Credit Card Fraud Analysis

The **credit card fraud detection** using a real word dataset can be used as a best example to demonstrate machine learning process. We use data set from Kaggle website, link: - ***https://www.kaggle.com/mlg-ulb/creditcardfraud/version/3***; then use python as a programming interface and its various tools such as **seaborn**, **pandas**, **matplotlib**, **TensorFlow**, **Sci kit learn** etc. for learning the dataset and overview the dataset graphically. The columns in dataset are **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, V2, …., V28:** These are the features of the dataset.

```
import pandas as pd
df = pd.read_csv("creditcard.csv")
df.shape
```

```
(284807, 31)
```

***Fig**: Load dataset*

Total number of columns is **31**, and the total number of records is **2,84,807**. Here **read_csv()** is a function , that read a **CSV(Comma Separated Values)** and load it into a Data Frame, **Data Frame** is a structured way to store and manipulate tabular data in Python and **shape** is a attribute , which provides the number of rows and columns in the DataFrame.

```
df.columns
```

```
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')
```

***Fig**: Dataset Columns*

The **columns** attribute allows us to see the names of the columns in **Data Frame.** It has columns like “Time”, “Class”, “Amount”, “V1”,…, “V28” . All of these columns were discussed earlier.

```
df.Class.value_counts()
```

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

***Fig**: Class Frequency*

The **value_counts()** function in pandas is used to count the unique values in a specific column of a **Data Frame**. Here, **df.Class.value_counts(),**counts the occurrences of each unique value in the Class column of the **Data Frame df**. The value 0 (representing non-fraudulent transactions) appears 284,315 times. The value 1 (representing fraudulent transactions) appears 492 times.

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.countplot(x='Class', data=df)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=14)
plt.show()
```

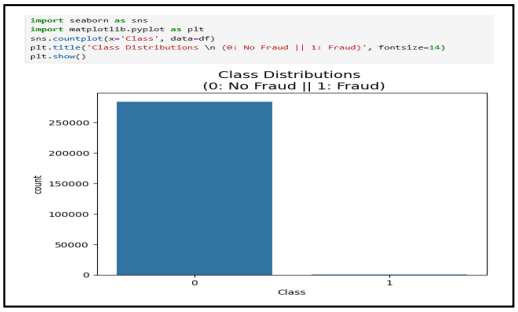


***Fig**: Class distribution*

Matplotlib is a comprehensive library provides a flexible way to create a wide range of plots and charts. Seaborn draws attractive statistical graphics, which comes with several built-in themes and colour palettes. **Seaborn** works well with Pandas Data Frames, making it easy to visualize data directly from them. Here ‘x’ axis of graph named as ‘Class’ and graph plot titled as, `Class Distribution`, where, ‘0’ stand as ‘No Fraud’ and ‘1’ stand as ‘Fraud’. We can now see the difference visually and the difference is so high that we can coin this dataset as a **highly imbalanced dataset**.

```
float(df.isnull().sum().max())
```

```
0.0
```

***Fig**: NULL count*

Hopefully we don’t have any NULL values to handle in the dataset, as we see there is ‘0’ NULL values. The function **isnull()** is used to detect missing values in the DataFrame df. It returns a DataFrame of the same shape as df, where each entry is True if the corresponding entry in **df is NaN (Not a Number, which represents missing values)** and False otherwise. After applying isnull(), calling **sum()** on the resulting **Data Frame** will sum up the True values (which are treated as 1) for each column. This gives us a Series where each entry represents the total count of missing values for that specific column in the **Data Frame**. Finally, calling **max()** on the Series returned by **sum()** will give us the maximum count of missing values across all columns. This means you will get the highest number of missing values found in any single column of your **Data Frame**.

```
df.sort_index(axis=1).head(3)
```

	Amount	Class	Time	V1	V10	V11	V12	V13	V14	V15	…	V26	V27	V28
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
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
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

***Fig**: Dataset rows*

Now let us analyse ‘Time and Amount.’

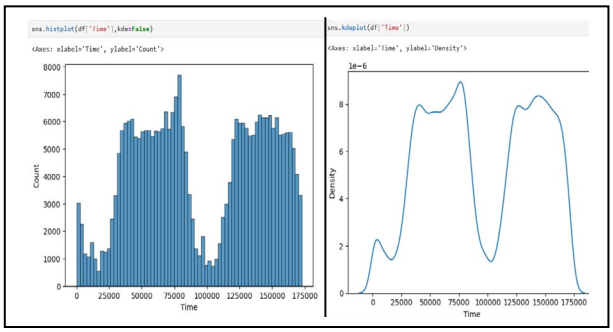


***Fig**: Bimodal `Time` distribution plot and KDE for `Time`.*

sns.histplot(df['Time'], kde=False), Here Seaborn's histplot function to create a histogram of the Time column from DataFrame df. The kde=False argument specifies that we do not want to overlay a Kernel Density Estimate (KDE) plot on top of the histogram. Therefore, the output will be a simple histogram showing the distribution of the Time values. The histogram will display the frequency of transactions that occurred at different

time intervals, which can help to visualize how transaction times are distributed across dataset**. sns.kdeplot(df['Time']):** This line uses Seaborn's kdeplot function to create a Kernel Density Estimate (KDE) plot for the Time column. A KDE plot is a smoothed version of the histogram and provides an estimate of the probability density function of the variable. It helps to visualize the distribution of the Time values in a continuous manner. The range for time feature is **[0,172792].** Similarly, we visualize **'Amount'** distribution which ranges from **[1,1000].**

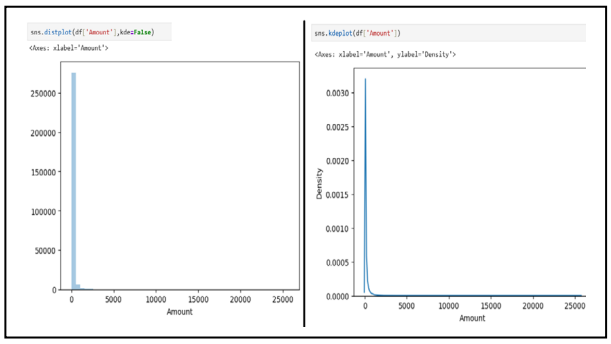


Fig*: *Bimodal `Amount` distribution plot and KDE for `Amount`

Now we need to plot which will help us to see the fraudulent transactions and the range for amounts.

