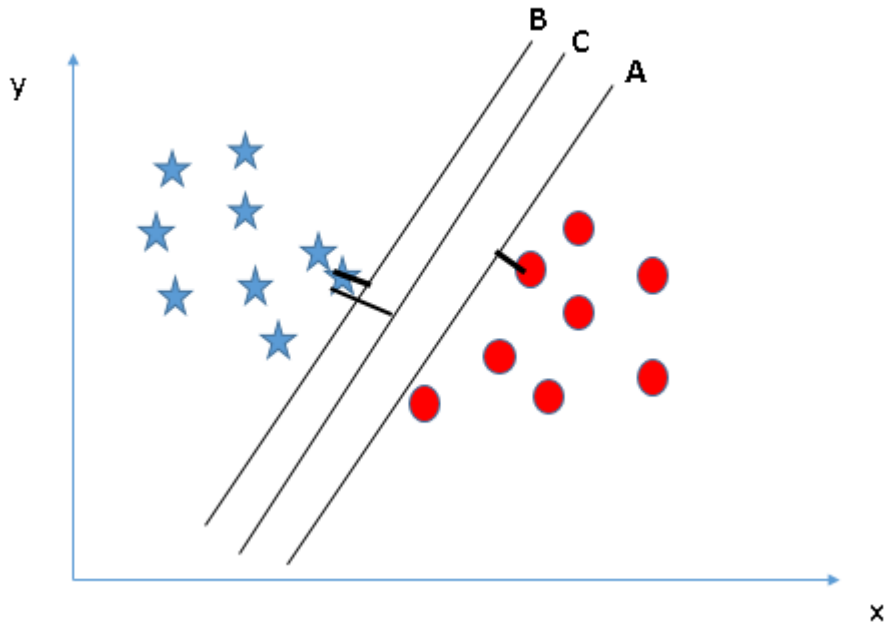


Fig: Find the best Hyperplane separating them.

To select optimal hyperplane from the three planes, we need to see the ,margin like the above image. Margin is the distance from the hyperplane to the nearest data point with both the classes. Our target is to maximize both the distance from two classes and come to an optimal point. Here we can see the particular hyperplane is C.

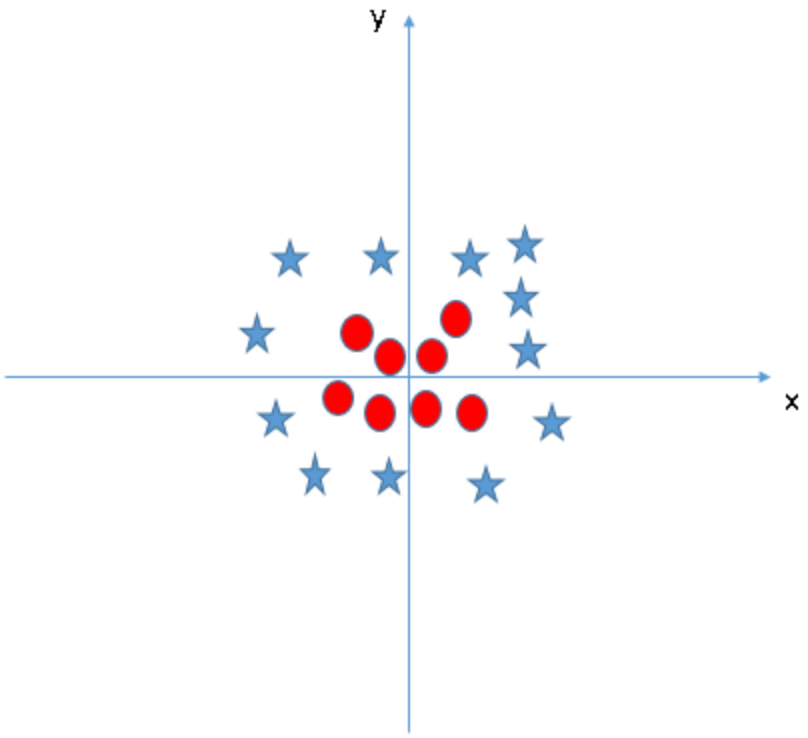


Fig: Non-linearly separable.

In the above example, we saw that we could easily divide both the classes using a Line,Plane,Hyperplane, but that won't be case all the time.

Let us see one more example where we cannot use a linear SVM to solve the problem.

From the above image, we can see that the data point and class won't be separated by a hyperplane. We have multiple techniques to handle nonlinear decision boundaries using kernels.

From the above plot we can transform the data and plot like the below figure.

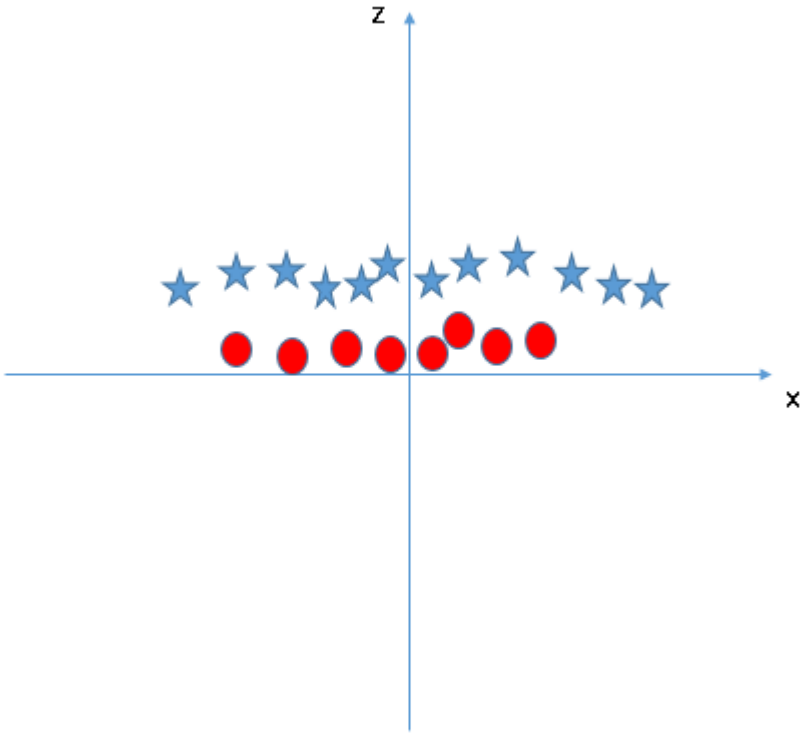


Fig: Transform Data.

