
```
# Install Kaggle package
!pip install kaggle

# Make a directory for the Kaggle API key
!mkdir -p ~/.kaggle

# Move the Kaggle API key to the correct location
!cp kaggle.json ~/.kaggle/

# Set the permissions for the API key
!chmod 600 ~/.kaggle/kaggle.json
```

```
Requirement already satisfied: kaggle in
/usr/local/lib/python3.10/dist-packages (1.6.17)
Requirement already satisfied: six>=1.10 in
/usr/local/lib/python3.10/dist-packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2024.8.30)
Requirement already satisfied: python-dateutil in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (from kaggle) (4.66.6)
Requirement already satisfied: python-slugify in
/usr/local/lib/python3.10/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: urllib3 in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2.2.3)
Requirement already satisfied: bleach in
/usr/local/lib/python3.10/dist-packages (from kaggle) (6.2.0)
Requirement already satisfied: webencodings in
/usr/local/lib/python3.10/dist-packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in
```

```
/usr/local/lib/python3.10/dist-packages (from python-slugify->kaggle)
(1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->kaggle)
(3.4.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.10)
cp: cannot stat 'kaggle.json': No such file or directory
chmod: cannot access '/root/.kaggle/kaggle.json': No such file or
directory
```

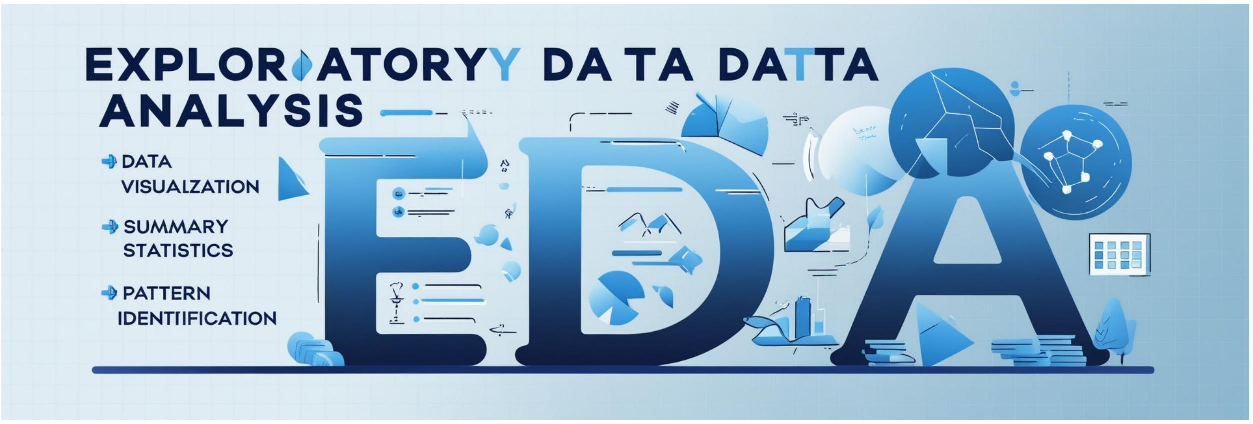
```
# Download the dataset from Kaggle
!kaggle datasets download -d mlg-ulb/creditcardfraud
```

```
Dataset URL: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
License(s): DbCL-1.0
Downloading creditcardfraud.zip to /content
 97% 64.0M/66.0M [00:02<00:00, 34.2MB/s]
100% 66.0M/66.0M [00:02<00:00, 23.3MB/s]
```

```
# Unzip the downloaded file
!unzip creditcardfraud.zip
```

```
Archive:  creditcardfraud.zip
replace creditcard.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn.ensemble import IsolationForest,RandomForestClassifier
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.model_selection import
KFold,cross_val_score,StratifiedKFold
from sklearn.metrics import
accuracy_score,precision_score,recall_score,f1_score
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay
from sklearn.metrics import
log_loss,roc_auc_score,f1_score,accuracy_score
from sklearn.preprocessing import StandardScaler

from xgboost import XGBClassifier

from warnings import simplefilter
simplefilter("ignore")

# Load the dataset
data = pd.read_csv('creditcard.csv')
```




```python
# Displaying all the columns in the dataset
pd.set_option("display.max_columns",1000)

data.head()
```

{"type":"dataframe","variable_name":"data"}

Time--> column represents the time elapsed in seconds between each transaction and the first transaction in the dataset.

Columns V1-V28--> These are the result of a PCA dimensionality reduction. Their meaning has been made obscure intentionally because of privacy reasons.

Amount--> Transaction amount.

Class--> Type of transaction (1 for fraudulent, 0 for legit).

DATA EXPLORATION

```
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
```

```
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

```python
print("rows --->",data.shape[0],"no's")
print("columns -->",data.shape[1],"no's")
```

```
rows ---> 284807 no's
columns --> 31 no's
```

```python
# Get the number of unique values for each column
unique_counts = data.nunique()
print(unique_counts)
```

```
Time      124592
V1        275663
V2        275663
V3        275663
V4        275663
V5        275663
V6        275663
V7        275663
V8        275663
V9        275663
V10       275663
V11       275663
V12       275663
V13       275663
V14       275663
V15       275663
V16       275663
V17       275663
```

```
V18       275663
V19       275663
V20       275663
V21       275663
V22       275663
V23       275663
V24       275663
V25       275663
V26       275663
V27       275663
V28       275663
Amount     32767
Class          2
dtype: int64
```

```
data.describe()

# if cat columns then
# data.describe(include='object')
```

{"type":"dataframe"}

V1 to V28: These features have very small mean values (close to zero) and have been scaled and centered around zero. Their standard deviations range between 1 and 10, showing moderate variability.

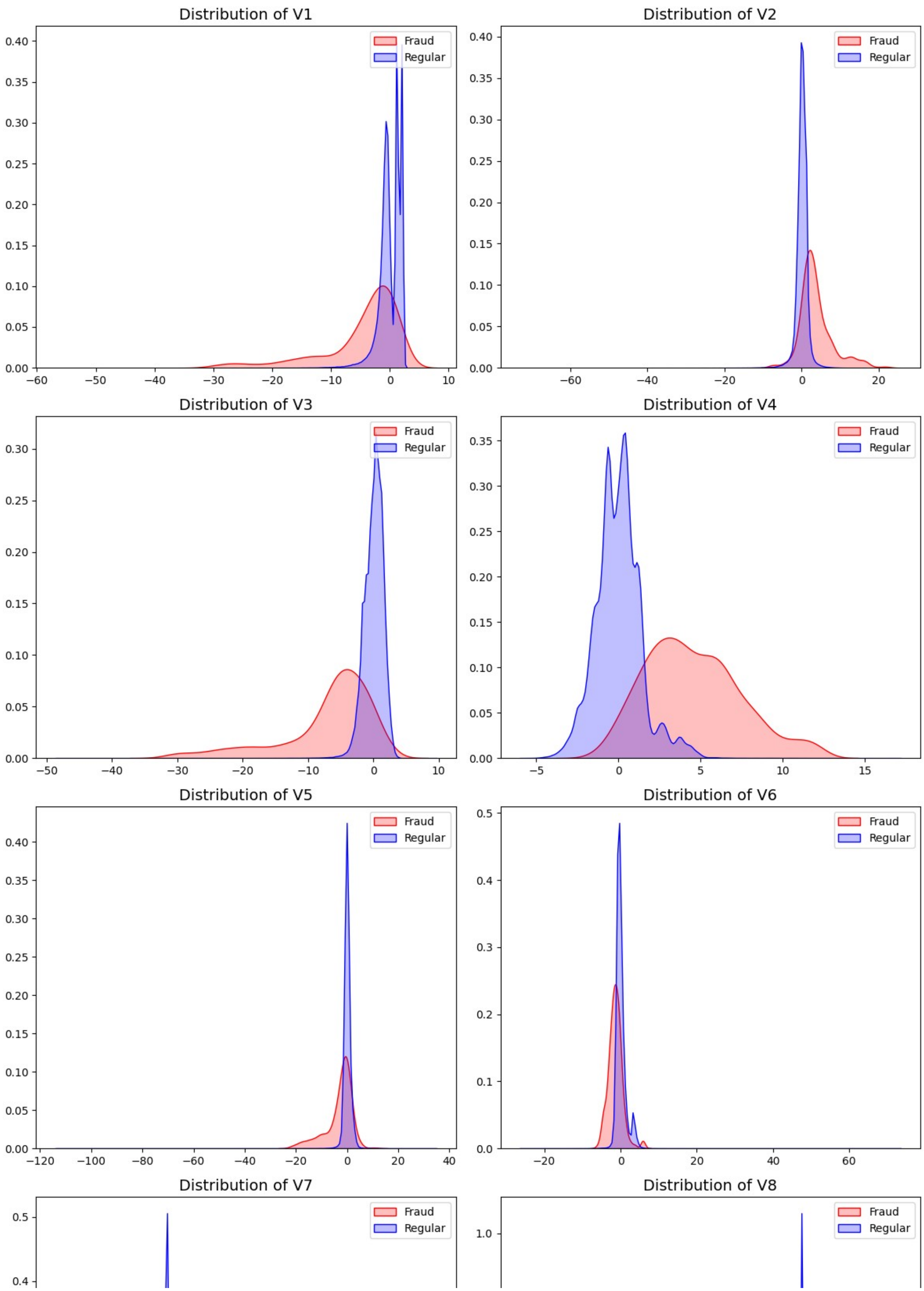
#Amount: Has a higher standard deviation than its mean, indicating a wide spread in transaction amounts (from very small to very large).

#Class: This categorical feature indicates whether a transaction is regular (0) or fraudulent (1). The mean of Class is close to zero because regular transactions are far more common than fraudulent ones

DISTRIBUTION OF V1 ... V28 Features

```
# Set up the plot grid for 28 features (V1 to V28) with 2 columns
num_features = 28
num_cols = 2
num_rows = num_features // num_cols + (num_features % num_cols > 0)  #
Calculate number of rows needed
```

```python
# Create the figure and axes
fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 5 *
num_rows))  # Adjust height based on number of rows

# Flatten the axes array for easy indexing
axes = axes.flatten()

# Plot distributions for each of V1 to V28, separating by fraud and
regular transactions
for i in range(1, num_features + 1):
    feature = f'V{i}'
#     This loop iterates over the feature indices from 1 to 28.
# feature = f'V{i}': This creates a string representing the current
feature being plotted (e.g., 'V1', 'V2', ..., 'V28').
    # Plot for fraud transactions
    sns.kdeplot(data[data.Class == 1][feature], ax=axes[i - 1],
color='red', label='Fraud', fill=True)
    # Plot for regular transactions
    sns.kdeplot(data[data.Class == 0][feature], ax=axes[i - 1],
color='blue', label='Regular', fill=True)

    axes[i - 1].set_title(f'Distribution of {feature}', fontsize=14)
    axes[i - 1].set_xlabel('')
    axes[i - 1].set_ylabel('')
    axes[i - 1].legend()

# Adjust layout to prevent overlap
plt.tight_layout()
plt.show()
```


Distribution of V1
Fraud
Regular
Distribution of V2
Fraud
Regular
Distribution of V3
Fraud
Regular
Distribution of V4
Fraud
Regular
Distribution of V5
Fraud
Regular
Distribution of V6
Fraud
Regular
Distribution of V7
Fraud
Regular
Distribution of V8
Fraud
Regular

Overview of the Plots

X-axis: Each feature V 1 V1 to V 28 V28 will have its own plot, representing the values of that feature.

Y-axis: The density of the values, which shows how likely a particular value is to occur.

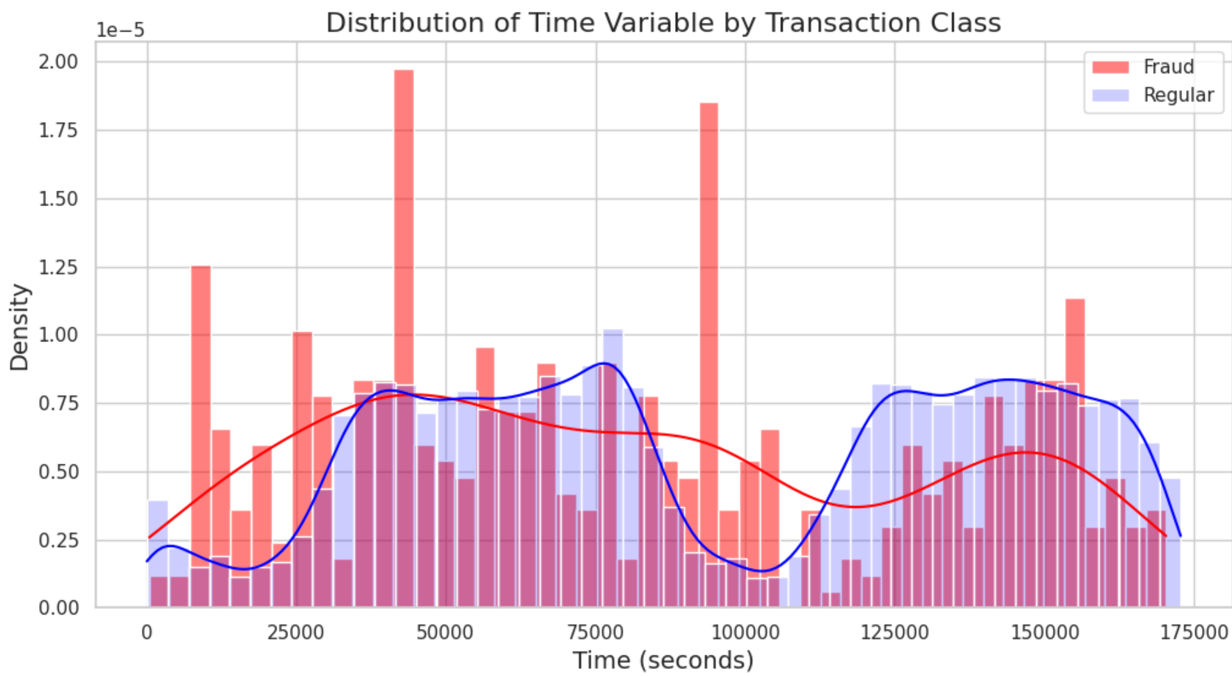
Classes:

Class 0 (Regular Transactions): This will be represented by one color (e.g., blue).

Class 1 (Fraudulent Transactions): This will be represented by another color (e.g., red).

DISTRIBUTION PLOT OF Time Variable

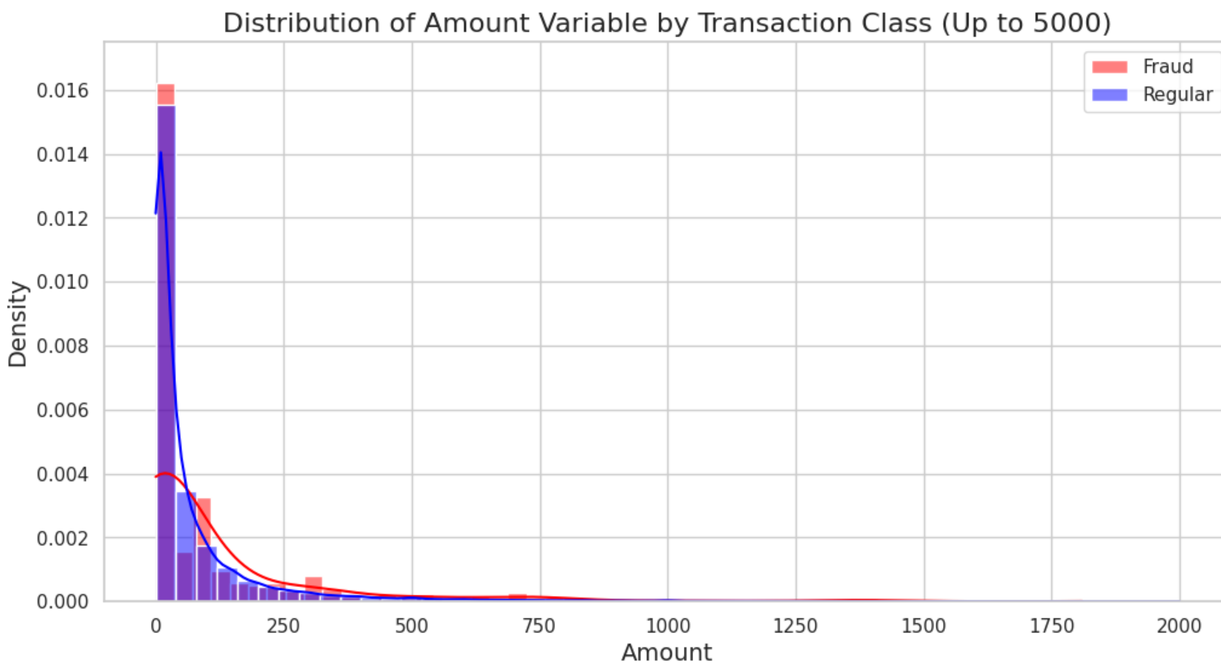
```
# Set the style for the plot
sns.set(style="whitegrid")