```
sns.scatterplot(x='Time', y='Amount',
                data=df, hue='Class',
                size_order=[1, 0], size='Class')
```

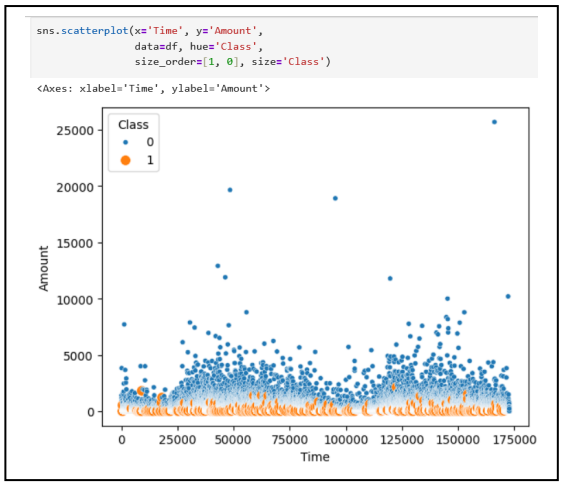


Fig*: *Class − wise ``Amount`` vs ``Time`` Scatter Plot.

Now let us check the zero-valued transactions in the Data Set.

```
df[df['Amount']==0].shape
```

```
(1825, 31)
```

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

It specifies that **1825 rows** and **31 columns** filtered in the Data Frames where Amount entered is zero.

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

```
Class
0    1798
1      27
Name: count, dtype: int64
```

Fig*: *Class − wise Zero ``Amount`` Transaction.

Here Class 0(No-Fraud) has 1798 Zero Amount transaction and Class 1(Fraud) has 27 Zero Amount transaction.

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

```
((492, 31), (284315, 31))
```

Fig*: *Shape of Fraudulent and Non − fradulent Class.

That is Class 1: Fraud transaction has 492 rows and 31 columns, while Class 0: Non-Fraud transaction has 284315 rows and 31 columns.

```
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.histplot(f_df['Amount'], kde=False, ax=ax[1], color='b')
ax[1].set_title('Fraudulent:Transaction Amount' , fontsize=14)

plt.show()
```

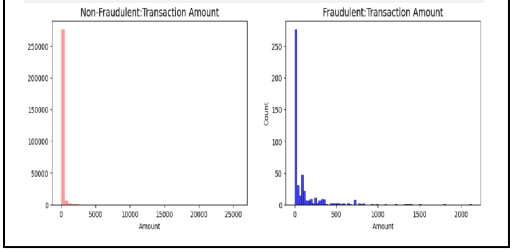


Fig*: *`Amount` Frequency Distribution for Fraudulent and Non − fraudulent Class.

```
fig, ax = plt.subplots(1,2, figsize=(14,4))

sns.histplot(n_df['Time'], kde=False, ax=ax[0], color='r')
ax[0].set_title('Non-Fraudulent:Transaction Time' , fontsize=14)

sns.histplot(f_df['Time'], kde=False, ax=ax[1], color='b')
ax[1].set_title('Fraudulent:Transaction Time' , fontsize=14)

plt.show()
```

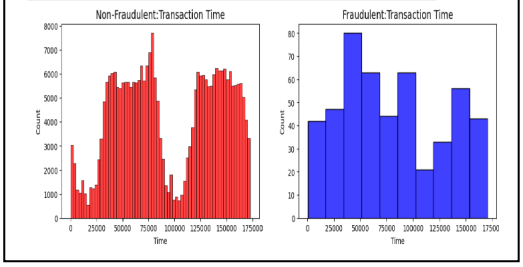


Fig*: *`Time` Frequency Distribution for Fraudulent and Non − fraudulent Class.

```
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*1)*Entire Dataset*, 2)*Non − Fraudulent and* 3)*Fraudulent Dataset.

Now removing the zero-amount transaction would not effect the "Amount" distribution and it won't change the time "Time" distribution as the size of the exception is very low. So, it will be safe to remove those records.

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

```
((282982, 31), (465, 31), (282517, 31))
```

Fig*: *Removing Zero ``Amount`` Transaction.

Now after removing all "Zero Amount" transaction, lets view the correlation of data.

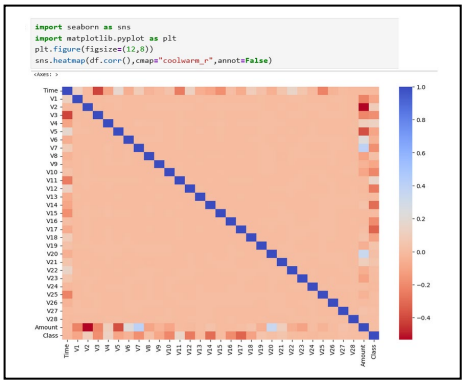


***Fig*: Correlation Heatmap**

Scaling

Scaling in machine learning refers to the process of adjusting the range of feature values in dataset so that they are on a similar scale. This is particularly important because many machine learning algorithms perform better or converge faster when features are on a relatively similar scale and close to normally distributed.