Here for this particular example, we are projecting the same data points to a higher dimension and trying to separate it by a hyperplane. But when we transform back to the original dimension, it will look like the below image.

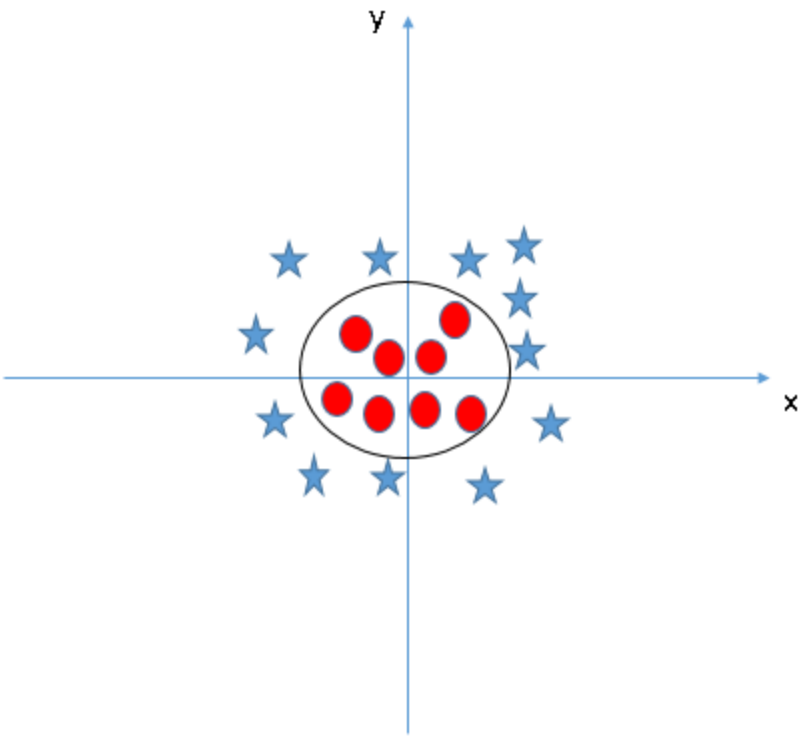


Fig: Separate using nonlinear kernel.

After bringing back to the original dimension, we can see an elliptical decision boundary or we can say this as non-linear hyper surface. In SVM, we generally achieve this Radial Basis Kernel.

Basic mathematics behind the Linear Hard-Margin Classifier (Linear SVM) for Strictly Separable Case:

Let us start by defining the plane.

$$d(x, w, b) = w^T x + b = \sum_{i=1}^{n} w_i x_i + b$$

$So, now\ we\ need\ to\ do\ a\ classification\ if$:

$1. d(x, w, b) > 0, classify\ x\ as\ class\ 1 (i.e. its\ associated\ y = +1)$

$2. d(x, w, b) < 0, classify\ x\ as\ class\ 2 (i.e. its\ associated\ y = -1)$

$3. d(x, w, b) = 0, then\ x\ is\ equi - probable\ for\ both\ classes\ so, as\ per\ the\ business\ problem\ we\ classify\ this\ data\ point.$

$Where, d(x, w, b) is\ the\ distance\ of\ point\ x\ from\ the\ hyperplane.$

$Greater\ the\ distance\ stronger\ the\ classification\ of\ x.$

$After\ finding\ the\ optimal\ of\ the\ hyperplane\ w^*, b^*\ our\ decision\ boundary\ has\ a\ from\ like\ below$:

$$f\hat{}(x) = sgn\big(d(x, w^*, b^*)\big) = sgn(w^{*T}x + b); sgn(x) =_= \begin{cases} 1, & x > 0 \\ -1 & if\ x < 0 \\ 0, & x = 0 \end{cases}$$

These are the basics of SVM.

Model training

In this section, we will train different models with different parameters of it on all datasets and find the optimal model to use for this problem statement.

Before we start with it, we need to make the dataset ready for training.

Data preparation

We already have decided with the list of columns we will use to for training as shown below:

```
[203]: col_set1=["robust_amount","robust_time","V2","V4","V10","V11","V12","V14","V16","V17","V19"]
       col_set2=["ptrans_amount","ptrans_time","V2","V4","V10","V11","V12","V14","V16","V17","V19"]
```

Fig: Multiple set of columns.

We have two sets of columns and we will also use different resampled data frame. In this case, we will use SMOTE and Random under-sampled dataset. So, there will be a total of 4 datasets that we need to train.

```
[204]: X_train_smote_robust = train_df_sm[col_set1]
       y_train_smote_robust = train_df_sm["Class"]

[205]: X_train_smote_power = train_df_sm[col_set2]
       y_train_smote_power = train_df_sm["Class"]
```

Fig: SMOTE data preparation.

The above set is for SMOTE with both Robust Sampling and Power Transformed data. We will similarly have two dataframe for Random Under Sample.

```
[206]: X_train_under_robust = train_df_under[col_set1]
       y_train_under_robust = train_df_under["Class"]

[207]: X_train_under_power = train_df_under[col_set2]
       y_train_under_power = train_df_under["Class"]
```

Fig: Random under-sample data preparation.

So, we now have four training datasets, and we will use multiple algorithms to find the best algorithm out of it.

Metric trap

This is one of the most important points which we need to ponder upon, i.e., the evaluation metrics. Here “accuracy” cannot be used as a metric because this is an imbalanced dataset, so we make a function that will always return class zero, then the accuracy of the model(function)is more than 95%.

```
[208]: def metric_trap_acc(x=None):
           return 0 #Class 0
```

Fig: Pseudo model (should not be used).

The above function/pseudo-model has great accuracy as it will always return class zero. The probability of getting class zero for the imbalanced data set is more than 95%.

But for quick analysis, we will take a look at accuracy to make some ideas and guess about the model’s performance.

Training & evaluation

In this section, we will train all the models with four different kinds of the dataset and evaluate some of the models, rest models can be evaluated by you as a practice and get more insight about the data and the process.