# Create a figure for the distribution plot
plt.figure(figsize=(12, 6))

# Plot the distribution of the 'Time' variable for fraud transactions
sns.histplot(data[data.Class == 1]['Time'], bins=50, kde=True,
color='red', stat='density', alpha=0.5, label='Fraud')

# Plot the distribution of the 'Time' variable for regular
transactions
sns.histplot(data[data.Class == 0]['Time'], bins=50, kde=True,
color='blue', stat='density', alpha=0.2, label='Regular')

# Set title and labels
plt.title('Distribution of Time Variable by Transaction Class',
fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Density', fontsize=14)

# Add legend
plt.legend()

# Show the plot
plt.show()
```



Distribution of Amount

```
# Set the style for the plot
sns.set(style="whitegrid")

# Create a figure for the distribution plot with increased height
plt.figure(figsize=(12, 6))  # Adjust height as needed

# Filter the data to include only amounts below or equal to 5000
amount_filter = data[data['Amount'] <= 2000]

# Plot the distribution of the 'Amount' variable for fraud
transactions
sns.histplot(amount_filter[amount_filter.Class == 1]['Amount'],
bins=50, kde=True, color='red', stat='density', alpha=0.5,
label='Fraud')

# Plot the distribution of the 'Amount' variable for regular
transactions
sns.histplot(amount_filter[amount_filter.Class == 0]['Amount'],
bins=50, kde=True, color='blue', stat='density', alpha=0.5,
label='Regular')

# Set title and labels
plt.title('Distribution of Amount Variable by Transaction Class (Up to
5000)', fontsize=16)
plt.xlabel('Amount', fontsize=14)
plt.ylabel('Density', fontsize=14)
```

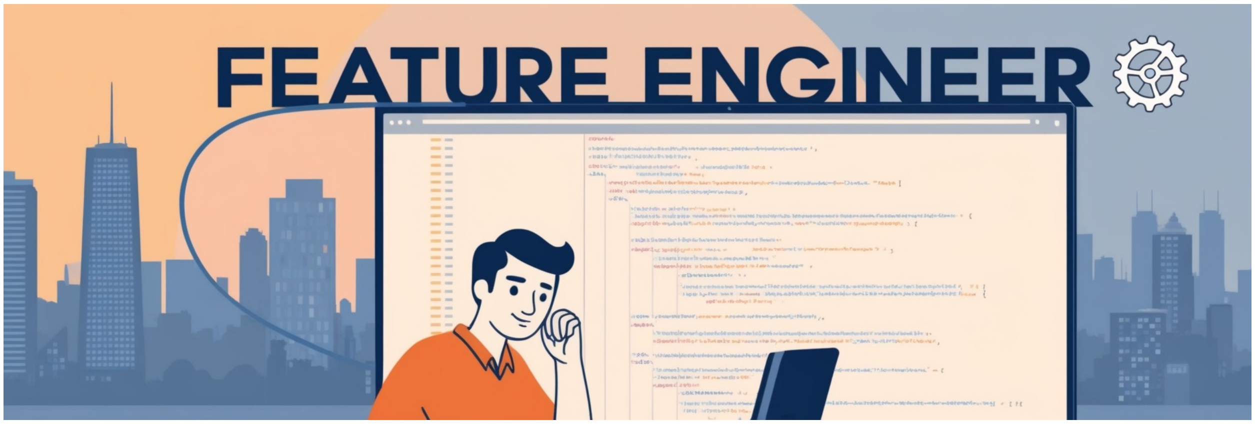
```
# Adjust y-axis limits if needed
plt.ylim(0, 0.0175)  # Adjust based on the data

# Add legend
plt.legend()

# Show the plot
plt.show()
```



Red (Fraud Transactions): This color is used to indicate transactions that are classified as fraudulent (where Class == 1). These transactions are typically of particular interest when analyzing credit card fraud data because they represent instances of fraudulent activity.

Blue (Regular Transactions): This color represents transactions that are classified as legitimate or regular (where Class == 0). These are the majority of transactions in the dataset and serve as a baseline for comparison against fraudulent transactions.

we visualized ->The Amount distribution for frauds is peaked on small quantities of money.



Duplicate Values

```
print('There are {} duplicate values in regular transactions out of {}'.format(data[data['Class'] == 0].duplicated().sum(),data[data['Class'] == 0].shape[0]))
print('There are {} duplicate values in fraudulent transactions out of {}'.format(data[data['Class'] == 1].duplicated().sum(),data[data['Class'] == 1].shape[0]))
```

```
There are 1062 duplicate values in regular transactions out of 284315
There are 19 duplicate values in fraudulent transactions out of 492
```

inplace=True: This modifies the DataFrame in place. The original DataFrame is updated, and no new DataFrame is returned. As a result, you do not need to assign the result to a new variable or even to the same variable.

inplace=False (default behavior): This will return a new DataFrame with the duplicates removed, leaving the original DataFrame unchanged. You would typically assign the result to a new variable or overwrite the existing variable.

```
print('No. of rows before dropping duplicates: {}.'.format(len(data)))

data.drop_duplicates(inplace=True)

print('No. of rows after dropping duplicates: {}.'.format(len(data)))
```

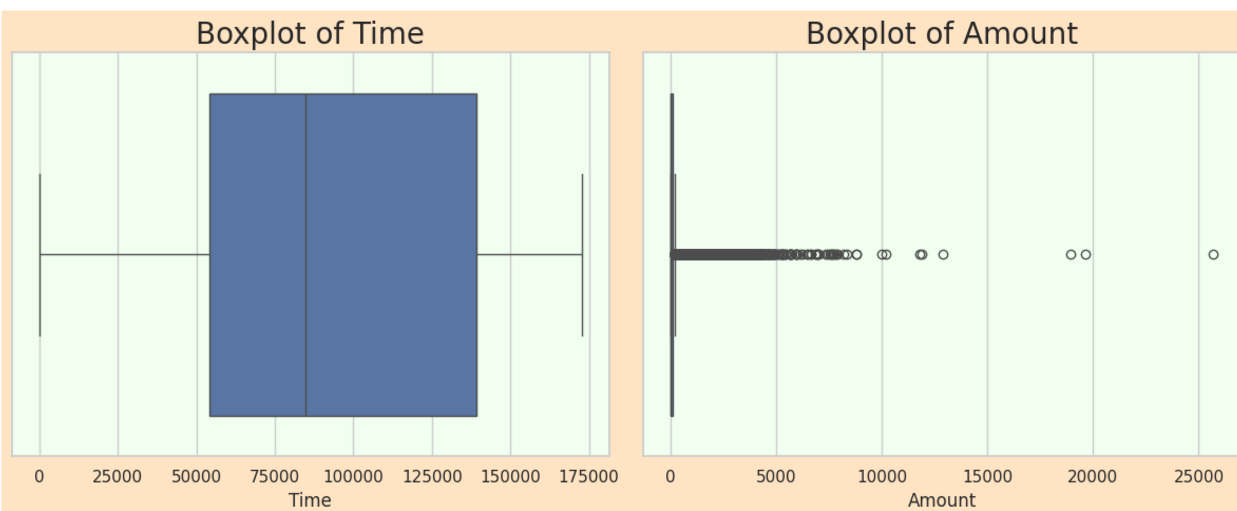
```
No. of rows before dropping duplicates: 284807.
No. of rows after dropping duplicates: 283726.
```

here few duplicate rows dropped

HANDLING OUTLIERS

outlier can be extremely high or low values compared to the other observations and can be caused by measurement errors, natural variations in the data, or even unexpected discoveries.that leads lower performance so we deal with it...

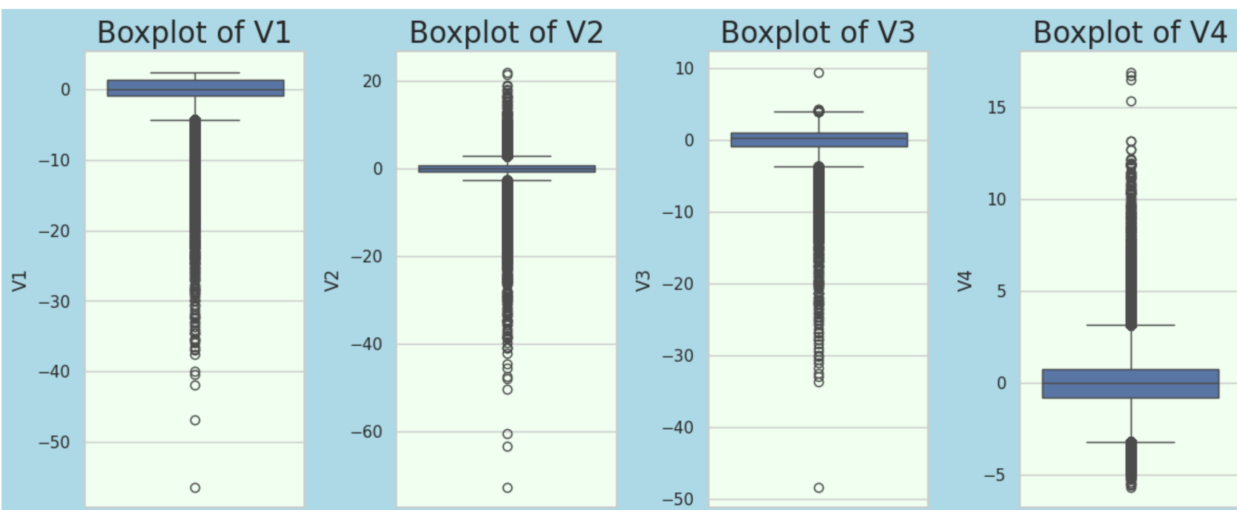
```
# Create subplots
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))

# Boxplot for Time
sns.boxplot(x=data['Time'], ax=ax1)
ax1.set_title('Boxplot of Time', fontsize=20)
ax1.set_facecolor('honeydew')

# Boxplot for Amount
sns.boxplot(x=data['Amount'], ax=ax2)
ax2.set_title('Boxplot of Amount', fontsize=20)
ax2.set_facecolor('honeydew')