1. *Standard Scaler*

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

$$x' = \frac{x_i - \bar{x}}{\sigma}\ , where\ \bar{x} = \frac{1}{N}\sum_{i=1}^{N}(x_i)\ and\ \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

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

```
import matplotlib.pyplot as plt
import seaborn as sns
# Function to compare KDE plots
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)
    print("\n \n")
    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.subplots_adjust(wspace=0.3, hspace=0.4)
    plt.show()
```

***Fig*: Function to plot KDE.**

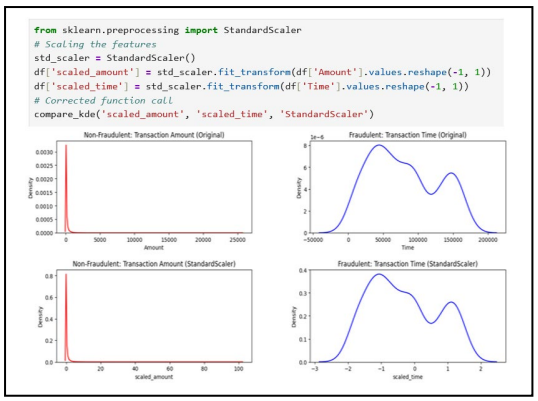


***Fig*: Standard Scaling's Features and Comparison with Original Features .**

From the above scale distribution, the values of the axis have been changed and scaled down a lot, keeping the distribution same. It is evident that the difference between the ``Time`` and ``Amount`` scale, reduced from original one.

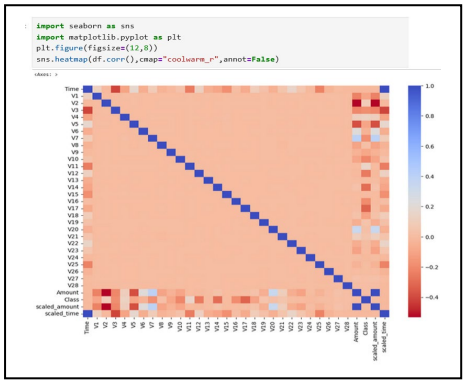


***Fig*: Correlation Heatmap after Scaling**

After Scaling , it did not make any visual change and it is expected as “Amount” and “scaled_amount” features correlation equals to one.

2. *Robust Scaling*

Robust scaling is like standard scaling, but it removes the median and scales the data according to the Interquartile Range.

$$x' = \frac{x_i - Median\ of\ x}{Q_3 - Q_1}, where\ Interquartile\ Range(IQR) = Q_3 - Q_1$$

$$Q_3\ \approx 75\ percentile\ of\ x_i\ and\ Q_1\ \approx 25\ percentile\ of\ x_i\ .$$

$$And, when\ n\ is\ odd\ i.e.x\ has\ odd\ set\ of\ numbers\ ,\ Median\ of\ x = \left(\frac{n+1}{2}\right)^{th}\ term\ and\ when\ n\ is\ even,$$

$$i.e.x\ has\ \ even\ set\ of\ number\ , Median\ of\ x = \left(\frac{\left(\frac{n}{2}\right)^{th}\ term\ + \left(\frac{n}{2}+1\right)^{th}\ term}{2}\right).$$

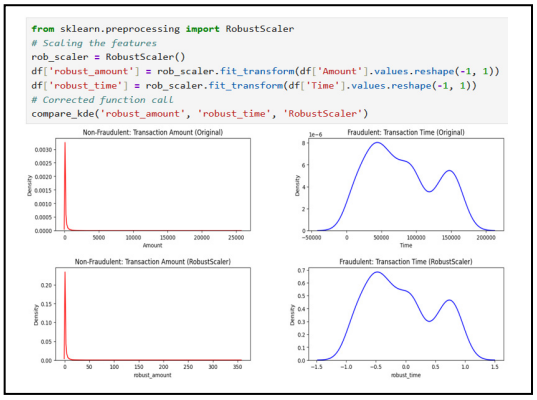


***Fig*: Robust Scaling's Features and Comparison with Original Features .**

Distribution remains same but the range and the scale are changed. Compared with both the Original and Standard Scalar, it quite similar.

3. *Power transformer*

Power transformer belongs to a family of parametric, monotonic transformations that target to data points from any distribution as close to Normal/Gaussian distribution. It has highest effect on skewed data and help to stabilize the variance of data and minimize the skewness. Due to Monotonic transformations, it preserves the rank of values among the feature. **Power transformer uses the algorithm Yeo-Jhonson transform to scale the values** with the below law:

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

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

***Fig*: Power Transformer's Features and Comparison with Original Features .**

There is a lot of change in "Amount" which is not similar to normal distribution. There is a huge change when "Amount" is compared to "Time" distribution.

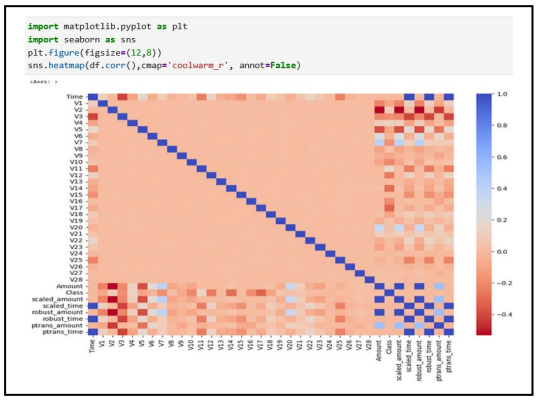


***Fig*: Correlation heatmap after applying all scaling algorithms .**

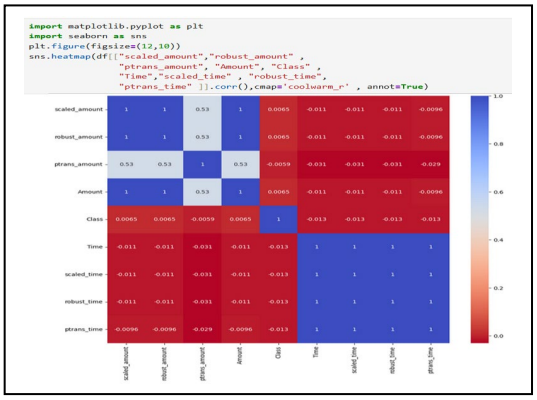
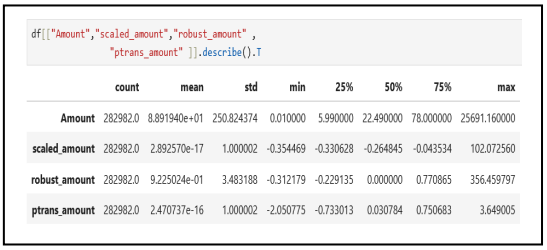