Let us prepare the dataset, we will use them in a loop to train.

```
[209]: data=[
           [X_train_smote_robust,y_train_smote_robust,"SMOTE -Robust Scaling"],
           [X_train_smote_power,y_train_smote_power,"SMOTE -Power Transformer"],
           [X_train_under_robust,y_train_under_robust,"Under Sampling -Robust Scaling"],
           [X_train_under_power,y_train_under_power,"Under Sampling -Power Transformer"]
       ]
```

Fig: Preparation of training data.

Above, we can see that we will be using one variable for all the training data. We are planning to write a small code that all the models for all the data at once using cross-validation.

```
[210]: #Classifier Libraries
       from sklearn.linear_model import LogisticRegression
       from sklearn.svm import SVC
       from sklearn.neighbors import KNeighborsClassifier
       from sklearn.tree import DecisionTreeClassifier

       classifiers = {
           "DecisionTreeClassifier":DecisionTreeClassifier(),
           "LogisticRegression":LogisticRegression(max_iter=1000),
           "KNearest":KNeighborsClassifier(),
           "Radial Basis Support Vector Classifier":SVC()
       }
```

Fig: Initializing models.

We have a separate set of testing data, but we need to test with the training data as well, using the k-fold cross-validation technique. This will decrease the chance of selection bias. With this process, we will select the optimal dataset and some of the models for the validation.

```
[99]: from sklearn.model_selection import cross_val_score
for X,y,name in data:
    print("\n\n" +name+ ":\n")
    for key, classifier in classifiers.items():
        classifier.fit(X,y)
        training_score = cross_val_score(classifier, X, y, cv=5)
        print("Classifiers:", classifier.__class__.__name__, "has a training score of", \
              round(training_score.mean(),2)*100, "% accuracy score")
```

Fig: Fitting and cross-validating.

From the above code, we will get the statistics about training data accuracy for all the combinations. With every step, we should always aim to reduce the number of computations and have limited models to evaluate.

Let's start with the SMOTE dataset and see how it looks.

```
SMOTE -Robust Scaling:

Classifiers: DecisionTreeClassifier has a training score of 67.0 % accuracy score
Classifiers: LogisticRegression has a training score of 96.0 % accuracy score
Classifiers: KNeighborsClassifier has a training score of 100.0 % accuracy score
Classifiers: SVC has a training score of 97.0 % accuracy score


SMOTE -Power Transformer:

Classifiers: DecisionTreeClassifier has a training score of 67.0 % accuracy score
Classifiers: LogisticRegression has a training score of 96.0 % accuracy score
Classifiers: KNeighborsClassifier has a training score of 99.0 % accuracy score
Classifiers: SVC has a training score of 96.0 % accuracy score


Under Sampling -Robust Scaling:

Classifiers: DecisionTreeClassifier has a training score of 74.0 % accuracy score
Classifiers: LogisticRegression has a training score of 93.0 % accuracy score
Classifiers: KNeighborsClassifier has a training score of 93.0 % accuracy score
Classifiers: SVC has a training score of 93.0 % accuracy score


 Under Sampling -Power Transformer:

 Classifiers: DecisionTreeClassifier has a training score of 74.0 % accuracy score
 Classifiers: LogisticRegression has a training score of 93.0 % accuracy score
 Classifiers: KNeighborsClassifier has a training score of 93.0 % accuracy score
 Classifiers: SVC has a training score of 93.0 % accuracy score
```

Fig: Cross-validation training score for SMOTE and under-sampling.

We have similar result for Robust Scaling and Power Transformer. For a new algorithm, we are using k-NN as well, but we won't be using it. We can see that k-NN's accuracy is 100%, and it is highly probable and possible that is suffering from over-fitting .

Random under-sampling has a different result and is not as great as the oversampling results. We can use Decision Tree, Logistic Regression for SMOTE and Support Vector Machine,

k-NN for Random Under Sampling. From now on, we will be using the test data to evaluate the model.

Note: The order of the features in the dataset during training and testing should be the same else it will give us a wrong result.

Let us write a code snippet to do the same, as we mentioned above.

```
[212]: from sklearn.linear_model import LogisticRegression
       from sklearn.svm import SVC
       from sklearn.neighbors import KNeighborsClassifier
       from sklearn.tree import DecisionTreeClassifier

       SVC_Under_Sampling_Power = SVC(shrinking=False, probability=True)
       kNN_Under_Sampling_Power = KNeighborsClassifier()

       DT_SMOTE_Robust = DecisionTreeClassifier()
       LR_SMOTE_Robust = LogisticRegression(max_iter=1000)

       SVC_Under_Sampling_Power.fit(X_train_under_power, y_train_under_power)
       kNN_Under_Sampling_Power.fit(X_train_under_power, y_train_under_power)

       DT_SMOTE_Robust.fit(X_train_smote_robust,y_train_smote_robust)
       LR_SMOTE_Robust.fit(X_train_smote_robust,y_train_smote_robust)

[212]: LogisticRegression(max_iter=1000)
```

Fig: Training models.

From now on we will use these names instead of model name and the scaling name every time.

We had discussed before the metric trap, and just using accuracy will give us wrong information. There are multiple metrics we can see. But we will **Receiver Operating Characteristics(ROC)**to analyze all the models.

ROC is a way to analyze the performance of a binary classifier system as its discrimination threshold is valid.

We will first take a look at the ROC Curve then it will be simpler to explain what ROC score is.

ROC curve is a plot between True Positive Rate and False Positive Rate at various threshold values.

```
[100]: from sklearn.metrics import roc_curve

log_fpr, log_tpr, log_thresold = roc_curve(y_test,
                                           LR_SMOTE_Robust.predict_proba(X_test_robust)[:,1])
knear_fpr, knear_tpr, knear_threshold = roc_curve(y_test,
                                                  kNN_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
svc_fpr, svc_tpr, svc_threshold = roc_curve(y_test,
                                            SVC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
tree_fpr, tree_tpr, tree_threshold = roc_curve(y_test,
                                               DT_SMOTE_Robust.predict_proba(X_test_robust)[:,1])
```

Fig: Generating values for plotting ROC curve.