# Adjust layout
plt.tight_layout()

# Set figure background color
fig.set_facecolor('bisque')

# Show plot
plt.show()
```

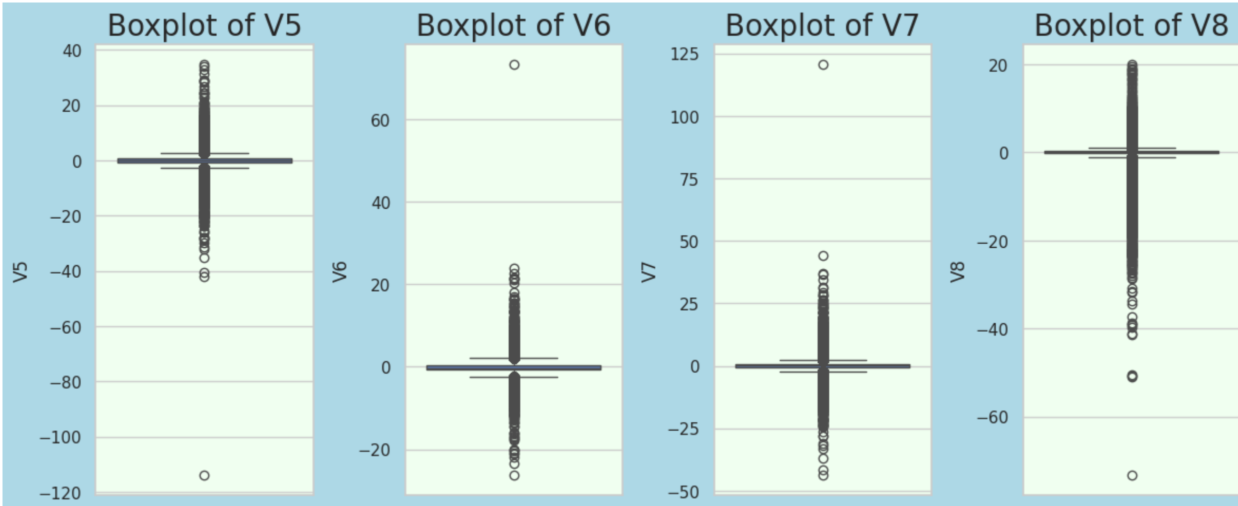


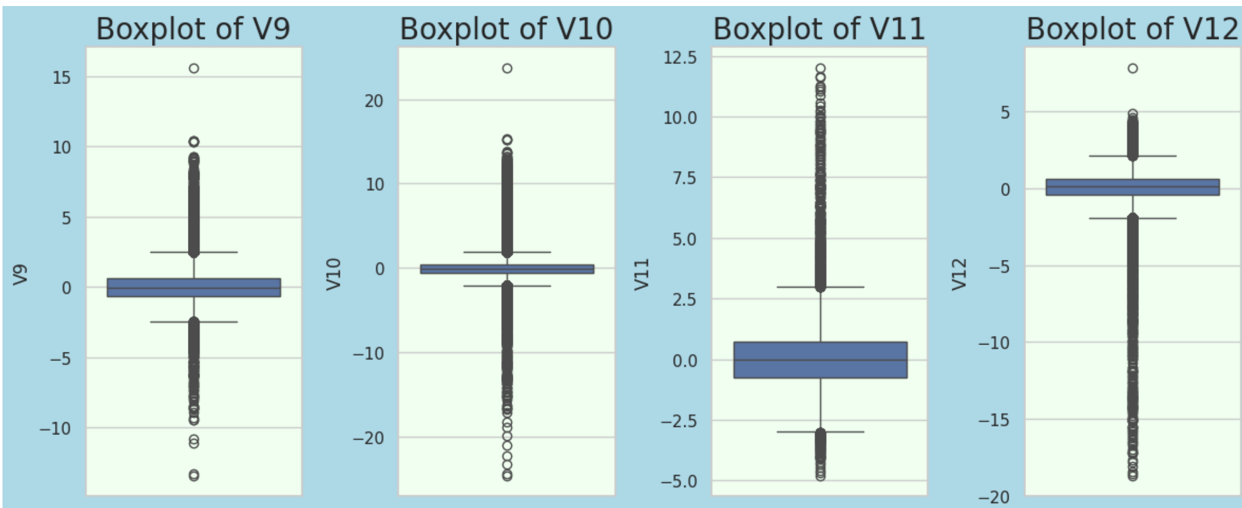
here no outlier in "time" but too many outlier in "amount"

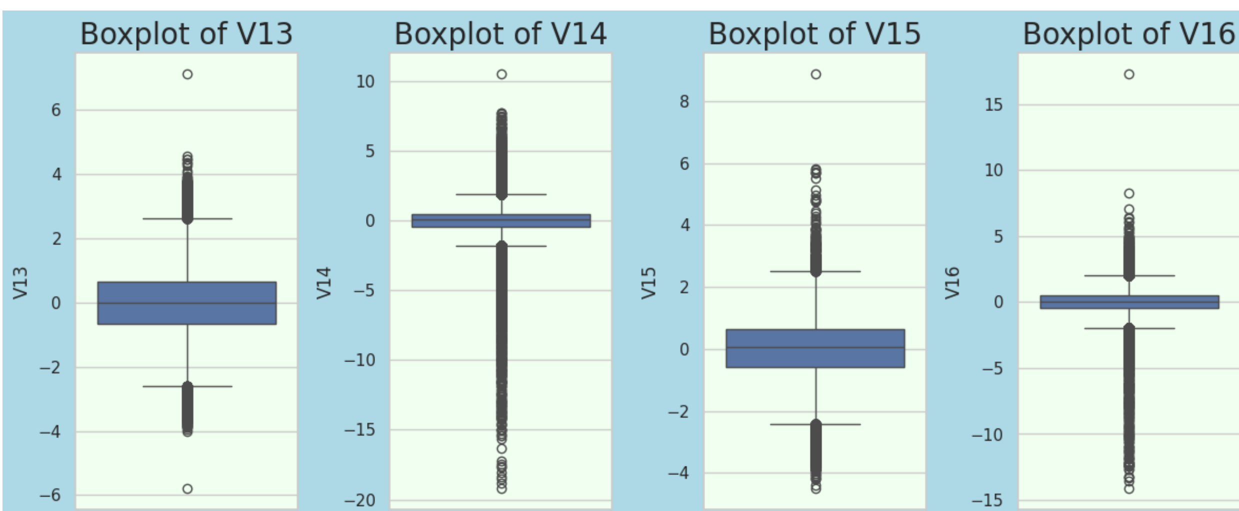
```
# Select only the V1-V28 features
features = list(data.columns.values)
del features[0]
```

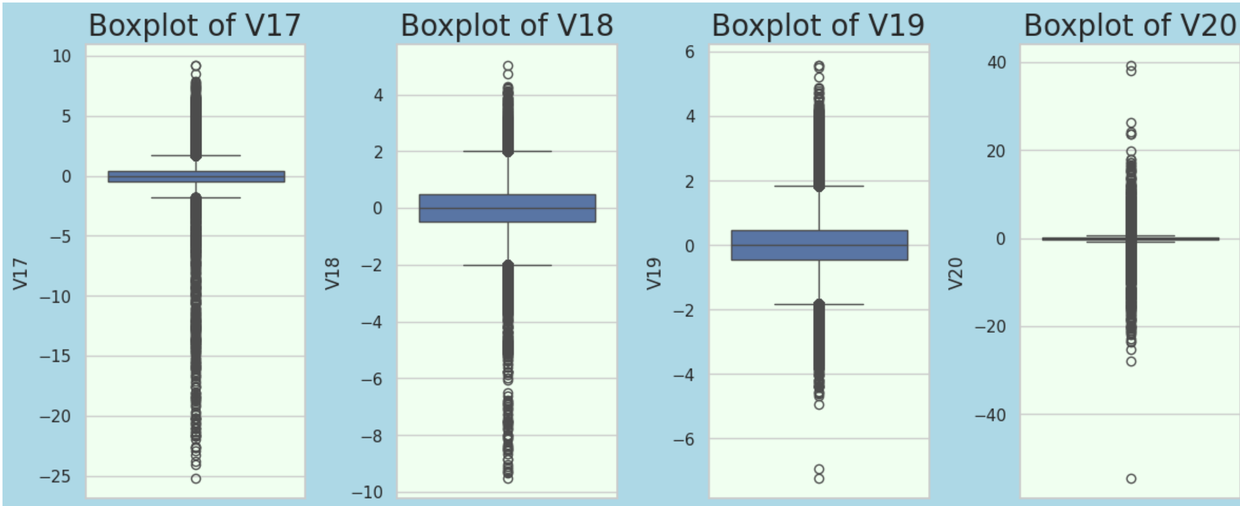
```
del features[28]
del features[28]

for i in range(7):#i 0 to 6
  fig,
(ax1,ax2,ax3,ax4)=plt.subplots(ncols=4,figsize=(12,5))#wide=12,tall=5
  #plot diagram by 4 rows
  ax1=sns.boxplot(data[features[i*4]],ax=ax1)
  ax1.set_title('Boxplot of '+str(features[i*4]),fontsize=20)
  ax1.set_facecolor('honeydew')

  ax2=sns.boxplot(data[features[i*4+1]],ax=ax2)
  ax2.set_title('Boxplot of '+str(features[i*4+1]),fontsize=20)
  ax2.set_facecolor('honeydew')

  ax3=sns.boxplot(data[features[i*4+2]],ax=ax3)
  ax3.set_title('Boxplot of '+str(features[i*4+2]),fontsize=20)
  ax3.set_facecolor('honeydew')

  ax4=sns.boxplot(data[features[i*4+3]],ax=ax4)
  ax4.set_title('Boxplot of '+str(features[i*4+3]),fontsize=20)
  ax4.set_facecolor('honeydew')

  plt.tight_layout()
  fig.set_facecolor('#ADD8E6')
```

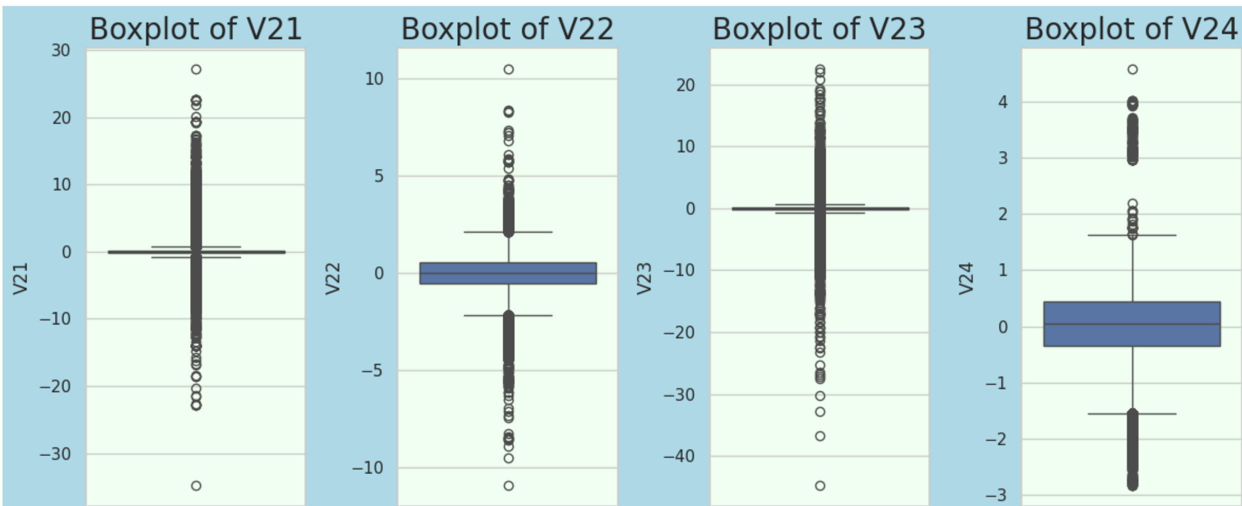


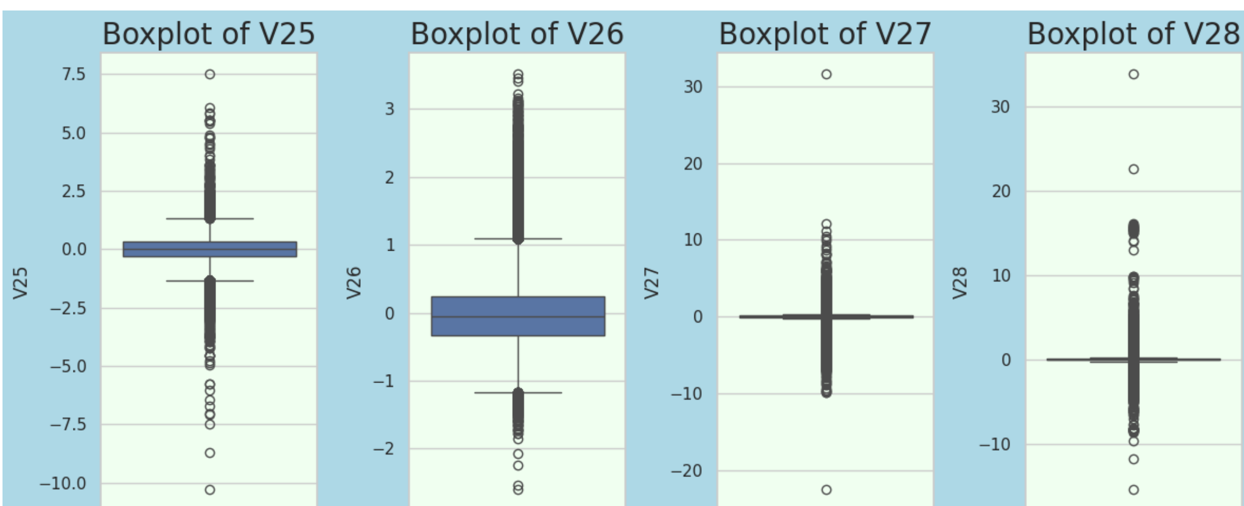

Boxplot of V5
Boxplot of V6
Boxplot of V7
Boxplot of V8
V5
V6
V7
V8
40
20
0
−20
−40
−60
−80
−100
−120
60
40
20
0
−20
125
100
75
50
25
0
−25
−50
20
0
−20
−40
−60


Boxplot of V9
Boxplot of V10
Boxplot of V11
Boxplot of V12
V9
V10
V11
V12
15
10
5
0
−5
−10
20
10
0
−10
−20
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
5
0
−5
−10
−15
−20


Boxplot of V13
Boxplot of V14
Boxplot of V15
Boxplot of V16
V13
V14
V15
V16
6
4
2
0
−2
−4
−6
10
5
0
−5
−10
−15
−20
8
6
4
2
0
−2
−4
15
10
5
0
−5
−10
−15


Boxplot of V17
Boxplot of V18
Boxplot of V19
Boxplot of V20
V17
V18
V19
V20


Boxplot of V21
Boxplot of V22
Boxplot of V23
Boxplot of V24
V21
V22
V23
V24


Boxplot of V25
Boxplot of V26
Boxplot of V27
Boxplot of V28
V25
V26
V27
V28

```
for column in features:
  #check how much skew
  print(f"Skewness for {column}: {data[column].skew()}")
```

```
Skewness for V1: -3.273271248440309
Skewness for V2: -4.6951619005404694
Skewness for V3: -2.1519839570997124
Skewness for V4: 0.6715041706728241
Skewness for V5: -2.414079246966253
Skewness for V6: 1.829880383771521
Skewness for V7: 2.890271192715498
Skewness for V8: -8.310970330052545
Skewness for V9: 0.5376630534496958
Skewness for V10: 1.2529670787468168
Skewness for V11: 0.34407419325686267
Skewness for V12: -2.1990082816149954
Skewness for V13: 0.06429340464018111
Skewness for V14: -1.9188037137586451
Skewness for V15: -0.3096590822936595
Skewness for V16: -1.0511614715174662
Skewness for V17: -3.690497194148406
Skewness for V18: -0.24866145737243997
Skewness for V19: 0.1083118109324772
Skewness for V20: -2.0431210560273323
Skewness for V21: 2.820033113572543
Skewness for V22: -0.18232972797521269
Skewness for V23: -5.867220791006341
Skewness for V24: -0.5521292366718961
Skewness for V25: -0.41574386205469593
Skewness for V26: 0.5802923172348093
Skewness for V27: -0.7538039138186547
Skewness for V28: 11.555115084196773
```

-ve indicates left skew

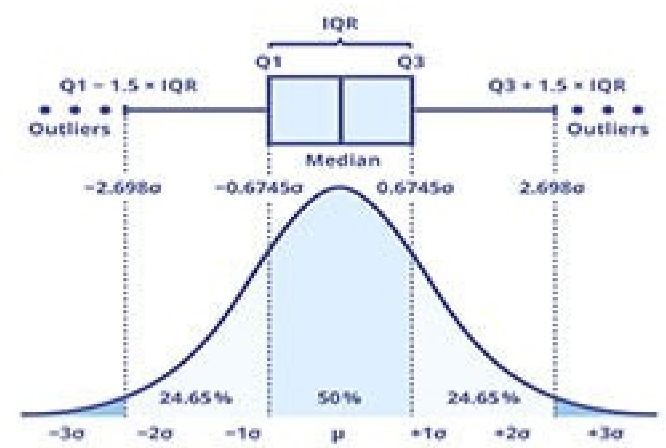
+ve indicates right skew

#**CAPING OUTLIER**