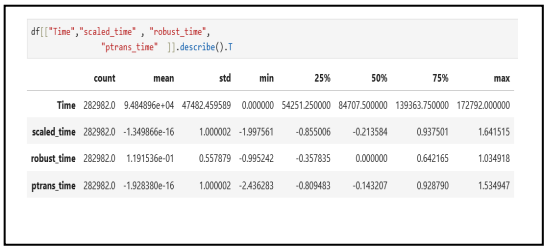
***Fig*: Description of scaled ``Amount``.**



```
df[["Amount","scaled_amount","robust_amount" ,
"ptrans_amount" ]].describe().T
```

	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
ptrans_amount	282982.0	2.470737e-16	1.000002	-2.050775	-0.733013	0.030784	0.750683	3.649005

***Fig*: Correlation for all scaled features.**



```
df[["Time","scaled_time" , "robust_time",
"ptrans_time" ]].describe().T
```

	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
ptrans_time	282982.0	-1.928380e-16	1.000002	-2.436283	-0.809483	-0.143207	0.928790	1.534947

***Fig*: Description of scaled ``Time``.**

One observation that none of the dimensionally reduced features changed its correlation with **"Time" and "Amount."** The reason is due to the imbalance. Next let us tackle the imbalance nature of the dataset.

Handling Imbalance

To tackle 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**, synthetic data points for the class is created which has low counts, here we use the test data from the original data, as if we split the data will not be correct and wrong result will be produced. But for **under-sampling,** original data frame can be used to make a sub-sample out of it. A **sub-sample** just means a part of the original dataset and creating a sub-sampled dataset is known as **Resampling**. For this dataset, we will take entire Fraudulent class and under-sample the non-fraudulent class so that the ratio is 50-50. Advantages of sub-sample is: 1. **Less prone to Overfitting** and 2. **Correlation with the class will improve as imbalance problem will get solved.**

1. *Train Test Split*

Continuing with ***train_test_split*** module of **Scikit Learn tool** and let's compare the **No Frauds** and **Frauds percentage of Train** and **Test** data with original dataset.

```
from sklearn.model_selection import train_test_split
import numpy as np

#Original percentage of No frauds and Frauds in Data Set
print('No Frauds',round(df['Class'].value_counts()[0]/len(df)*100 , 2), '% of the dataset' )
print('Frauds',round(df['Class'].value_counts()[1]/len(df)*100 , 2), '% of the dataset' )

# Separate features and target variable
X = df.drop('Class', axis=1)
y = df['Class']

#Train Test Split
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('\nTrain:')
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('\nTest:')
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**(**Traditional**).*

```
No Frauds 99.84 % of the dataset
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**: **Split Data Statistics**(**Traditional**).*

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. And we need to make the fraud percentage same or similar to the original fraud percentage. As we are dealing with an extremely small percentage of fraud data our target should be same. Hence we apply **stratified K-fold**, which is similar concept like a cross-validation K-fold technique. **K-fold cross-validation** is a technique used to evaluate the performance of machine learning models by splitting the dataset into 'k' subsets (folds), training the model on k-1 folds, and validating it on the remaining fold, repeating this process 'k' times, with each fold used as a validation set exactly once.

```
from sklearn.model_selection import StratifiedKFold
import numpy as np

#Original percentage of No frauds and Frauds in Data Set
print('No Frauds',round(df['Class'].value_counts()[0]/len(df)*100 , 2), '% of the dataset' )
print('Frauds',round(df['Class'].value_counts()[1]/len(df)*100 , 2), '% of the dataset' )

# Separate features and target variable
X = df.drop('Class', axis=1)
y = df['Class']

# creating an instance of the StratifiedKFold class describing how many folds etc.
skf = StratifiedKFold(n_splits=10, shuffle=False, random_state=None)

#Train Test Split
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('\nTrain:')
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('\nTest:')
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**).*

```
No Frauds 99.84 % of the dataset
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**).*

Note **iloc** allows us to select rows and columns from a DataFrame. StratifiedKFold's percentage is nearly same for the train,test and the original dataset i.e. for Train its exact and Test its nearly similar, hence we can go with it handling the imbalance nature of the dataset.

2. *Random Under Sampling*

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

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

```
(254684, 41)
```

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

```
Class
0    254266
1       418
Name: count, dtype: int64
```

***Fig**: **Train Dataframe and its Class Distribution**.*

```
import seaborn as sns

# 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]

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(x='Class', data=train_df_under)
```

```
Random under-sampling:
Class
0    418
1    418
Name: count, dtype: int64
<Axes: xlabel='Class', ylabel='count'>
```