The above code snippet will give true positive rates, false-positive rates, and the threshold for all the models. Now we have a task to plot the data and visualize it.

```
plt.figure(figsize=(16,8))
plt.title('ROC Curve \n Top 4 Classifiers', fontsize=18)
plt.plot(log_fpr, log_tpr,
         label='LR_SMOTE_Robust: {:.4f}'.format( \
             roc_auc_score(y_test, LR_SMOTE_Robust.predict_proba(X_test_robust)[:,1])))
plt.plot(tree_fpr, tree_tpr,
         label='DT_SMOTE_Robust: {:.4f}'.format( \
             roc_auc_score(y_test, DT_SMOTE_Robust.predict_proba(X_test_robust)[:,1])))
plt.plot(knear_fpr, knear_tpr,
         label='kNN_Under_Sampling_Power: {:.4f}'.format( \
             roc_auc_score(y_test, kNN_Under_Sampling_Power.predict_proba(X_test_power)[:,1])))
plt.plot(svc_fpr, svc_tpr,
         label='SVC_Under_Sampling_Power: {:.4f}'.format( \
             roc_auc_score(y_test, SVC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])))
plt.plot([0, 1], [0, 1], 'k--')
plt.axis([-0.01, 1, 0, 1])
plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.annotate('Minimum ROC Score of 50%', xy=(0.5, 0.5), xytext=(0.6, 0.3),
             arrowprops=dict(facecolor='#6E726D', shrink=0.05),)
plt.legend()
plt.show()
```

Fig: Plotting ROC curve.

Below we will see the ROC curve generated by the above code.

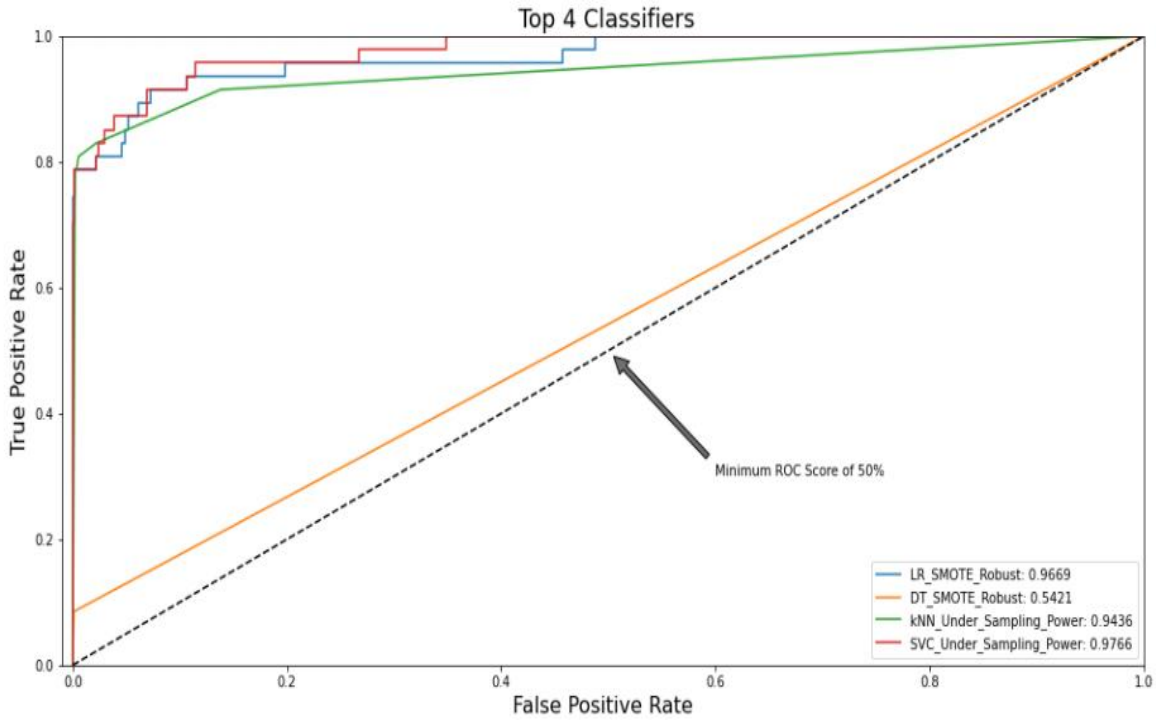


Fig: ROC curve.

We should know the higher the value of ROC, the better the model is. We have drawn a reference line worth 0.5 as a ROC score. It means that lines/models close to the reference line signifies a random guess like a coin toss where the probability is 0.5 for each class.

ROC score is the area under the curve which is generated from the ROC curve. ROC score is also referred to as AUC or AUROC.

ROC score summarizes the curve information in a single value. We will see what the score for all the model is:

```
[99]: from sklearn.metrics import roc_auc_score

print("ROC Accuracy Score:\n")

print('LR_SMOTE_Robust: ', roc_auc_score(y_test,
                                         LR_SMOTE_Robust.predict_proba(X_test_robust)[:,1]))
print('DT_SMOTE_Robust: ', roc_auc_score(y_test,
                                         DT_SMOTE_Robust.predict_proba(X_test_robust)[:,1]))
print('kNN_Under_Sampling_Power: ', roc_auc_score(y_test,
                                                  kNN_Under_Sampling_Power.predict_proba(X_test_power)[:,1]))
print('SVC_Under_Sampling_Power: ', roc_auc_score(y_test,
                                                  SVC_Under_Sampling_Power.predict_proba(X_test_power)[:,1]))
```

Fig: Generating AUROC curve.

The above code will give us the scores for all the values, and this will help us to select the best or optimal model, among others.

```
ROC Accuracy Score:

LR_SMOTE_Robust:  0.9668966717050875
DT_SMOTE_Robust:  0.5420576338099874
kNN_Under_Sampling_Power:  0.9435670512887135
SVC_Under_Sampling_Power:  0.9765830168316392
```

Fig: ROC Score.

We can see from the score and curve that Decision Tree performed worst hence we can drop that model. The best model as per score is SVC, but we also see a close score to SVC that is of the model Logistic Regression model. Now, both of the models are similar and deciding the right model will be more of a business decision.