```
def outlier_imputer(data,features):

    data_out = data.copy()

    for column in features:

        # First define the first and third quartiles
        Q1 = data_out[column].quantile(0.25)
        Q3 = data_out[column].quantile(0.75)
        # Define the inter-quartile range
        IQR = Q3 - Q1
```

```
        # ... and the lower/higher threshold values
        lowerL = Q1 - 1.5 * IQR
        higherL = Q3 + 1.5 * IQR

        # Impute 'left' outliers
        data_out.loc[data_out[column] < lowerL,column] = lowerL
        #which rows value lower than lowerL that capped
        # Impute 'right' outliers
        data_out.loc[data_out[column] > higherL,column] = higherL
        #data_out.loc[mask, column] locates the rows in column where
data_out[column] > higherL is True.-->[rows,columns]

    return data_out

#only made because of extracting features
data2 = data.drop('Class',axis=1)

feats = list(data2.columns.values) #list of column names

capped_data = outlier_imputer(data,feats)

pd.set_option("display.max_columns",1000)

capped_data.head()

{"type":"dataframe","variable_name":"capped_data"}
```

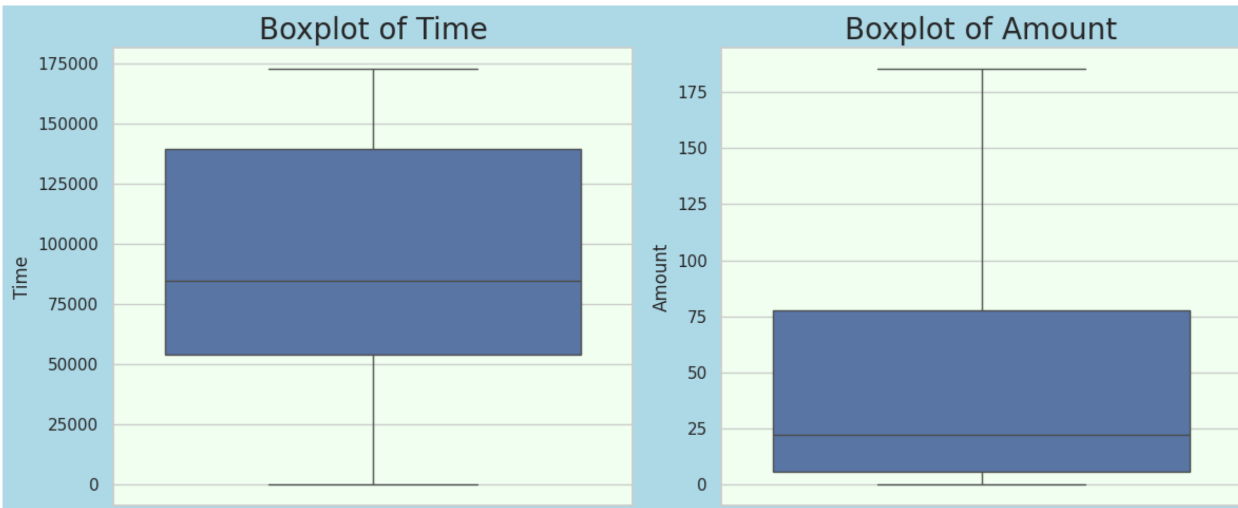
the outliers have been imputed and the extreme values have been set to either $Q1 - 1.5 * IQR$ (lower threshold) or to $Q3 + 1.5 * IQR$ (higher threshold), where $Q1$ and $Q3$ are the first and third quartiles and IQR is the so called interquartile range



AFTER CAPPING

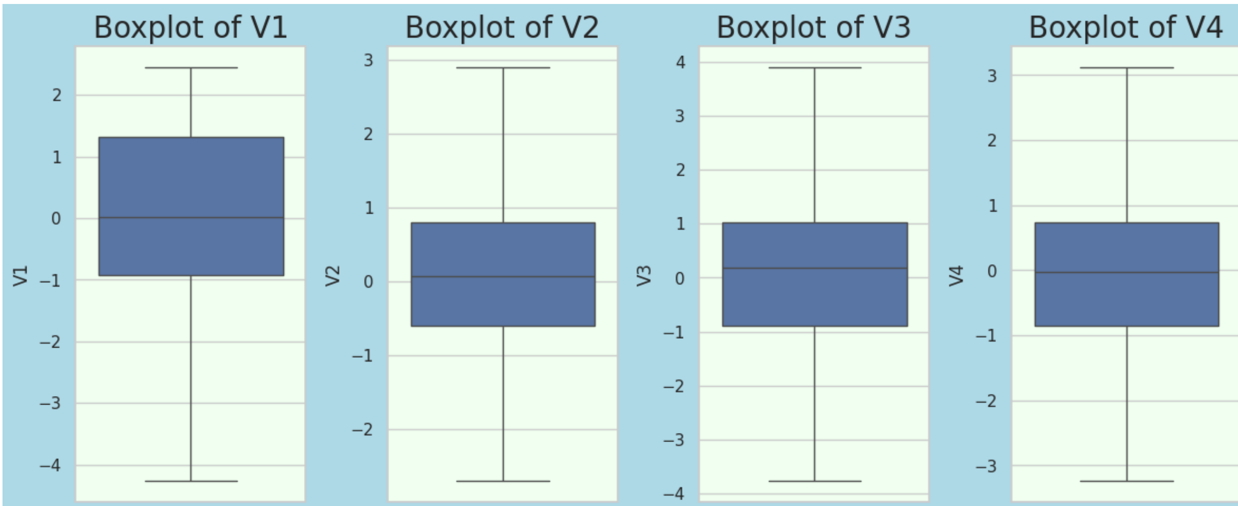
```
fig,(ax1,ax2) = plt.subplots(ncols=2,figsize=(12,5))
```

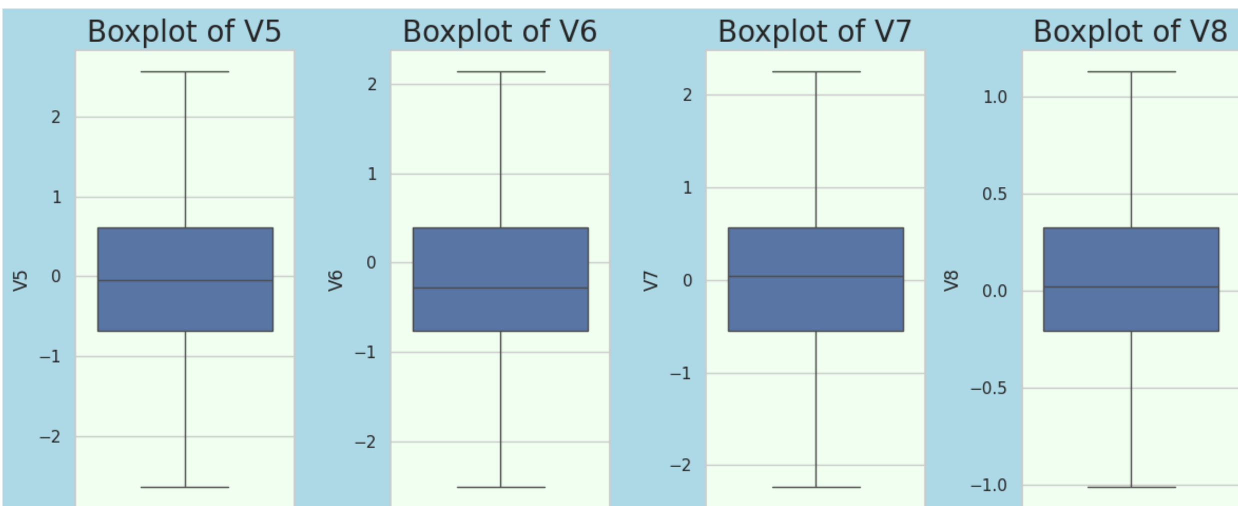
```
ax1 = sns.boxplot(capped_data['Time'],ax=ax1)
ax1.set_title('Boxplot of Time',fontsize=20)
ax1.set_facecolor('honeydew')
ax2 = sns.boxplot(capped_data['Amount'],ax=ax2)
ax2.set_title('Boxplot of Amount',fontsize=20)
ax2.set_facecolor('honeydew')


plt.tight_layout()
fig.set_facecolor('#ADD8E6')
```

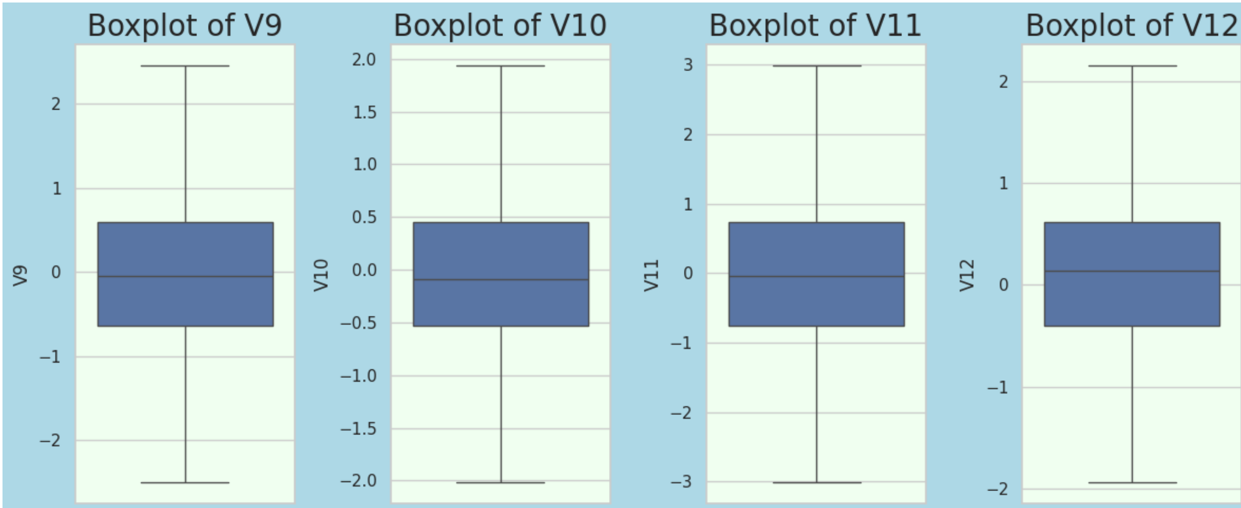


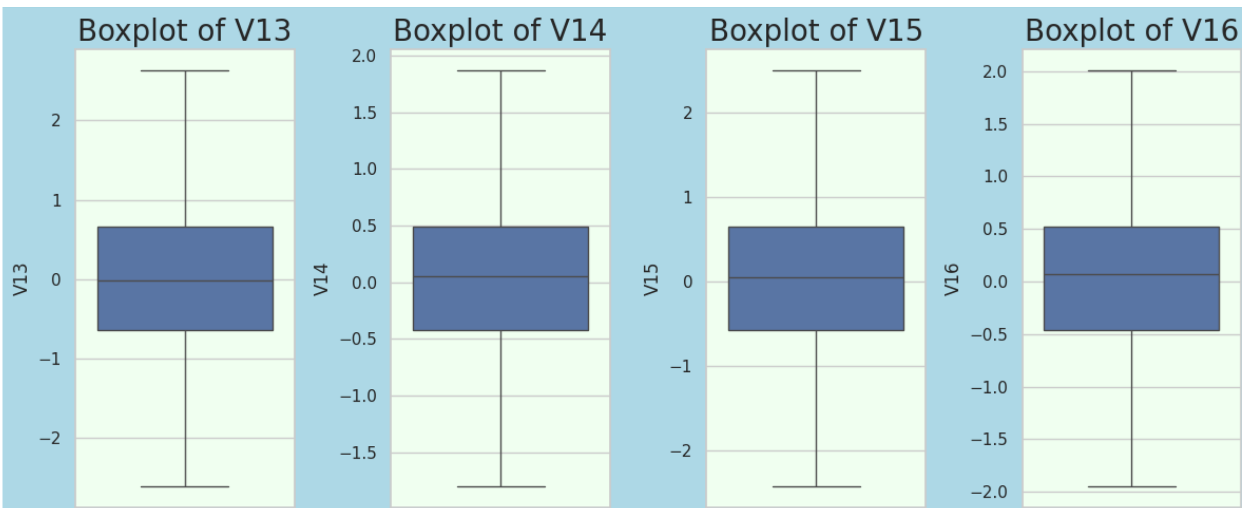