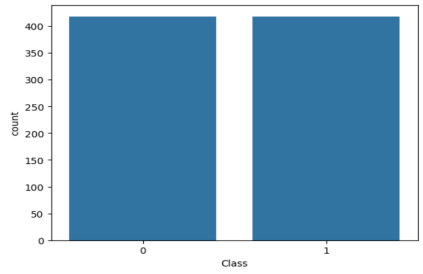


Fig**: **Dividing the Training dataset and Random Under Sampling

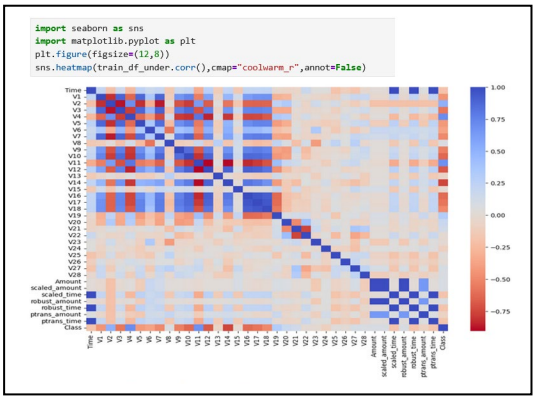


Fig: Correlation Heatmap for Random Under Sampling

The correlation uses coolwarm_r colormap, where: Dark blue represents strong positive correlation (close to +1) and Dark red represents strong negative correlation (close to -1) and Light colors (white/grayish) indicate weak or no correlation. We can print this correlation according to class.

```
print(train_df_under.corr()['Class'].sort_values(ascending=False))
```

Class	1.000000	V9	-0.537548
V4	0.708778	V3	-0.571595
V11	0.686859	V17	-0.581361
V2	0.512480	V16	-0.615428
V19	0.308157	V10	-0.636816
V20	0.175159	V12	-0.687642
V21	0.136458	V14	-0.752766
V27	0.093109	Name: Class, dtype: float64	
Amount	0.083567		
scaled_amount	0.083567		
robust_amount	0.083567		
V28	0.076365		
V8	0.066365		
V25	0.030439		
V26	0.009634		
V22	0.000037		
V23	-0.011391		
V15	-0.012241		
V13	-0.060063		
ptrans_amount	-0.076988		
V24	-0.097043		
ptrans_time	-0.190136		
robust_time	-0.193899		
scaled_time	-0.193899		
Time	-0.193899		
V6	-0.382778		
V5	-0.427769		
V1	-0.445557		
V18	-0.502470		
V7	-0.503567		

Fig: Positive and Negative Correlation according to Correlation Heatmap from Random Undersampling dataset.

Now we can see the correlated features, **Positive Correlations- *V2, V4, V11 and V19* are positively correlated and Negative Correlations – V17, V14, V12, V16 and V10 are negatively correlated.**

2. *SMOTE*

SMOTE or Synthetic Minority Over-sampling Technique over samples the minor class using the pattern of the dataset. For each instance in the minority class, SMOTE identifies its k-nearest neighbours (typically k=5). It then randomly selects one or more of these neighbours and creates synthetic samples along the line segments connecting the minority instance to its neighbours. This is done by interpolating between the feature values of the minority instance and its selected neighbours.

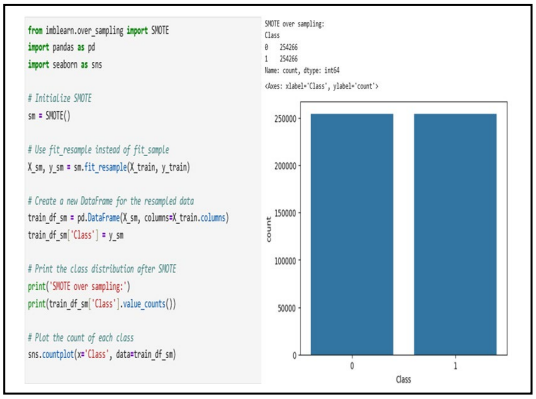


Fig: SMOTE

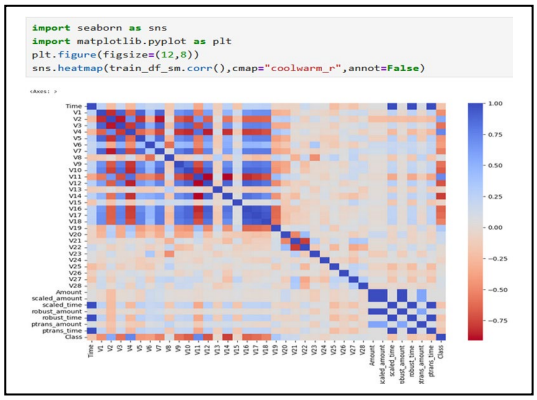


Fig: Correlation Heatmap for SMOTE dataset

```
print(train_df_sm.corr()['Class'].sort_values(ascending=False))
```

Class	1.000000	ptrans_amount	-0.099779
V4	0.737218	ptrans_time	-0.187944
V11	0.718525	scaled_time	-0.188589
V2	0.546286	robust_time	-0.188589
V19	0.338333	Time	-0.188589
V20	0.165067	V6	-0.409919
V21	0.155108	V5	-0.437015
V27	0.094649	V1	-0.477978
V28	0.087576	V18	-0.530579
V8	0.061271	V7	-0.538116
robust_amount	0.042342	V9	-0.591528
scaled_amount	0.042342	V3	-0.596795
Amount	0.042342	V17	-0.613804
V26	0.032673	V16	-0.652928
V25	0.027199	V10	-0.672759
V22	-0.003419	V12	-0.713662
V23	-0.008378	V14	-0.788222
V13	-0.031896	Name: Class, dtype: float64	
V15	-0.066022		
V24	-0.081600		

Fig: Positive and Negative Correlation according to Correlation Heatmap from SMOTE dataset.

Same positive and negative correlation we have obtained from the above dataset, hence we will continue with **random under sampling** and **smote** for machine learning techniques.