Now we will view precision recall curve.

```
[101]: from sklearn.metrics import precision_recall_curve

log_tpr, log_fpr, log_thresold = precision_recall_curve(y_test,
                                        LR_SMOTE_Robust.predict_proba(X_test_robust)[:,1])
knear_tpr, knear_fpr, knear_threshold = precision_recall_curve(y_test,
                                                kNN_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
svc_tpr, svc_fpr, svc_threshold = precision_recall_curve(y_test,
                                         SVC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
tree_tpr, tree_fpr, tree_threshold = precision_recall_curve(y_test,
                                           DT_SMOTE_Robust.predict_proba(X_test_robust)[:,1])

plt.figure(figsize=(16,8))
plt.title('Precision-Recall Curve \n Top 4 Classifiers', fontsize=18)
plt.plot(log_fpr, log_tpr,
         label='LR_SMOTE_Robust: {:.4f}'.format( \
             roc_auc_score(y_test, LR_SMOTE_Robust.predict_proba(X_test_robust)[:,1])))
plt.plot(tree_fpr, tree_tpr,
         label='DT_SMOTE_Robust: {:.4f}'.format( \
             roc_auc_score(y_test, DT_SMOTE_Robust.predict_proba(X_test_robust)[:,1])))
plt.plot(knear_fpr, knear_tpr,
         label='kNN_Under_Sampling_Power: {:.4f}'.format( \
             roc_auc_score(y_test, kNN_Under_Sampling_Power.predict_proba(X_test_power)[:,1])))
plt.plot(svc_fpr, svc_tpr,
         label='SVC_Under_Sampling_Power: {:.4f}'.format( \
             roc_auc_score(y_test, SVC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])))
plt.xlabel('Recall', fontsize=16)
plt.ylabel('Precision', fontsize=16)
plt.legend()
plt.show()
```

Fig: Plotting Precision Recall Curve.

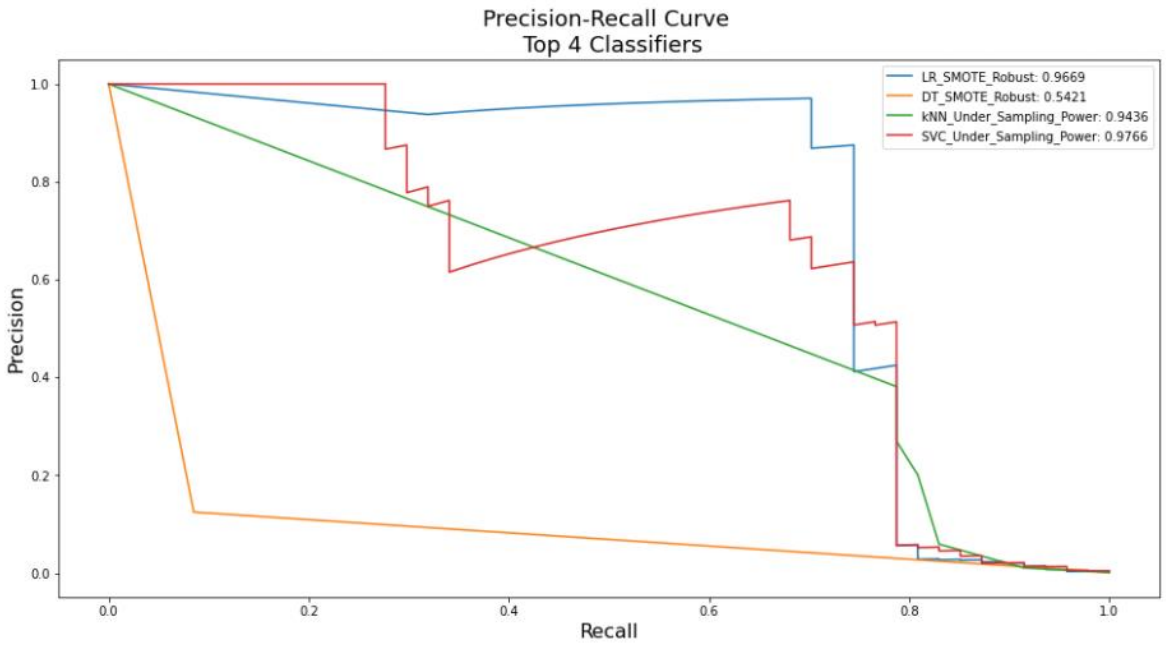


Fig: Precision Recall Curve.

Precision-Recall curve is calculated in a similar way like the ROC curve. Precision and Recall values are calculated for different thresholds and the plot is made.

When we should we use ROC vs. Precision-Recall-Curves?

- ROC curves are generally used when there is a roughly equal number of records for each class.

- Precision-Recall curves are used when there is a moderate to high-class imbalance.

The reason for this recommendation is that ROC curves present an optimistic picture of the model on datasets with a class imbalance, but here we are making the distribution equal using Random Under Sampling and SMOTE. So, ROC will be the right choice of plot.

We will mostly complete confusion matrix here, but before that let us see the confusion matrix for all the models, then we will validate the above model selection. Here we will focus on Class 1 i.e., the Fraudulent transaction class.

Confusion Matrices

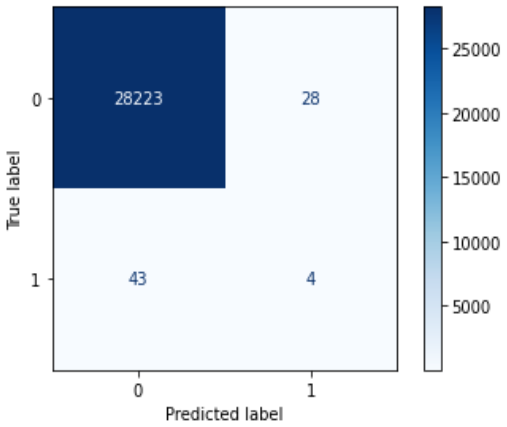
A confusion matrix is **a table that** is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known.