```
# Select only the V1-V28 features
features_c = list(capped_data.columns.values)
del features_c[0]
del features_c[28]
del features_c[28]

for i in range(7):#i 0 to 6
  fig,
(ax1,ax2,ax3,ax4)=plt.subplots(ncols=4,figsize=(12,5))#wide=12,tall=5
  #plot diagram by 4 rows
  ax1=sns.boxplot(capped_data[features_c[i*4]],ax=ax1)
  ax1.set_title('Boxplot of '+str(features_c[i*4]),fontsize=20)
  ax1.set_facecolor('honeydew')

  ax2=sns.boxplot(capped_data[features_c[i*4+1]],ax=ax2)
  ax2.set_title('Boxplot of '+str(features_c[i*4+1]),fontsize=20)
  ax2.set_facecolor('honeydew')

  ax3=sns.boxplot(capped_data[features_c[i*4+2]],ax=ax3)
  ax3.set_title('Boxplot of '+str(features_c[i*4+2]),fontsize=20)
  ax3.set_facecolor('honeydew')

  ax4=sns.boxplot(capped_data[features_c[i*4+3]],ax=ax4)
```

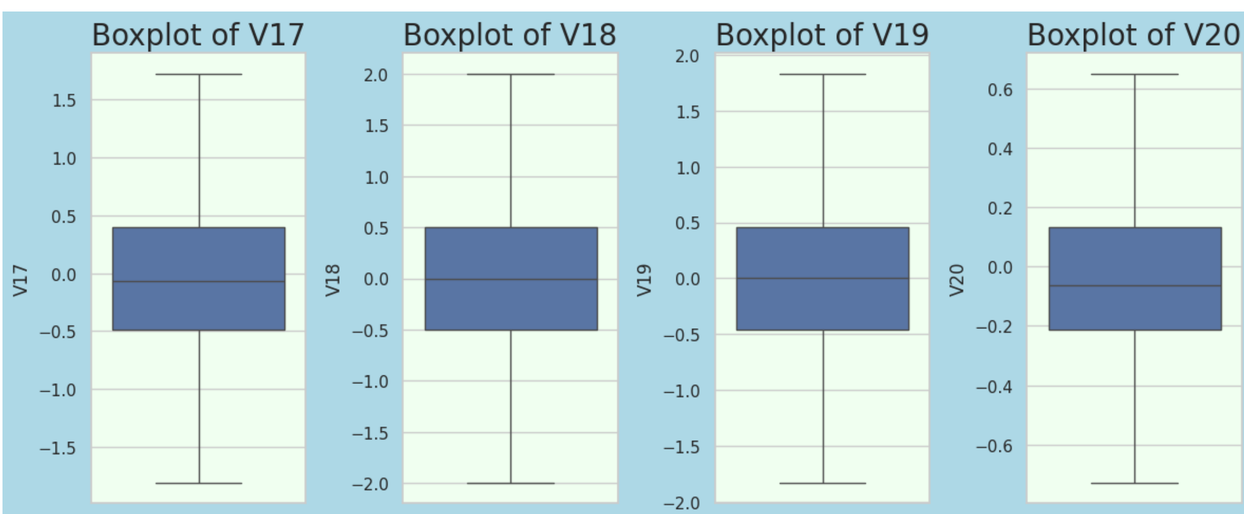
```
ax4.set_title('Boxplot of '+str(features_c[i*4+3]),fontsize=20)
ax4.set_facecolor('honeydew')

plt.tight_layout()
fig.set_facecolor('#ADD8E6')
```

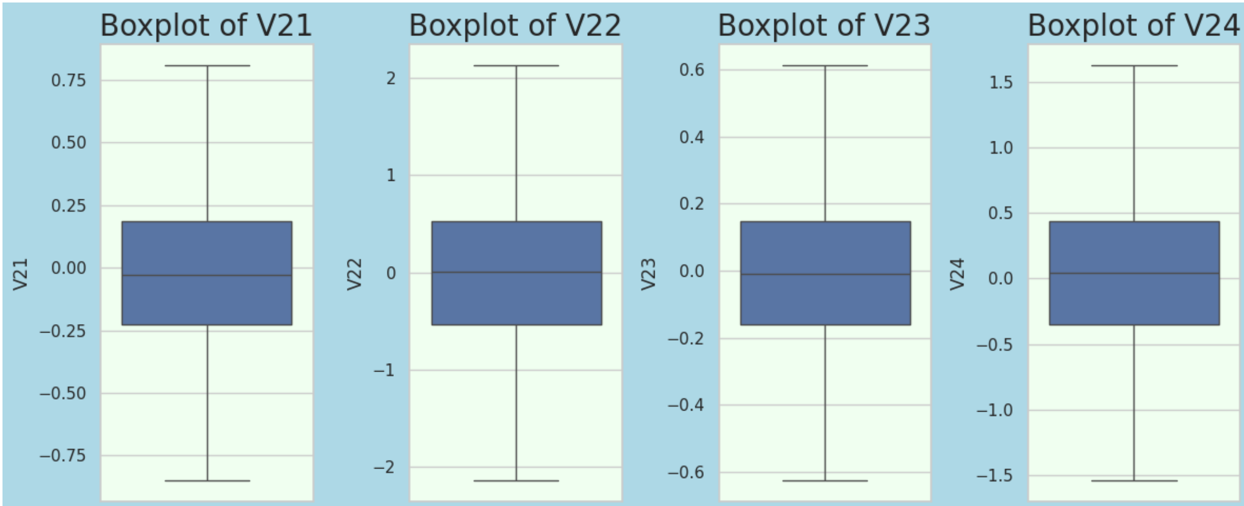


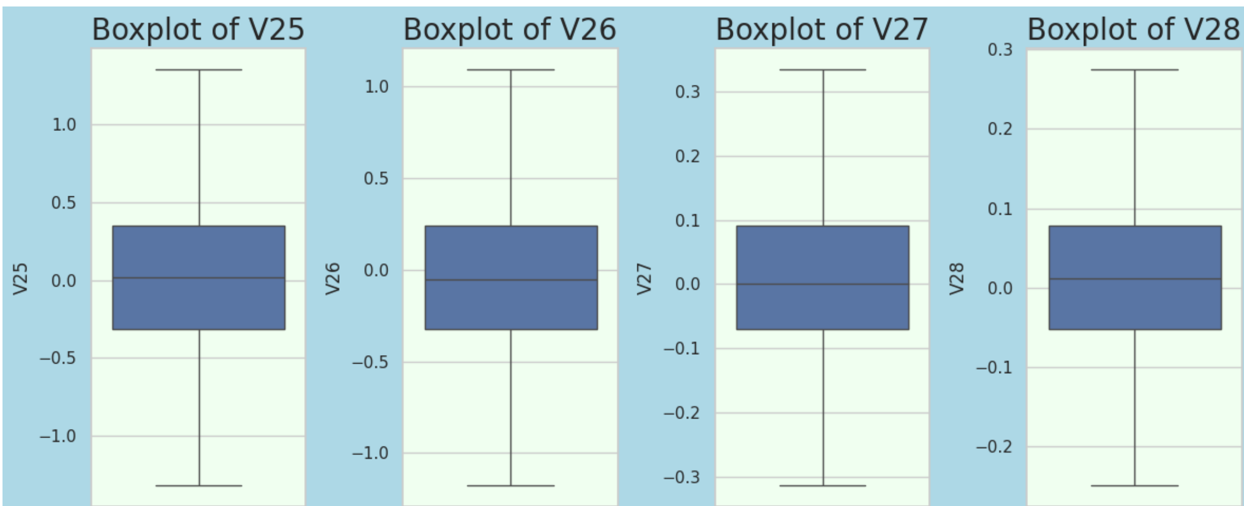



Boxplot of V9
V9
2
1
0
−1
−2
Boxplot of V10
V10
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Boxplot of V11
V11
3
2
1
0
−1
−2
−3
Boxplot of V12
V12
2
1
0
−1
−2


Boxplot of V13
V13
2
1
0
−1
−2
Boxplot of V14
V14
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Boxplot of V15
V15
2
1
0
−1
−2
Boxplot of V16
V16
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

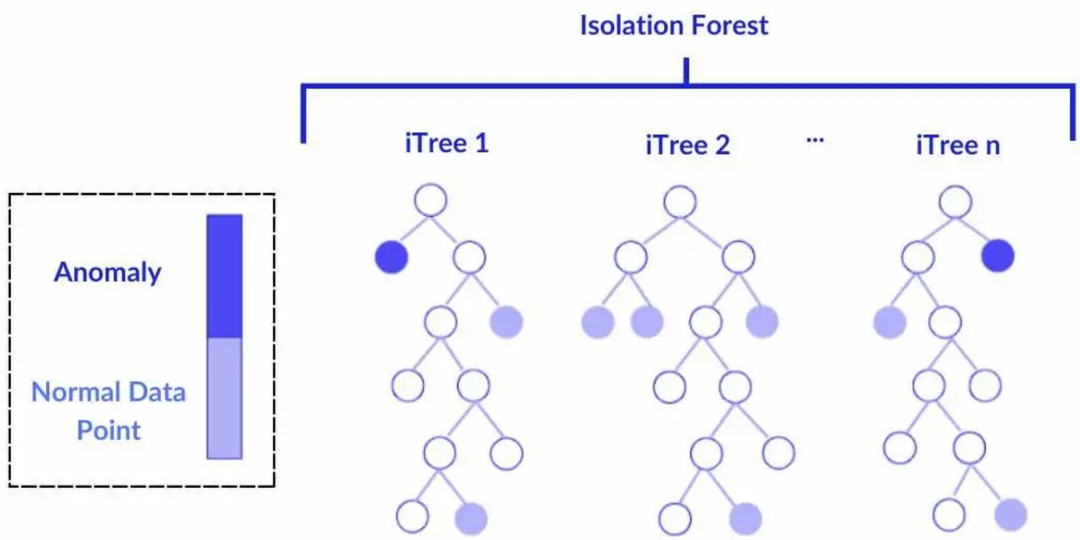

Boxplot of V17
V17
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Boxplot of V18
V18
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Boxplot of V19
V19
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Boxplot of V20
V20
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6





NOTE : after capping outliers, the data points are restricted to a defined range, which can potentially lead to a situation where many values are concentrated around the capped limits. This is where using anomaly detection techniques like Isolation Forest becomes especially important.

Isolation Forest & Dropping Outliers




```
data3 = data.copy()
data3 = data3.drop('Class',axis=1)

model =
IsolationForest(n_estimators=150,max_samples='auto',contamination=floa
t(0.1),max_features=1.0)
model.fit(data3)

IsolationForest(contamination=0.1, n_estimators=150)
```

Then, I am adding the score values and also an anomaly column to the dataframe. The negative score value of -1 indicates the presence of an anomaly. The value of 1 for the anomaly represents normal data.

You might set a threshold where scores less than 0 are considered anomalies, while scores equal to or greater than 0 are normal.

```
scores  = model.decision_function(data3)
scores

array([ 0.06627987,  0.07926464, -0.02267625, ...,  0.05559711,
        0.04616394,  0.07965383])
```

Calculating Scores: scores = model.decision_function(data3) computes the anomaly score for each data point in data3. The score reflects how isolated a data point is:

Points with lower scores are more likely to be anomalies.

scores that are more negative (e.g., -0.5, -1.0). These scores suggest that the point is more isolated, meaning it does not fit the typical pattern of the majority of the data.

```
anomaly = model.predict(data3)
anomaly
```

```
array([ 1,  1, -1, ...,  1,  1,  1])
```

generates binary predictions for the data points: The method returns -1 for points identified as anomalies and 1 for normal points.

```
#adding scores and anomalies
data3['score']   = scores
data3['anomaly'] = anomaly

data3.head()
```

{"type":"dataframe","variable_name":"data3"}

```
anomaly = data3.loc[data3['anomaly'] == -1]
print('The total number of outliers is {} out of 
{}.'.format(len(anomaly),len(data)))
```