Machine Learning

1. *Data Preparation*

We will take robust scaling data in accordance with “Time” and “Amount” along with Power Transformer and strongly positive correlations: -V2, V4, V11 and V19 and strongly negative correlations-V17, V14, V12, V16 and V10 respectively.

```
#Data Preparation
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

```
#Train SMOTE-Robust sampled dataset

X_train_smote_robust = train_df_sm[col_set1]
y_train_smote_robust = train_df_sm["Class"]

#Train SMOTE-Power transformer sampled dataset
X_train_smote_power = train_df_sm[col_set2]
y_train_smote_power = train_df_sm["Class"]
```

Fig: SMOTE data preparation

```
#Train Random Undersampled-Robust  dataset

X_train_under_robust = train_df_under[col_set1]
y_train_under_robust = train_df_under["Class"]

#Train Random Undersampled-Power transformer sampled dataset
X_train_under_power = train_df_under[col_set2]
y_train_under_power = train_df_under["Class"]
```

Fig: Random Under sampled data preparation

Now all these data will be looped to train.

```
#dataset to use in a loop to train

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

2. *Ensemble Learning*

Next, we will use **Ensemble Learning method** for training as it achieves better predictive performance. Techniques used in Ensemble Learning are three one which is single model having a single weight and it completes with one iteration, number two is **Bagging** also known as **Bootstrap Aggregating** involves each model in the ensemble vote with an equal weight where each model takes in random data from the data set. This method helps to reduce variance. $\boldsymbol{D(x) = \sum \alpha d_i(x)}$, where $\boldsymbol{D(x)}$ is a strong classifier, $\boldsymbol{\alpha}$ is the strong constant for all weak classifier, $\boldsymbol{d_i(x)}$ is a weak classifier, that is computed parallelly. Example of Bagging is: **Random Forest**. **Boosting**, where multiple ML models are trained sequentially, the training of the model at a given step will have a dependency from the previously trained models using the entire dataset. It mainly focuses on **reducing the Bias** : $\boldsymbol{\hat{Y} = E(\hat{Y}) - Y}$, **i.e. it will closely match the trained data and produce low error rate and try to fit all kinds of observations.** $\boldsymbol{D(x) = \sum a_j d_j(x)}$, where $\boldsymbol{D(x)}$ is a strong classifier, $\boldsymbol{a_j}$ is the different weight for each weak classifier (which depends on the previous classifier), $\boldsymbol{d_j(x)}$ is a weak classifier that is computed serially or sequentially. Example of Boosting is: **Gradient Boosting**.

```
#Classifier libraries

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

classifiers = {
    "Random Forest Classifiers" : RandomForestClassifier(n_estimators=100, random_state=42),
    "Gradient Boosting Classifier" : GradientBoostingClassifier(n_estimators=100, random_state=42)
}
```

Fig*: *Initializing models

```
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

```
SMOTE-Robust Scaling:

Classifiers: RandomForestClassifier has a training score of 82.0 % accuracy score
Classifiers: GradientBoostingClassifier has a training score of 79.0 % accuracy score


SMOTE-Power Transformer:

Classifiers: RandomForestClassifier has a training score of 83.0 % accuracy score
Classifiers: GradientBoostingClassifier has a training score of 80.0 % accuracy score


Under Sampling-Robust Scaling:

Classifiers: RandomForestClassifier has a training score of 93.0 % accuracy score
Classifiers: GradientBoostingClassifier has a training score of 81.0 % accuracy score


Under Sampling-Power Transformer:

Classifiers: RandomForestClassifier has a training score of 93.0 % accuracy score
Classifiers: GradientBoostingClassifier has a training score of 81.0 % accuracy score
```

Fig*: *Cross Validation training score for SMOTE and Under Sampling

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

# Initialize classifiers
RFC_Under_Sampling_Power = RandomForestClassifier(n_estimators=100, random_state=42)
RFC_SMOTE_Robust = RandomForestClassifier(n_estimators=100, random_state=42)
GBC_Under_Sampling_Power = GradientBoostingClassifier(n_estimators=100, random_state=42)
GBC_SMOTE_Robust = GradientBoostingClassifier(n_estimators=100, random_state=42)

# Train classifiers
RFC_SMOTE_Robust.fit(X_train_smote_robust, y_train_smote_robust)
RFC_Under_Sampling_Power.fit(X_train_smote_power, y_train_smote_power)
GBC_SMOTE_Robust.fit(X_train_smote_robust, y_train_smote_robust)
GBC_Under_Sampling_Power.fit(X_train_smote_power, y_train_smote_power)
```

Fig*: *Training Models

```
X_test_robust = X_test[col_set1]
X_test_power = X_test[col_set2]
```

Fig*: *Preparing X Test Models Robust Scaling and Power Transformer for ROC curve

Receiver Operating Characteristics (ROC) is used here analyse the models used, it is a way to analyse the performance of a binary classifier system. ROC analysing and calculation should be done on test dataset to get an un biased assessment of models used to train dataset, as test set is the data which help us to estimate generalised performance of the used models.

```
from sklearn.metrics import roc_curve, auc

# Compute ROC curve and AUC for RandomForestClassifier for SMOTE Robust
RFCSMR_fpr, RFCSMR_tpr, RFCSMR_threshold = roc_curve(y_test, RFC_SMOTE_Robust.predict_proba(X_test_robust)[:,1])
RFCSMR_auc = auc(RFCSMR_fpr, RFCSMR_tpr)

# Compute ROC curve and AUC for RandomForestClassifier for Under Smapling
RFCUS_fpr, RFCUS_tpr, RFCUS_threshold = roc_curve(y_test, RFC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
RFCUS_auc = auc(RFCUS_fpr, RFCUS_tpr)

# Compute ROC curve and AUC for GradientBoostingClassifier for SMOTE Robust
GBCSMR_fpr, GBCSMR_tpr, GBCSMR_threshold = roc_curve(y_test, GBC_SMOTE_Robust.predict_proba(X_test_robust)[:,1])
GBCSMR_auc = auc(GBCSMR_fpr, GBCSMR_tpr)

# Compute ROC curve and AUC for RandomForestClassifier for Under Smapling
GBCUS_fpr, GBCUS_tpr, GBCUS_threshold = roc_curve(y_test, GBC_Under_Sampling_Power.predict_proba(X_test_power)[:,1])
GBCUS_auc = auc(GBCUS_fpr, GBCUS_tpr)
```