1. **Plotting confusion matrix of Decision Tree+ SMOTE+ Robust Scaling**

```
[102]: from sklearn.metrics import plot_confusion_matrix

       plot_confusion_matrix(DT_SMOTE_Robust, X_test_robust, y_test,cmap=plt.cm.Blues)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function
precated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in 1.2. Use one
sionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)
```

```
[102]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x24b0d6dc100>
```

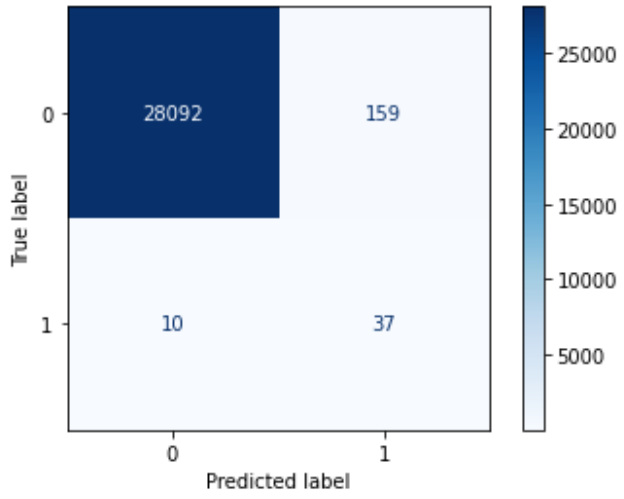


This is the confusion matrix for the decision tree and it is the worst confusion matrix among all. During the training, cross-validation, we saw a good performance for the decision tree, and that was a metric trap. Reading the matrix we can mostly say all the values of Class 1 , are misclassified and the model is extremely biased towards Class 0. Seeing the performance , we will surely drop this model and not consider it for further re-tuning.

Mostly this is self-explanatory, and we have analyzed the confusion matrix before.

2. **Plotting confusion matrix of Logistic Regression+ SMOTE+ Robust Scaling**

```
[103]: plot_confusion_matrix(LR_SMOTE_Robust, X_test_robust, y_test,cmap=plt.cm.Blues)

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: Futu
precated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be remo
sionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)

[103]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x24b0d727e80>
```

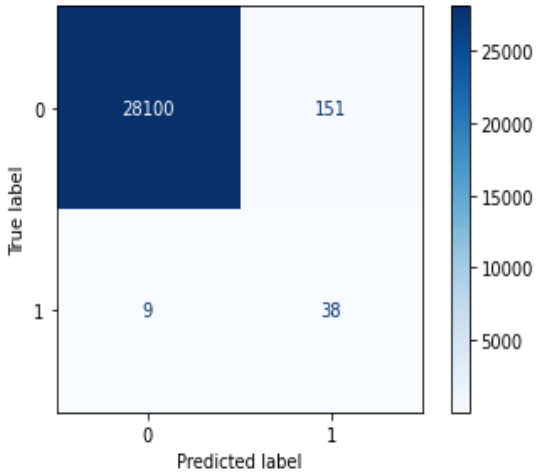


We always need to choose whether we need a low misclassification for Fraudulent Class or Low misclassification for Non-Fraudulent class. This is purely a business call which one to select.

k-NN was a similar performing model like Logistic Regression and SVC. We can see the confusion matrix below.

3. **Plotting confusion matrix K-NN + Random Under Sampling +Power Transformer.**

```
[104]: plot_confusion_matrix(kNN_Under_Sampling_Power, X_test_power, y_test,cmap=plt.cm.Blues)

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Func
precated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in 1.2. Use
sionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)

[104]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x24b15652be0>
```



For k-NN the misclassification for Class 0 is a bit high compared to Logistic and SVC. So, for this reason we will be rejecting this model too.

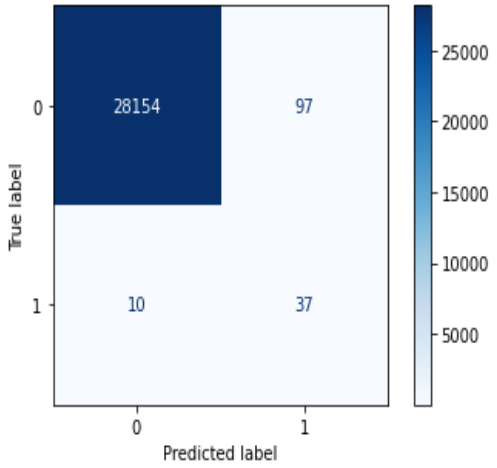
We will see the last model's confusion matrix, which we select as one of the best models.

4. **Plotting SVC + Random Under Sampling + Power Transformer.**

```
[105]: plot_confusion_matrix(SVC_Under_Sampling_Power, X_test_power, y_test, cmap=plt.cm.Blues)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function plot_
precated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be removed in 1.2. Use one of t
sionMatrixDisplay.from_predictions or ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)
```

```
[105]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x24b090c1e20>
```



Exact report we can get from Classification Report:

```
[128]: from sklearn.metrics import classification_report
print('LR_SMOTE_Robust:')
print(classification_report(y_test_robust,LR_SMOTE_Robust.predict(X_test_robust)))
print('DT_SMOTE_Robust:')
print(classification_report(y_test_robust,DT_SMOTE_Robust.predict(X_test_robust)))
print('SVC_Under_Sampling_Power:')
print(classification_report(y_test_power,DT_SMOTE_Robust.predict(X_test_power)))
print('kNN_Under_Sampling_Power:')
print(classification_report(y_test_robust,kNN_Under_Sampling_Power.predict(X_test_power)))
```

Fig: Generating a classification report.