```
The total number of outliers is 28373 out of 283726.
```

---

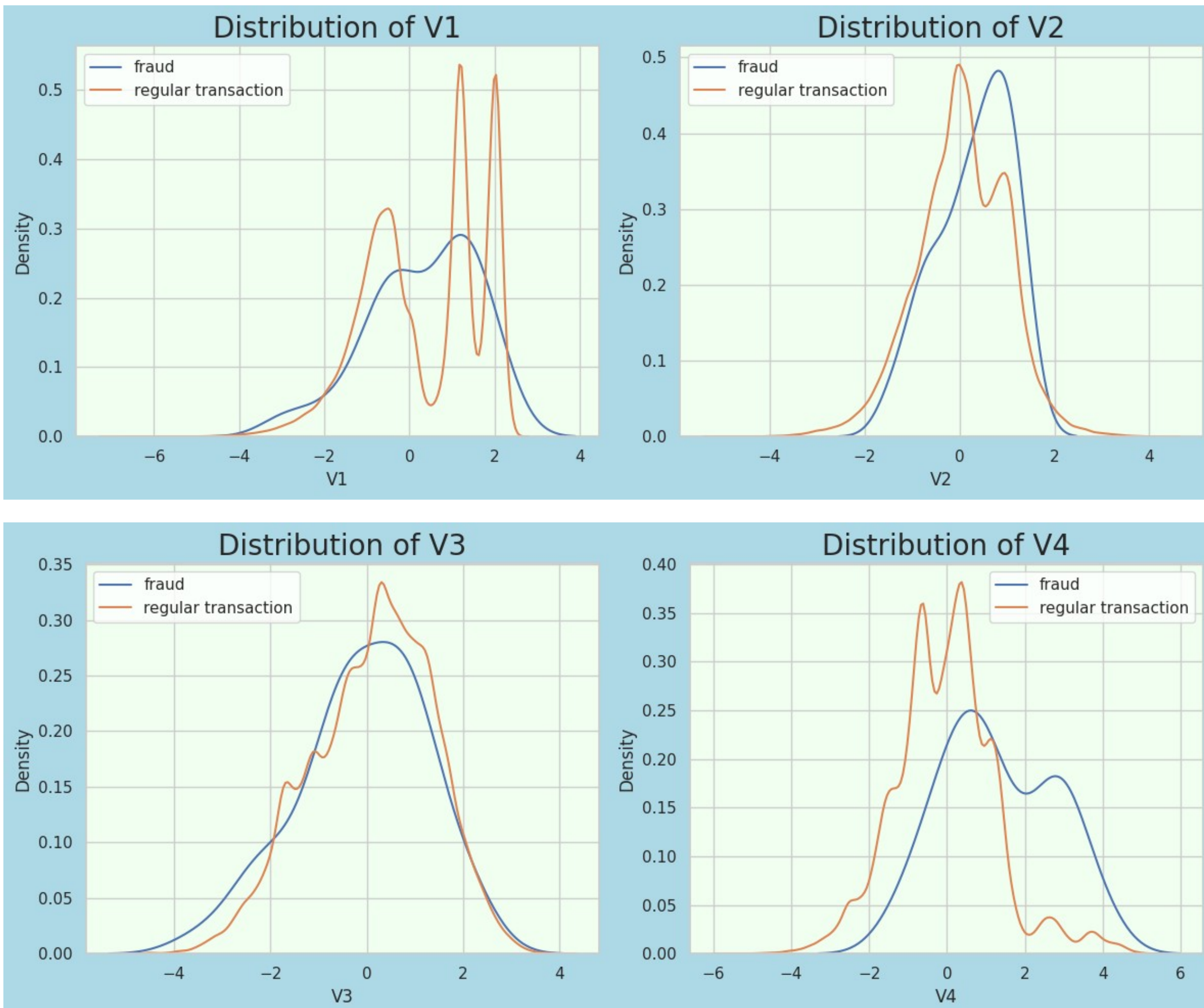
dropping the outliers.

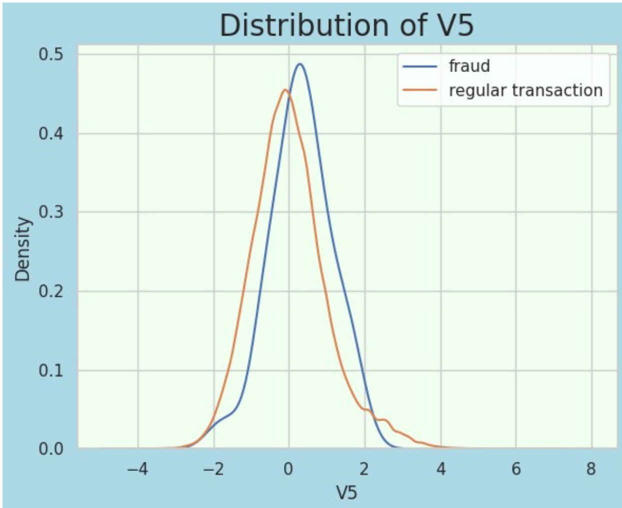
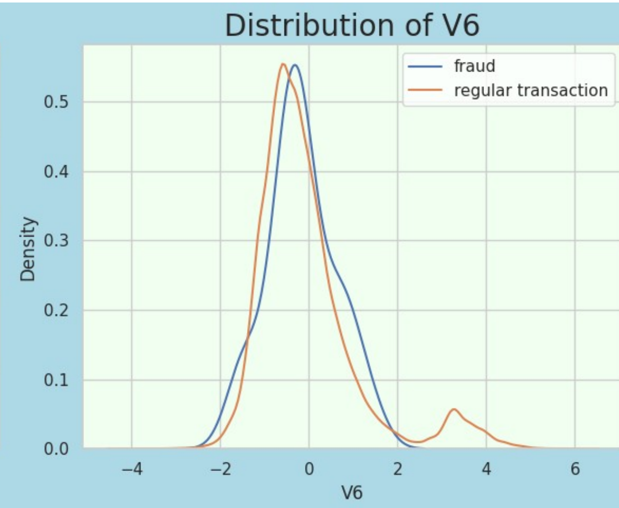
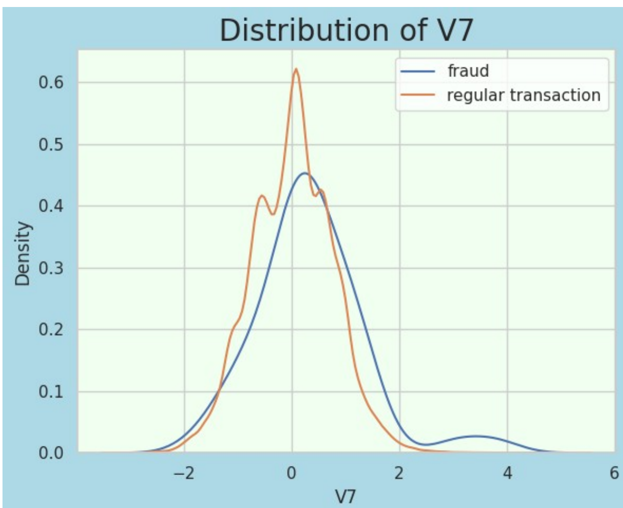
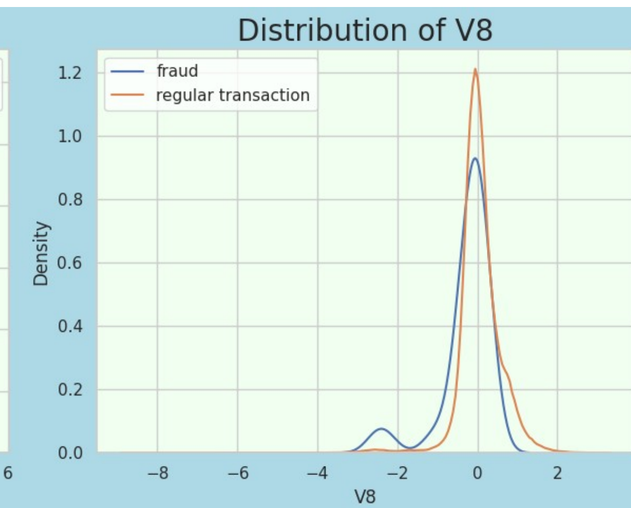
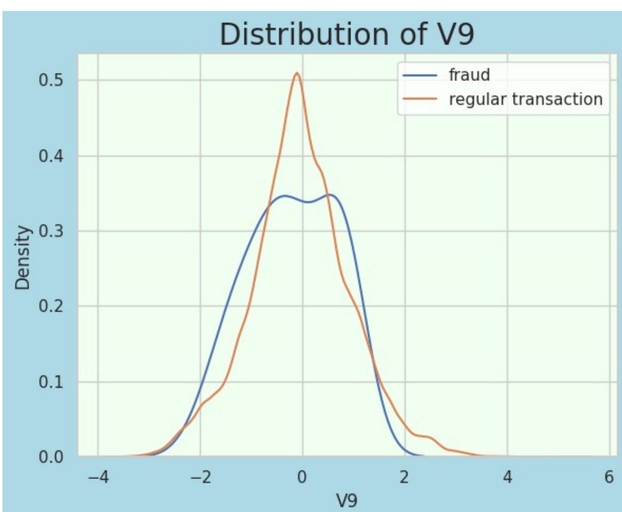
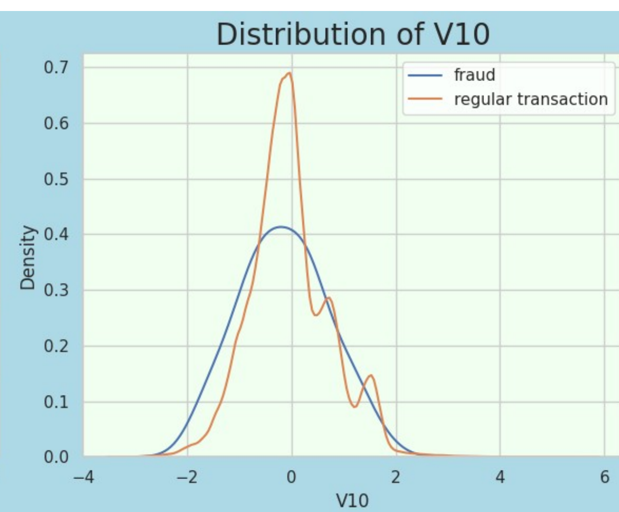
```
anomaly_index = list(anomaly.index)
forest_data = data.drop(anomaly_index,axis=0).reset_index(drop=True)
```

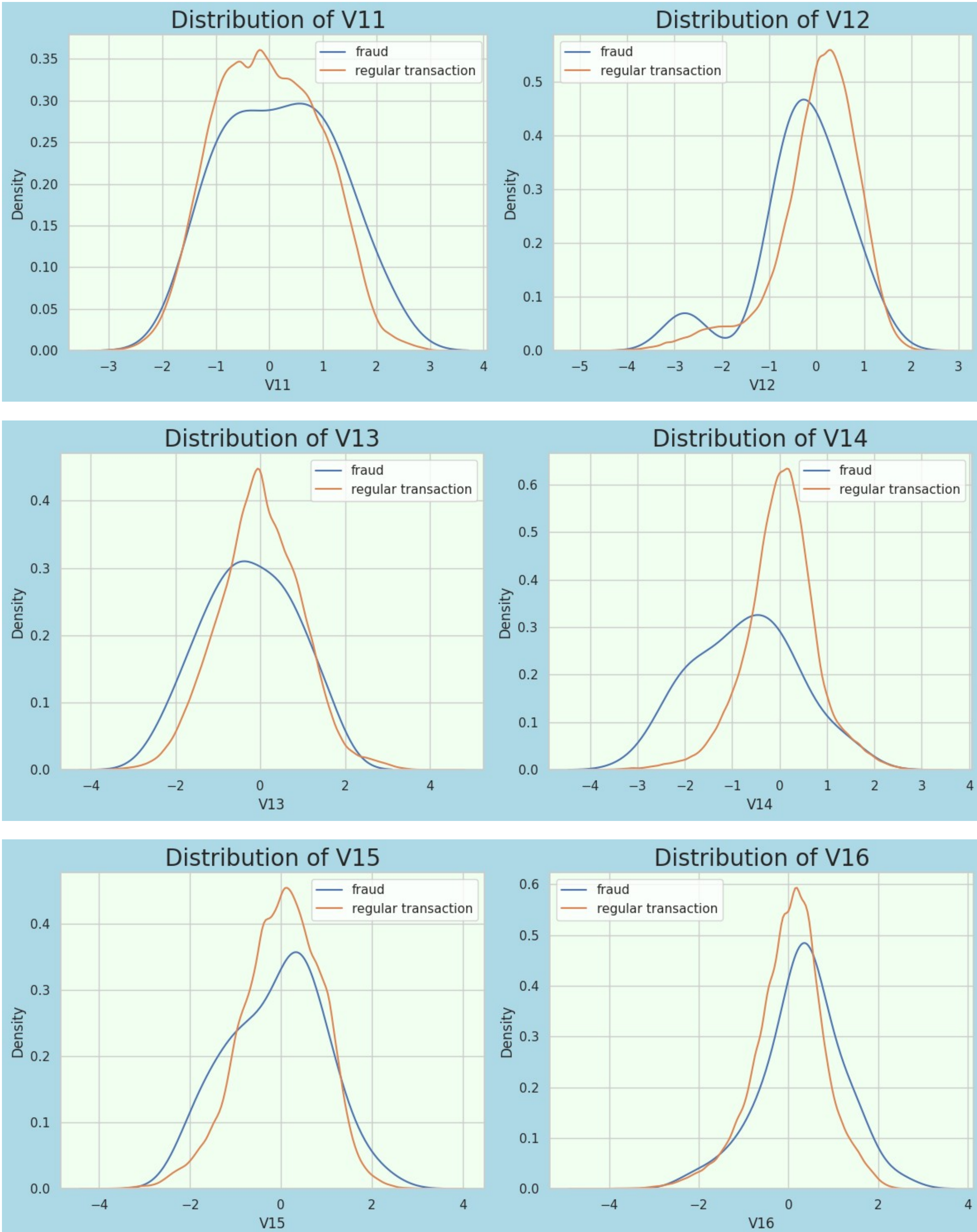
FINAL DISTRIBUTION

```
for i in range(14):
    fig,(ax1,ax2) = plt.subplots(ncols=2,figsize=(12,5))
    ax1 = sns.distplot(forest_data[forest_data.Class == 1]
[features[i*2]],ax=ax1,hist=False)
    ax1 = sns.distplot(forest_data[forest_data.Class == 0]
[features[i*2]],ax=ax1,hist=False)
    ax1.set_title('Distribution of '+str(features[i*2]),fontsize=20)
    ax1.set_facecolor('honeydew')
    ax1.legend(labels=['fraud','regular transaction'])
    ax2 = sns.distplot(forest_data[forest_data.Class == 1]
[features[i*2+1]],ax=ax2,hist=False)
    ax2 = sns.distplot(forest_data[forest_data.Class == 0]
```

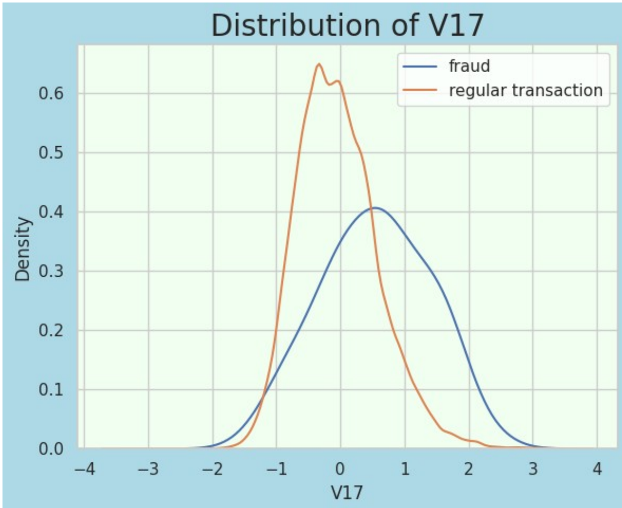
```
[features[i*2+1]],ax=ax2,hist=False)
    ax2.set_title('Distribution of '+str(features[i*2+1]),fontsize=20)
    ax2.set_facecolor('honeydew')
    ax2.legend(labels=['fraud','regular transaction'])