Fig*: *Generating Values for plotting ROC curve

```
import matplotlib.pyplot as plt

# Plot ROC curves
plt.figure(figsize=(16, 8))
plt.title('ROC Curve \n Top 2 Classifiers for SMOTE Robust and Power Transformer', fontsize=4)
plt.plot(RFCSMR_fpr, RFCSMR_tpr, color='blue', label=f'Random Forest SMOTE Robust (AUC = {RFCSMR_auc:.2f})')
plt.plot(GBCSMR_fpr, GBCSMR_tpr, color='red', label=f'Gradient Boosting SMOTE Robust (AUC = {GBCSMR_auc:.2f})')
plt.plot(RFCUS_fpr, RFCUS_tpr, color='yellow', label=f'Random Forest Under Sampling Power (AUC = {RFCUS_auc:.2f})')
plt.plot(GBCUS_fpr, GBCUS_tpr, color='orange', label=f'Gradient Boosting Under Sampling Power (AUC = {GBCUS_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
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

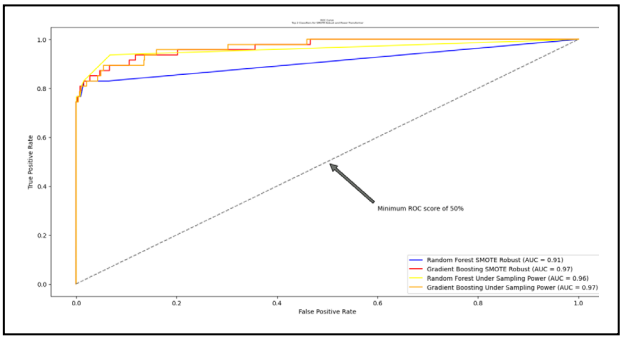


Fig*: *ROC curve

We have drawn reference line with 0.5 as a ROC score, it means that limes/models close to the reference line signifies a random guess like a coin toss where probability is 0.5 for each class. While explaining ROC curve, it was mentioned for each threshold, what is the value for true positive rate and

false-positive rate.

```
from sklearn.metrics import roc_auc_score

print("ROC Accuracy Score\n")
print('RFC_Under_Sampling_Power:' , roc_auc_score(y_test,
    RFC_Under_Sampling_Power.predict_proba(X_test_power)[:,1]))

print('GBC_Under_Sampling_Power:' , roc_auc_score(y_test,
     GBC_Under_Sampling_Power.predict_proba(X_test_power)[:,1]))

print('GBC_SMOTE_Robust:' , roc_auc_score(y_test,
      GBC_SMOTE_Robust.predict_proba(X_test_robust)[:,1]))

print('ROC_SMOTE_Robust:' , roc_auc_score(y_test,
       RFC_SMOTE_Robust.predict_proba(X_test_robust)[:,1]))
```

```
ROC Accuracy Score

RFC_Under_Sampling_Power: 0.9607598149415912
GBC_Under_Sampling_Power: 0.9705741916874342
GBC_SMOTE_Robust: 0.9696888906963941
ROC_SMOTE_Robust: 0.9084159702123141
```

Fig*: *ROC Score

We can see **Random Forest** performed against **Gradient Boost** is slightly lower but both the models act quite similar, which can be clearer while generating confusion matrix of each model.

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
import matplotlib.pyplot as plt
cm = confusion_matrix(y_test, RFC_SMOTE_Robust.predict(X_test_robust))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

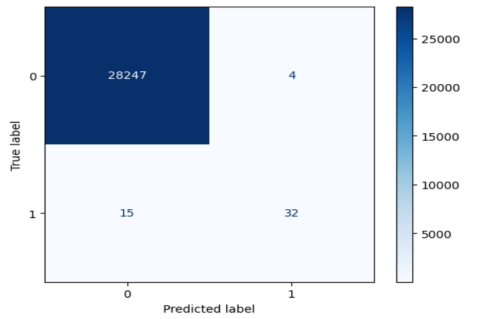


Fig*: *Random Forest + SMOTE + Robust Scaling

False Negatives(FN) are 15,i.e. 15 fraud cases were misclassified as non-fraud. False Positive(FP) are 4 i.e. 4 legitimate transactions are misclassified as Fraud. Since FN is much higher than FP, this model tends to misclassify fraud transactions as non-fraud. Also indicates that the model is biased towards Class 0, since it Favors predicting transactions as legitimate rather than fraud.

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
import matplotlib.pyplot as plt
cm = confusion_matrix(y_test, GBC_SMOTE_Robust.predict(X_test_robust))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

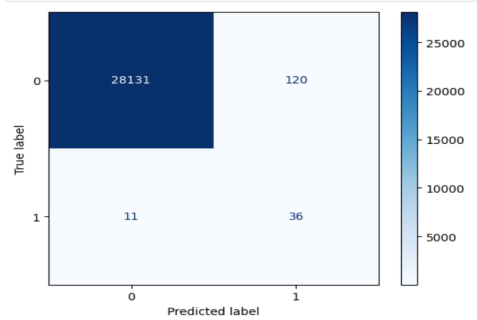


Fig*: *Gradient Boose + SMOTE + Robust Scaling

False Negatives (FN) is 11 i.e. 11 fraud cases were misclassified as non-fraud.False Positives (FP) 120 legitimate transactions were misclassified as fraud.Since FP (120) is much higher than FN (11), this model misclassifies more legitimate transactions as fraud.This indicates that the model is biased towards Class 1 (Fraud) as it Favors predicting transactions as fraudulent rather than legitimate.

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
import matplotlib.pyplot as plt
cm = confusion_matrix(y_test, RFC_Under_Sampling_Power.predict(X_test_power))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

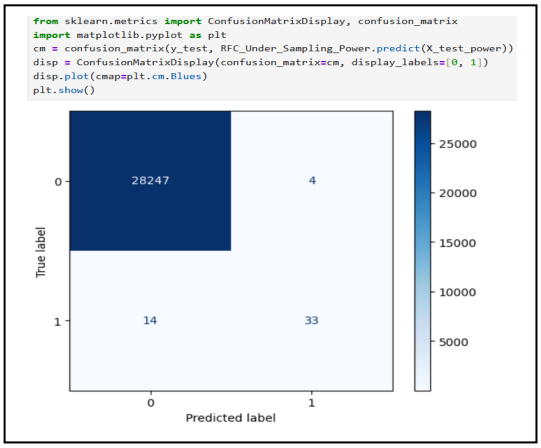


Fig*: *Random Forest + Under Sampling + Power Transformer

False Negatives (FN) is 14 i.e. 14 fraud cases were misclassified as non-fraud.False Positives (FP) is only 4 i.e. 4 legitimate transactions were misclassified as fraud.This suggests that the model is biased towards Class 0 (Non-Fraud) since it favors predicting transactions as legitimate rather than fraudulent.

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
import matplotlib.pyplot as plt
cm = confusion_matrix(y_test, GBC_Under_Sampling_Power.predict(X_test_power))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

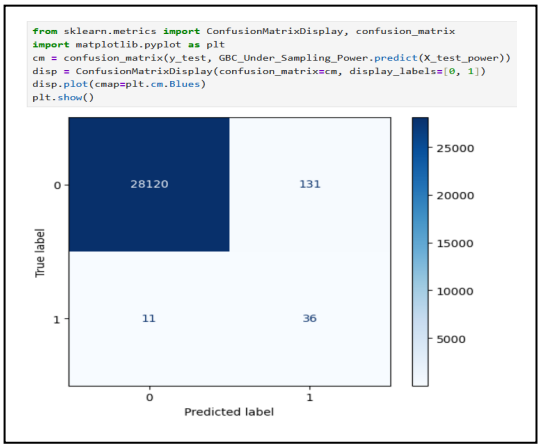


Fig*: *Gradient Boost + Under Sampling + Power Transformer

False Negatives(FN) is 11 i.e. 11 fraud cases were misclassified as non-fraud. False Positives (FP) is 131 i.e. 131 legitimate transactions were misclassified as fraud. The model has a relatively low FN (meaning it correctly identifies fraud cases better than some previous models). However, FP is higher than in previous models, meaning it erroneously flags more legitimate transactions as fraud. Hence, we will go for a **classification report**:

```
RFC_SMOTE_Robust:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.89      0.68      0.77        47

    accuracy                           1.00     28298
   macro avg       0.94      0.84      0.89     28298
weighted avg       1.00      1.00      1.00     28298

GBC_SMOTE_Robust:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.23      0.77      0.35        47

    accuracy                           1.00     28298
   macro avg       0.62      0.88      0.68     28298
weighted avg       1.00      1.00      1.00     28298

GBC_Under_Sampling_Power:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.22      0.77      0.34        47

    accuracy                           0.99     28298
   macro avg       0.61      0.88      0.67     28298
weighted avg       1.00      0.99      1.00     28298

RFC_Under_Sampling_Power:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28251
           1       0.89      0.70      0.79        47

    accuracy                           1.00     28298
   macro avg       0.95      0.85      0.89     28298
weighted avg       1.00      1.00      1.00     28298
```

Classification Report Analysis for Different Models:

RFC_SMOTE_Robust:

Class 0 (Non-Fraud) :Precision: 1.00 (All predicted non-fraud cases are correct),Recall: 1.00 (All actual non-fraud cases are detected),F1-score: 1.00 (Perfect balance between precision & recall).**Class 1 (Fraud):**Precision: 0.89 (11% of predicted fraud cases are incorrect),Recall: 0.68 (Only 68% of actual fraud cases are detected),F1-score: 0.77 (Decent but could improve recall).Result: ***High precision for fraud but recall is lower than desired. The model is biased towards class 0.***

GBC_SMOTE_Robust:

Class 0 (Non-Fraud) :Precision: 1.00, Recall: 1.00, F1-score: 1.00 (Perfect). **Class 1 (Fraud):**Precision: 0.23 (Very low, most predicted fraud cases are wrong).Recall: 0.77 (Higher than RFC_SMOTE_Robust but still misses some).F1-score: 0.35 (Very low due to poor precision).***This model heavily misclassifies legitimate transactions as fraud, making it impractical despite decent recall for fraud cases.***

GBC_Under_Sampling_Power:

Class 0 (Non-Fraud) :Precision: 1.00, Recall: 1.00, F1-score: 1.00 (Perfect) . **Class 1 (Fraud):**Precision: 0.22 (Very poor, most fraud predictions are incorrect).Recall: 0.77 (Same as GBC_SMOTE_Robust).F1-score: 0.34 (Low, making the model unreliable).***Similar to GBC_SMOTE_Robust, it has high recall but extremely poor precision for fraud cases, leading to excessive false alarms.***

RFC_Under_Sampling_Power:

Class 0 (Non-Fraud) :Precision: 1.00, Recall: 1.00, F1-score: 1.00 (Perfect). **Class 1 (Fraud):**Precision: 0.89 (Better than GBC models),Recall: 0.70 (Improved recall for fraud cases),F1-score: 0.79 (Best among all models).***Balanced model with a good trade-off between fraud detection (recall) and fraud prediction accuracy (precision). The best performing model in this set.***

Final Recommendation:

Best Model: RFC_Under_Sampling_Power,It detects 70% of fraud cases (highest recall among RFC models).It has a high precision (0.89), reducing false positives.It achieves the best F1-score (0.79) for fraud cases, meaning a good balance between catching fraud and avoiding false alarms.

Hence more or less *RFC or Rain Forest Classifier* act as best model to go with Credit Card Fraud Detection , in respect to Ensemble learning.