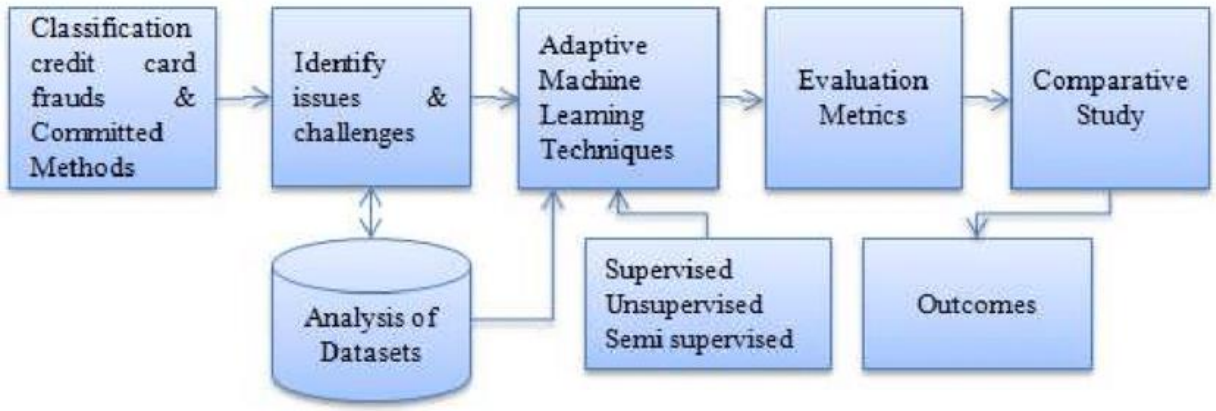
```
[106]: from sklearn.metrics import classification_report

       print('LR_SMOTE_Robust:')
       print(classification_report(y_test, LR_SMOTE_Robust.predict(X_test_robust)))

       print('DT_SMOTE_Robust:')
       print(classification_report(y_test, DT_SMOTE_Robust.predict(X_test_robust)))

       print('SVC_Under_Sampling_Power:')
       print(classification_report(y_test, SVC_Under_Sampling_Power.predict(X_test_power)))

       print('kNN_Under_Sampling_Power:')
       print(classification_report(y_test, kNN_Under_Sampling_Power.predict(X_test_power)))
```

```
LR_SMOTE_Robust:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     28251
           1       0.19      0.79      0.30        47

    accuracy                           0.99     28298
   macro avg       0.59      0.89      0.65     28298
weighted avg       1.00      0.99      1.00     28298

DT_SMOTE_Robust:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.12      0.09      0.10        47

    accuracy                           1.00     28298
   macro avg       0.56      0.54      0.55     28298
weighted avg       1.00      1.00      1.00     28298

SVC_Under_Sampling_Power:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.28      0.79      0.41        47

    accuracy                           1.00     28298
   macro avg       0.64      0.89      0.70     28298
weighted avg       1.00      1.00      1.00     28298

kNN_Under_Sampling_Power:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     28251
           1       0.20      0.81      0.32        47

    accuracy                           0.99     28298
   macro avg       0.60      0.90      0.66     28298
weighted avg       1.00      0.99      1.00     28298
```

Fig: Random Sampling Classification Report.

From the report we can visualize that we need to choose between SVC and Logistic Regression. We can think in the direction of computation time, business logic, convenience and explain-ability to select a model. SVC takes a huge time to compute and it is not advised for extremely large dataset.

Data Flow Diagram of Credit Card Fraud Detection System:

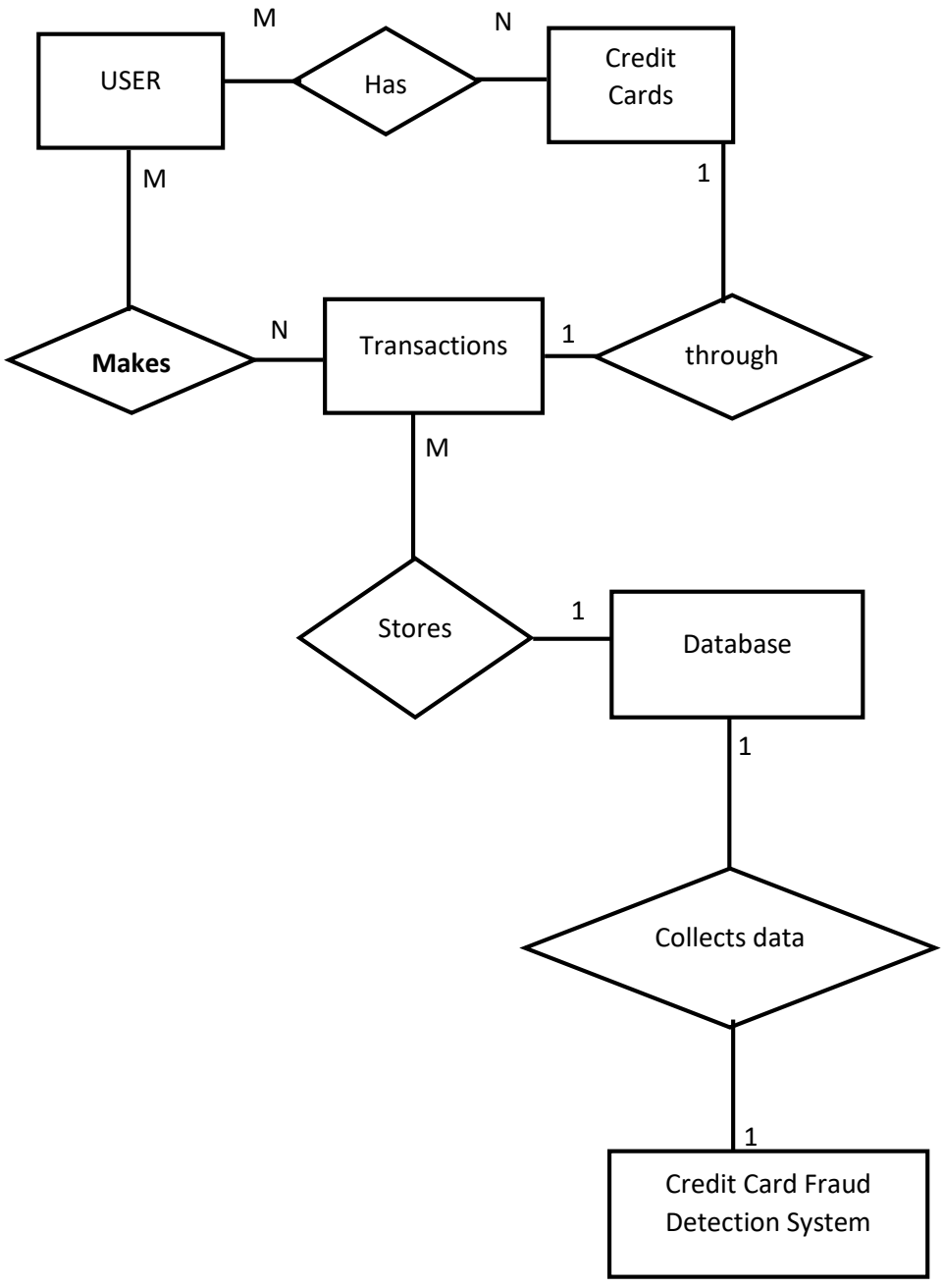


If we view the flow diagram , is just we have done classification of data , and we have applied machine learning techniques to generate report. If we see entity –relationship diagram and Activity Diagram , then we can see this how it actually works.