    plt.tight_layout()
    fig.set_facecolor('#ADD8E6')
```

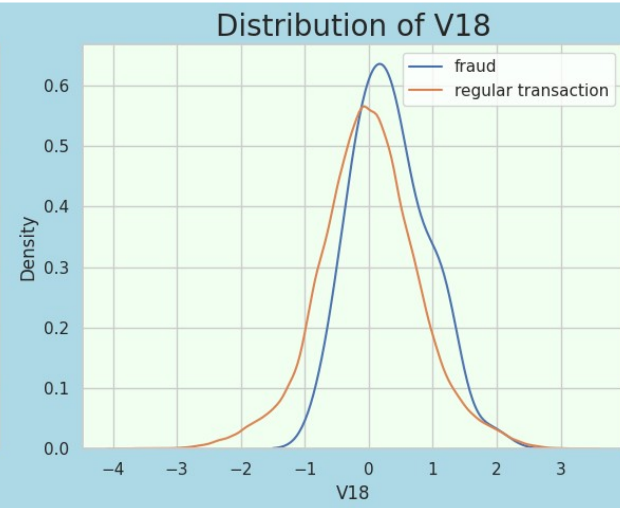


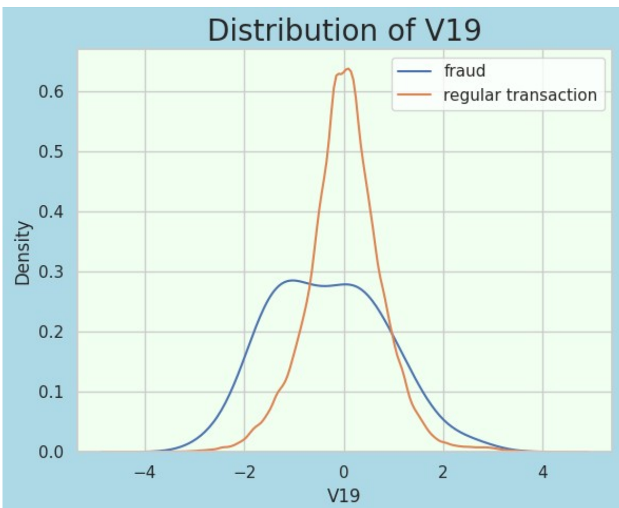

Distribution of V5
fraud
regular transaction
Density
V5

Distribution of V6
fraud
regular transaction
Density
V6

Distribution of V7
fraud
regular transaction
Density
V7

Distribution of V8
fraud
regular transaction
Density
V8

Distribution of V9
fraud
regular transaction
Density
V9

Distribution of V10
fraud
regular transaction
Density
V10

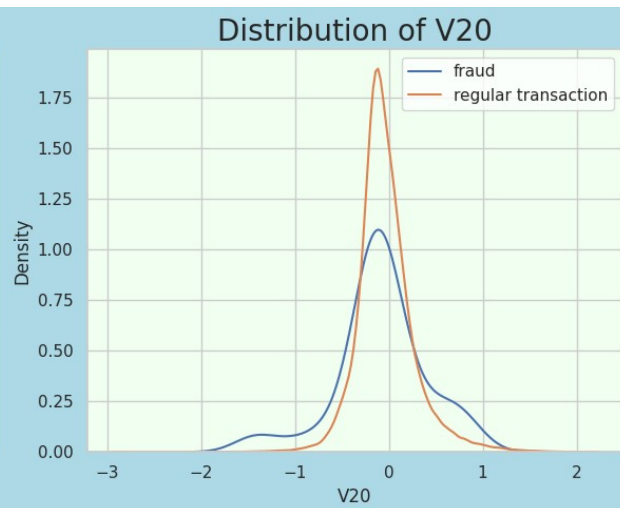


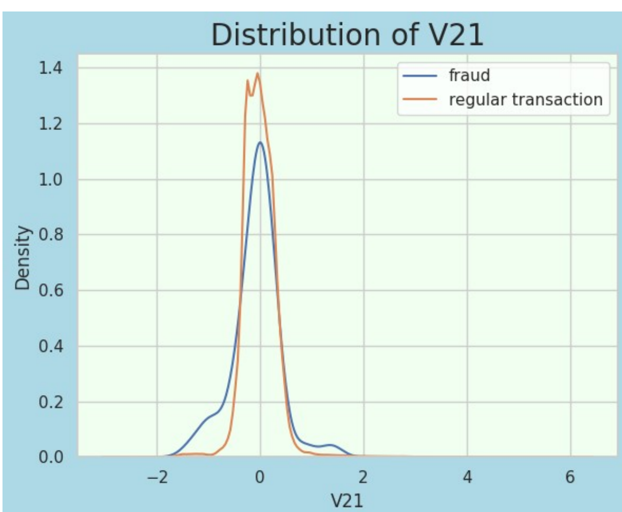
Distribution of V11
fraud
regular transaction
Density
V11
Distribution of V12
fraud
regular transaction
Density
V12
Distribution of V13
fraud
regular transaction
Density
V13
Distribution of V14
fraud
regular transaction
Density
V14
Distribution of V15
fraud
regular transaction
Density
V15
Distribution of V16
fraud
regular transaction
Density
V16

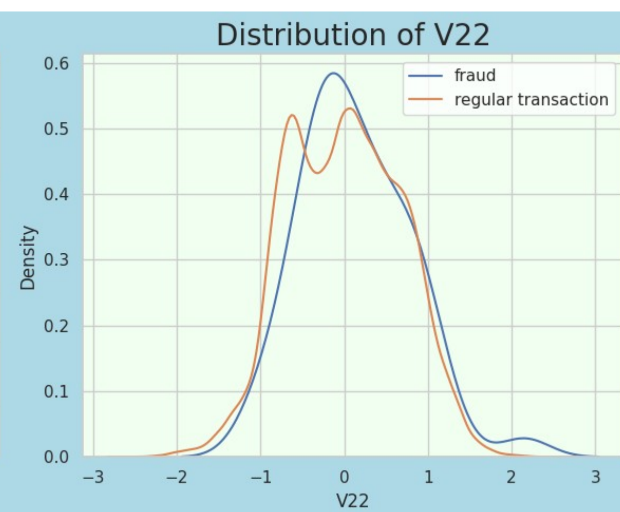

Distribution of V17
fraud
regular transaction
Density
V17

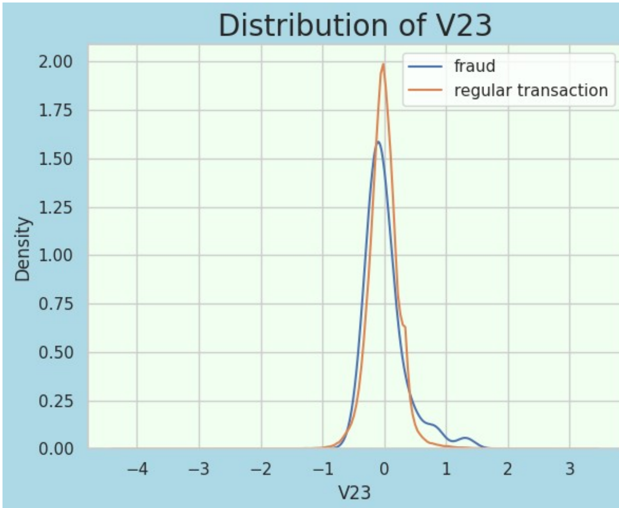

Distribution of V18
fraud
regular transaction
Density
V18

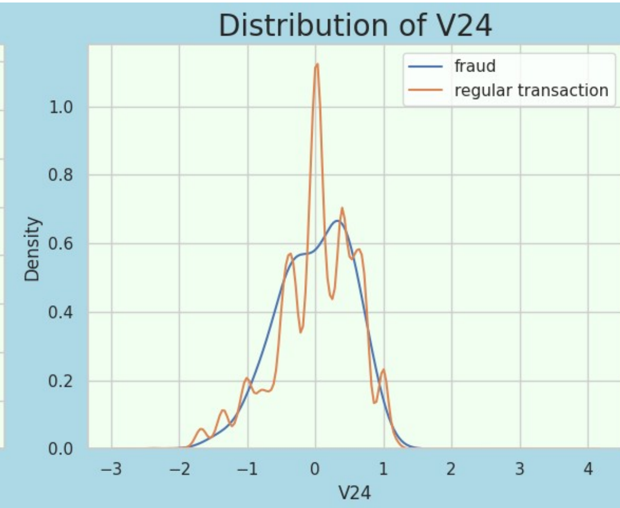

Distribution of V19
fraud
regular transaction
Density
V19

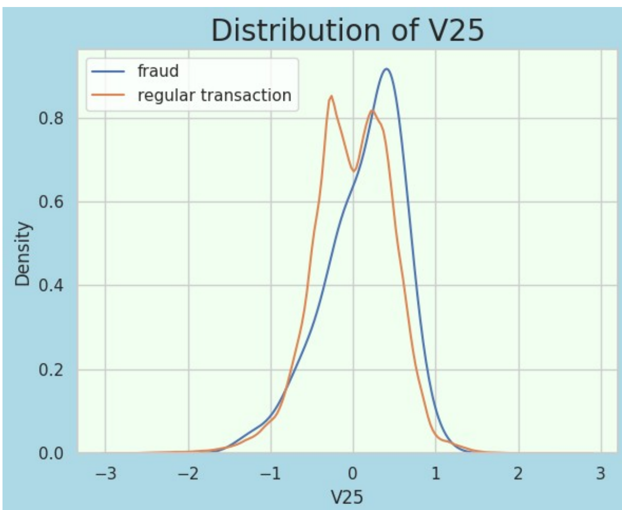

Distribution of V20
fraud
regular transaction
Density
V20

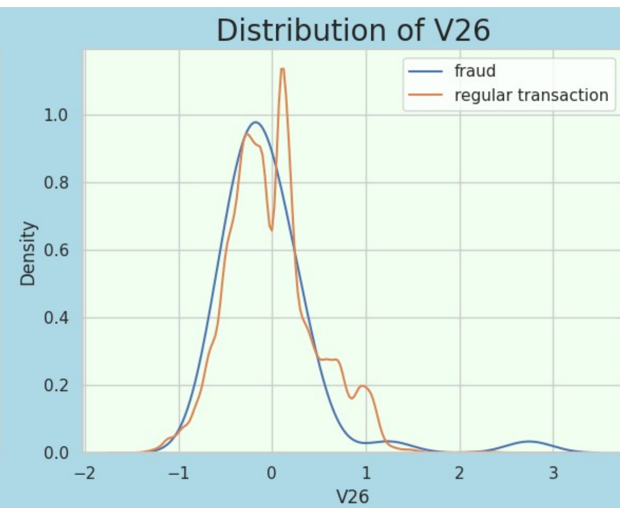

Distribution of V21
fraud
regular transaction
Density
V21

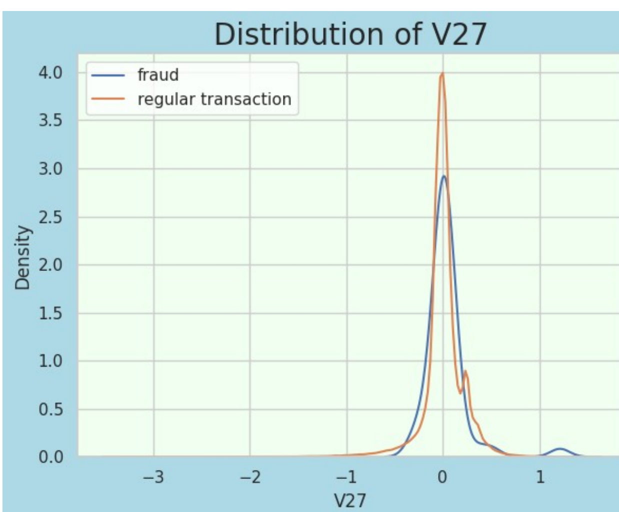

Distribution of V22
fraud
regular transaction
Density
V22

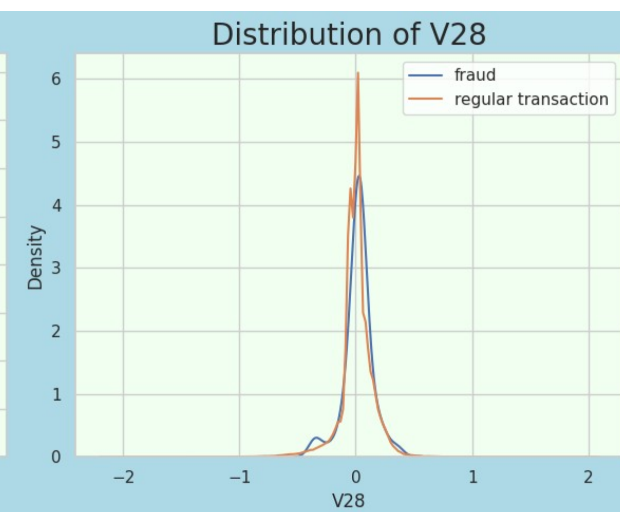

Distribution of V23
fraud
regular transaction
Density
V23


Distribution of V24
fraud
regular transaction
Density
V24


Distribution of V25
fraud
regular transaction
Density
V25

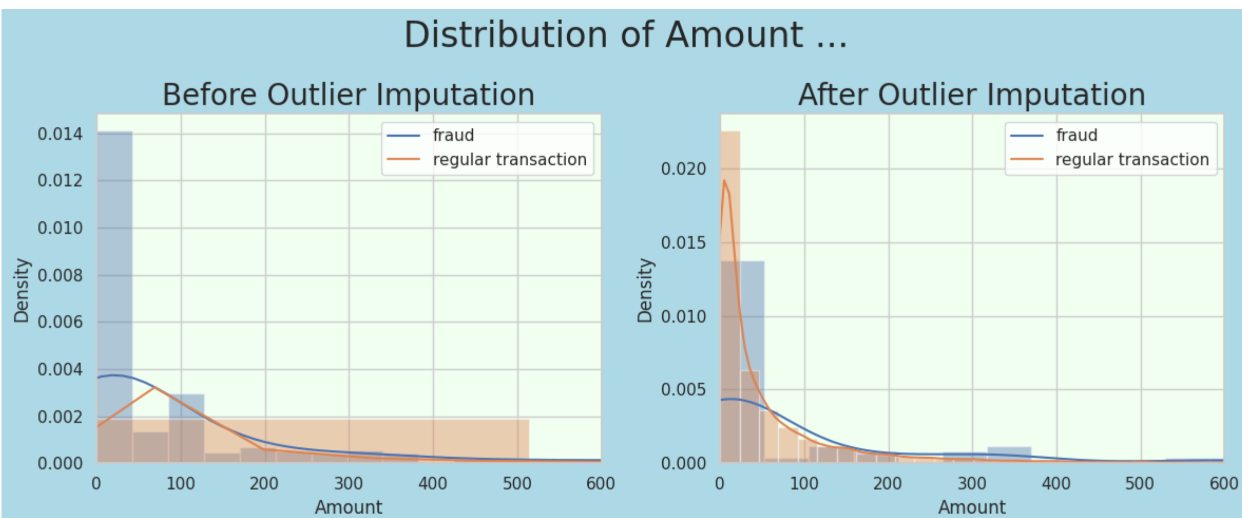

Distribution of V26
fraud
regular transaction
Density
V26


Distribution of V27
fraud
regular transaction
Density
V27


Distribution of V28
fraud
regular transaction
Density
V28

COMPARING

```
fig,(ax1,ax2) = plt.subplots(ncols=2,figsize=(12,5))


#BEFORE
ax1 = sns.distplot(data['Amount'][data.Class == 1],ax=ax1)
ax1 = sns.distplot(data['Amount'][data.Class == 0],ax=ax1)
ax1.set_xlim(0,600)
ax1.set_title('Before Outlier Imputation',fontsize=20)
ax1.legend(labels=['fraud','regular transaction'])
ax1.set_facecolor('honeydew')

#AFTER
ax2 = sns.distplot(forest_data['Amount'][forest_data.Class ==
1],ax=ax2)
ax2 = sns.distplot(forest_data['Amount'][forest_data.Class ==
0],ax=ax2)
ax2.set_xlim(0,600)
ax2.set_title('After Outlier Imputation',fontsize=20)
ax2.legend(labels=['fraud','regular transaction'])
ax2.set_facecolor('honeydew')

#PLOT

fig.suptitle("Distribution of Amount ...",fontsize=24)

plt.tight_layout()
fig.set_facecolor('#ADD8E6')
```



Before Outlier Imputation:

In the left plot, some transactions have very high amounts, which makes the data look stretched out to the right. Regular transactions usually have smaller amounts, while fraud transactions sometimes go higher.

After Outlier Imputation:

In the right plot, those high-amount transactions have been reduced or adjusted, so the data is now focused more on smaller amounts. This makes the graph more compact, with fewer large transactions.

In short: MACHINE FOCOUS ON HIGH VALUE SO ,

We reduced the impact of big, unusual transaction amounts, so now the data focuses more on typical, smaller amounts. This can help make patterns in the data clearer.

FEATURE IMPORTANCE

identify which features are most influential in making predictions.