E-R (Entity Relationship)Diagram And Activity Diagram

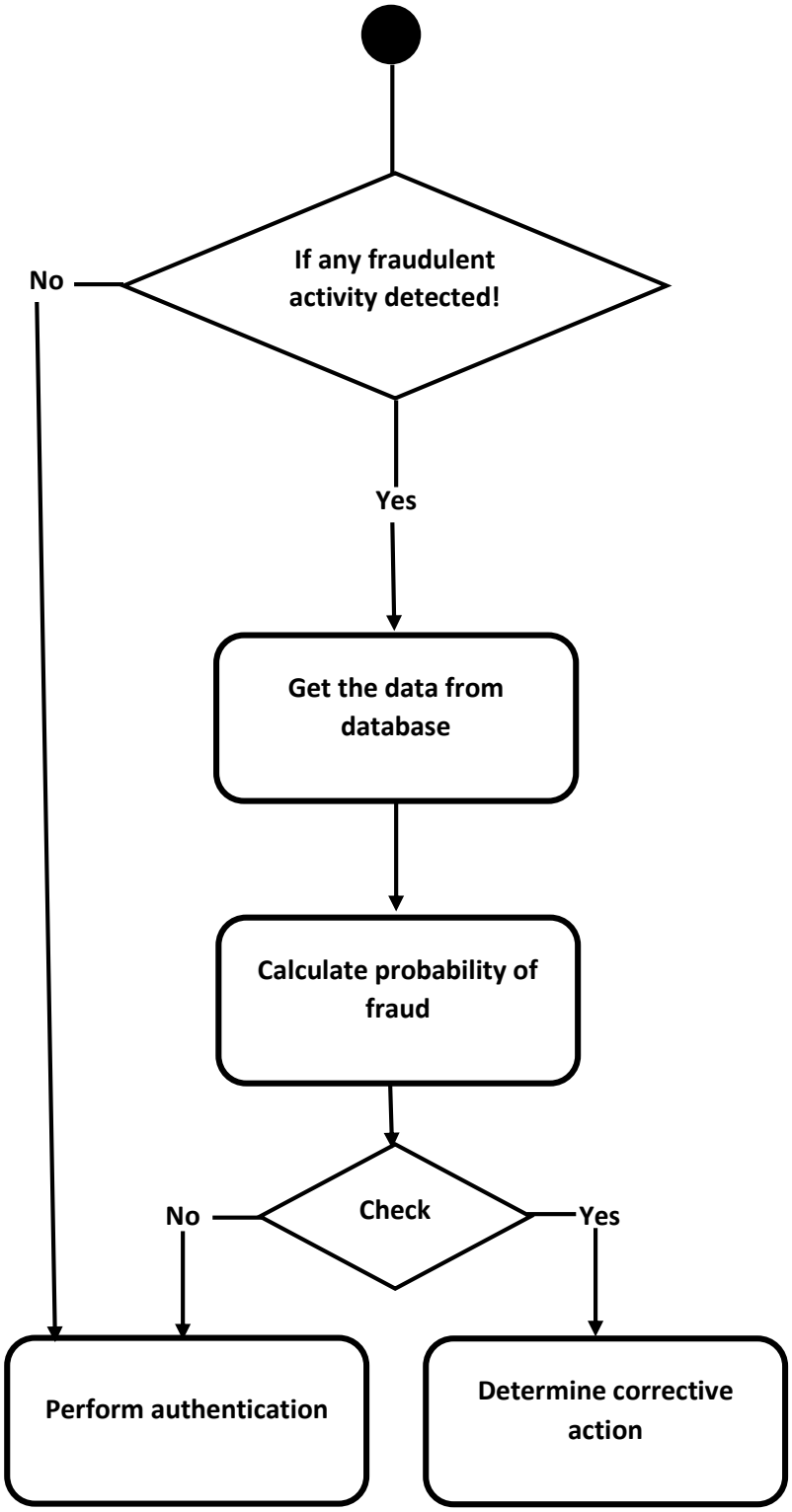
E-R Diagram

To start with the diagram, lets understand the process, a user has a Credit card make numerous transaction. Hence ‘n’ no. of users can do ‘n’ no. of transactions through a system. Such dataset is stored in database. Machine Algorithm collects an amount of data from database and analyses to raise an alarm if we get any anomaly to the system and user. This can be reflected through E-R model.



Activity Diagram

After E-R diagram comes to picture, **activity diagram** shows credit card fraud system works:



This is a type of activity that can be done when fraudulent action / transaction is detected. It depends on business entities.

Further Reading

- Sparse Matrix
- Log Transformation
- Over Sampling
    - ADASYN
    - Borderline SMOTE
    - SMOTE for Nominal and Continuous.

- Under Sampling
    - Under-sampling with ensemble learning
    - Cluster
    - Instance hardness threshold
    - One-sided selection method
    - Neighborhood cleaning rule

- Decision Tree
    - Gini Index
    - CART
    - C4.5
- SVM
    - The linear Soft-Margin Classifier
    - Different Kernel Tricks
- Classification Report

Conclusion

To end with we have not gone through building front end logic to show data acceptance, data storage and accessing data from it and analyze it is fraud or not. We have done it with a large dataset which have lastly generated reports. These reports give us the gist for the important metrics and we use it very often to compare the model fast which is the main priority of the project, if we have many models to deal with for the same data. The documentation is very extensive and detailed, as this is one of the most fields of research and application where the dataset is highly imbalanced. Financial services, Litigation, etc. are some areas where handling an imbalanced dataset is common. With this we end this documentation process.

Bibliography

1. https://en.wikipedia.org/wiki/Credit_card_fraud
2. https://en.wikipedia.org/wiki/Financial_transaction
3. https://en.wikipedia.org/wiki/Dimensionality_reduction
4. https://en.wikipedia.org/wiki/Normalization_(statistics)
5. https://en.wikipedia.org/wiki/Feature_scaling
6. https://en.wikipedia.org/wiki/Principal_component_analysis
7. https://en.wikipedia.org/wiki/Orthogonal_transformation
8. https://en.wikipedia.org/wiki/Correlation
9. https://en.wikipedia.org/wiki/Covariance_matrix
10. https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors
11. https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix
12. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeavePOut.html
13. https://www.researchgate.net/publication/334442824_Lecture_13_Validation
14. https://en.wikipedia.org/wiki/Mode_(statistics)

etc.