```
X = forest_data.drop('Class',axis=1)
y = forest_data['Class']
```

I can look at the relative importance of the features by means of a random forest classifier.

```
#USING RANDOM FOREST FIND FEATURE IMPORTANCE
# Random Forest Model
random_forest = RandomForestClassifier(random_state=1,max_depth=4)
random_forest.fit(X,y)
```

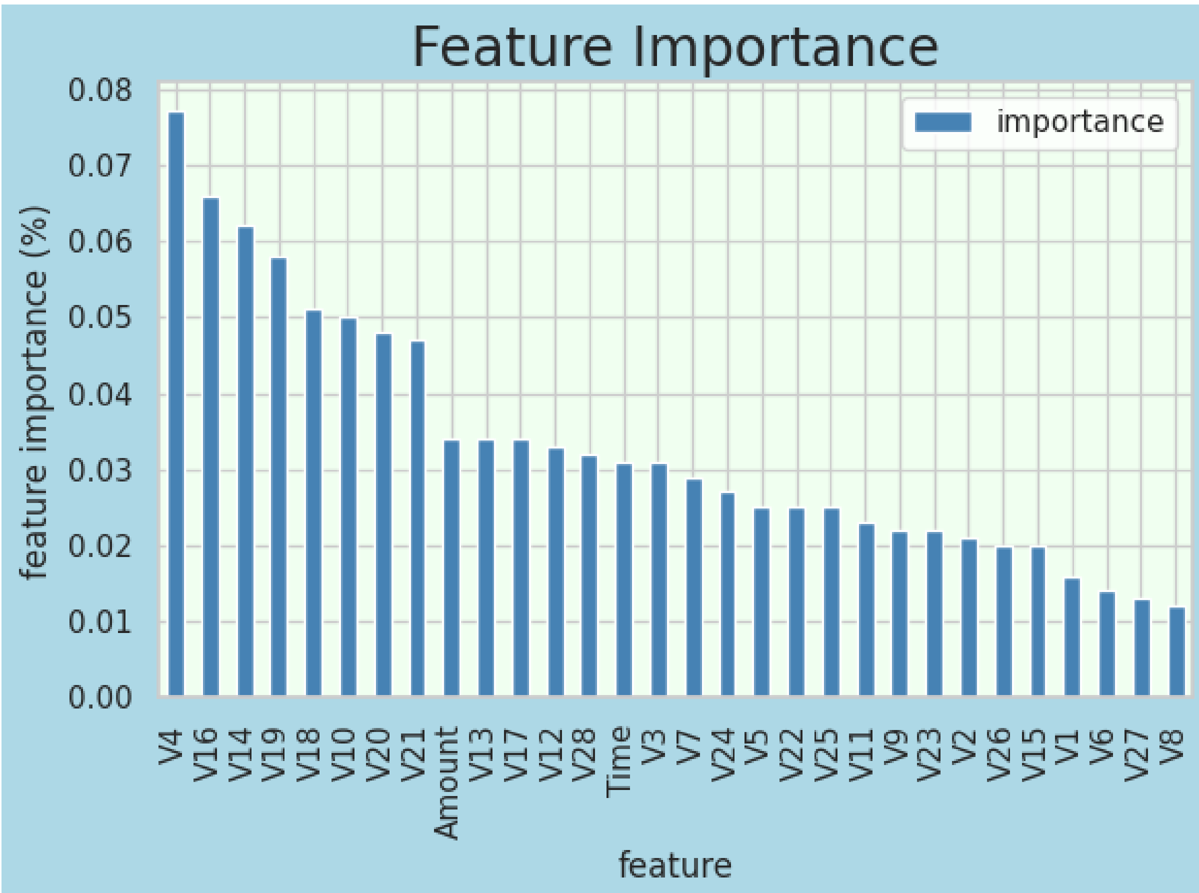
```
RandomForestClassifier(max_depth=4, random_state=1)
```

```
FI=random_forest.feature_importances_
importances =
pd.DataFrame({'feature':X.columns,'importance':np.round(FI,3)})
importances =
importances.sort_values('importance',ascending=False).set_index('featu
re')
importances.head(30)
```


```
{"summary":"{\n  \"name\": \"importances\",\n  \"rows\": 30,\n
\"fields\": [\n    {\n      \"column\": \"feature\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 30,\n        \"samples\": [\n          \"V6\",\
n          \"V7\",\n          \"V2\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"importance\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.016703499530090215,\n        \"min\": 0.012,\n        \"max\":
0.077,\n        \"num_unique_values\": 23,\n        \"samples\": [\n
0.023,\n          0.033,\n          0.077\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"importances"}
```


```
importances.plot.bar(color='steelblue')
```

```
plt.ylabel('feature importance (%)',fontsize=12)
plt.title('Feature Importance',fontsize=20)

plt.tight_layout()
plt.gcf().patch.set_facecolor('#ADD8E6')
plt.gca().set_facecolor('honeydew')
plt.show()
```



TO AVOID OVERFITTING :Apparently, there are three dominant features: V14, V17 and V4. All the others are much less important.SO WE DROP SOME COLUMNS

#experiment :

without drop any column random_forest accuracy --0.9997 DRP -'V1','V2','V3','V5','V6','V7','V8','V8','V9','V11','V13','V14','V15','V16','V18','V19','V20','V21', 'V22','V23','V24','V25','V26','V27','V28' random_forest accuracy --->0.9997

so no DIFFERENCE

```
# Drop all of the features that have very similar distributions
between frauds and non-frauds.
X = X.drop(['V2','V5','V6','V7','V8','V11','V15','V16','V18',
            'V22','V23','V25','V26','V27','V28'],axis=1)
```

```python
# Train-test split
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.3,random_state=0)

random_forest = RandomForestClassifier(class_weight='balanced')

random_forest.fit(X_train,y_train)
```

```
RandomForestClassifier(class_weight='balanced')
```

```python
def get_test_scores(model_name:str,preds,y_test_data):

    accuracy  = accuracy_score(y_test_data,preds)
    precision = precision_score(y_test_data,preds,average='macro')
    recall    = recall_score(y_test_data,preds,average='macro')
    f1        = f1_score(y_test_data,preds,average='macro')

    table = pd.DataFrame({'model': [model_name],'precision':
[precision],'recall': [recall],
                          'F1': [f1],'accuracy': [accuracy]})

    return table

# Use the model to predict on test data
rf_test_preds = random_forest.predict(X_test)

rf_test_results = get_test_scores('RF (test)',rf_test_preds,y_test)
rf_test_results
```


{"summary":"{\n \"name\": \"rf_test_results\",\n \"rows\": 1,\n
\"fields\": [\n   {\n   \"column\": \"model\",\n
\"properties\": {\n   \"dtype\": \"string\",\n
\"num_unique_values\": 1,\n   \"samples\": [\n   \"RF
(test)\"\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"precision\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": null,\n   \"min\":
0.49989556953763414,\n   \"max\": 0.49989556953763414,\n
\"num_unique_values\": 1,\n   \"samples\": [\n
0.49989556953763414\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": null,\n   \"min\": 0.5,\n   \"max\": 0.5,\n
\"num_unique_values\": 1,\n   \"samples\": [\n   0.5\n
],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   },\n   {\n   \"column\": \"F1\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": null,\n   \"min\":
0.4999477793153868,\n   \"max\": 0.4999477793153868,\n
\"num_unique_values\": 1,\n   \"samples\": [\n
0.4999477793153868\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":

```
\"accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.9997911390752683,\n        \"max\": 0.9997911390752683,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.9997911390752683\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"rf_test_results"}
```

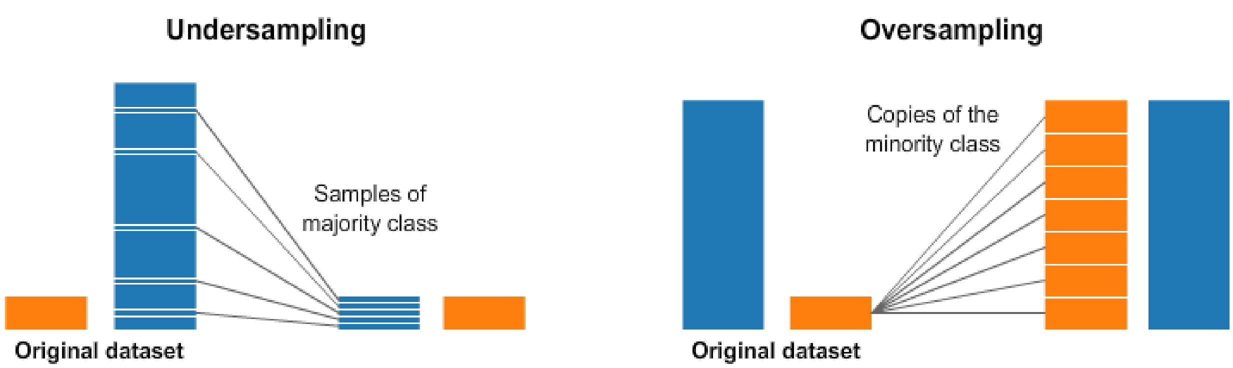

#Accuracy gives a general idea of how well the model performs overall.

#Precision is important when the cost of false positives is high (e.g., in spam detection).

#Recall is crucial when the cost of false negatives is high (e.g., in disease detection).

#F1 Score is a balanced measure that is useful when you need a balance between precision and recall, especially when the class distribution is imbalanced.

Resampling and Cross Validation



Undersampling

```
from imblearn.under_sampling import RandomUnderSampler

# Create a RandomUnderSampler object
rus = RandomUnderSampler(random_state=42,sampling_strategy='majority')

# Balancing the data
X_resampled,y_resampled = rus.fit_resample(X_train,y_train)# sample we
used
```

Random Forest Classifier

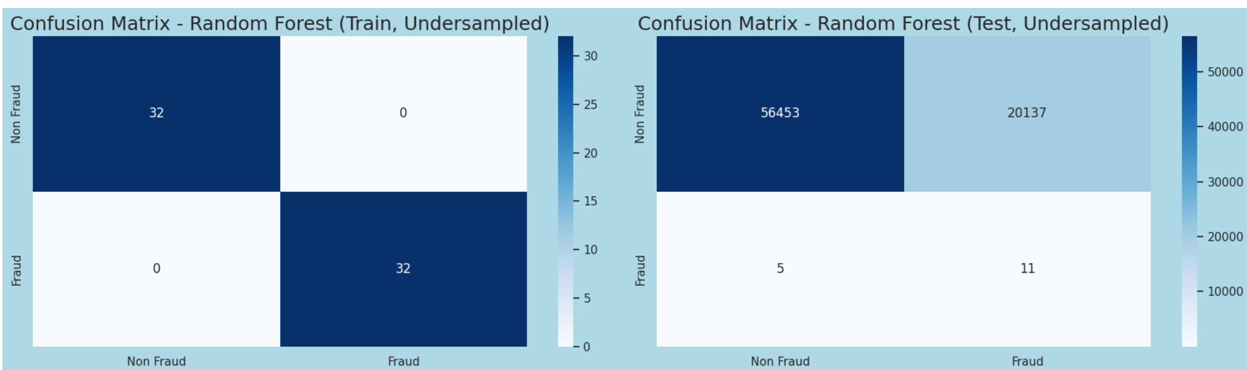
```
random_forest.fit(X_resampled,y_resampled)

# Use the model to predict on train data
rf_train_resampled_preds = random_forest.predict(X_resampled)
```

```
# Use the model to predict on test data
rf_test_preds = random_forest.predict(X_test)

rf_test_results = get_test_scores('RF (test)',rf_test_preds,y_test)
rf_test_results
```


```
{"summary":"{\n  \"name\": \"rf_test_results\",\n  \"rows\": 1,\n
\"fields\": [\n    {\n      \"column\": \"model\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          \"RF
(test)\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"precision\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.5002286992441416,\n        \"max\": 0.5002286992441416,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.5002286992441416\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"recall\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": null,\n        \"min\": 0.7122902794098447,\n        \"max\":
0.7122902794098447,\n        \"num_unique_values\": 1,\n
\"samples\": [\n          0.7122902794098447\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"F1\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": null,\n        \"min\":
0.42485104048074507,\n        \"max\": 0.42485104048074507,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.42485104048074507\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.7370702033783254,\n        \"max\": 0.7370702033783254,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.7370702033783254\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"rf_test_results"}
```


```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(17, 5))
# Generate array of values for confusion matrix
cm1 =
confusion_matrix(y_resampled,rf_train_resampled_preds,labels=random_fo
rest.classes_)
# Plot confusion matrix for the training set
sns.heatmap(cm1, annot=True, ax=ax1, fmt='d', cmap='Blues')
ax1.xaxis.set_ticklabels(['Non Fraud', 'Fraud'])
ax1.yaxis.set_ticklabels(['Non Fraud', 'Fraud'])
ax1.set_title('Confusion Matrix - Random Forest (Train,
Undersampled)', fontsize=18)
ax1.set_facecolor('honeydew')
```

```
# Generate array of values for confusion matrix
cm2 =
confusion_matrix(y_test,rf_test_preds,labels=random_forest.classes_)
# Plot confusion matrix for the test set
sns.heatmap(cm2, annot=True, ax=ax2, fmt='d', cmap='Blues')
ax2.xaxis.set_ticklabels(['Non Fraud', 'Fraud'])
ax2.yaxis.set_ticklabels(['Non Fraud', 'Fraud'])
ax2.set_title('Confusion Matrix - Random Forest (Test, Undersampled)',
fontsize=18)
ax2.set_facecolor('honeydew')

#plot
# Set figure background color
plt.gcf().patch.set_facecolor('#ADD8E6')

# Adjust layout
plt.tight_layout()
plt.show()
```

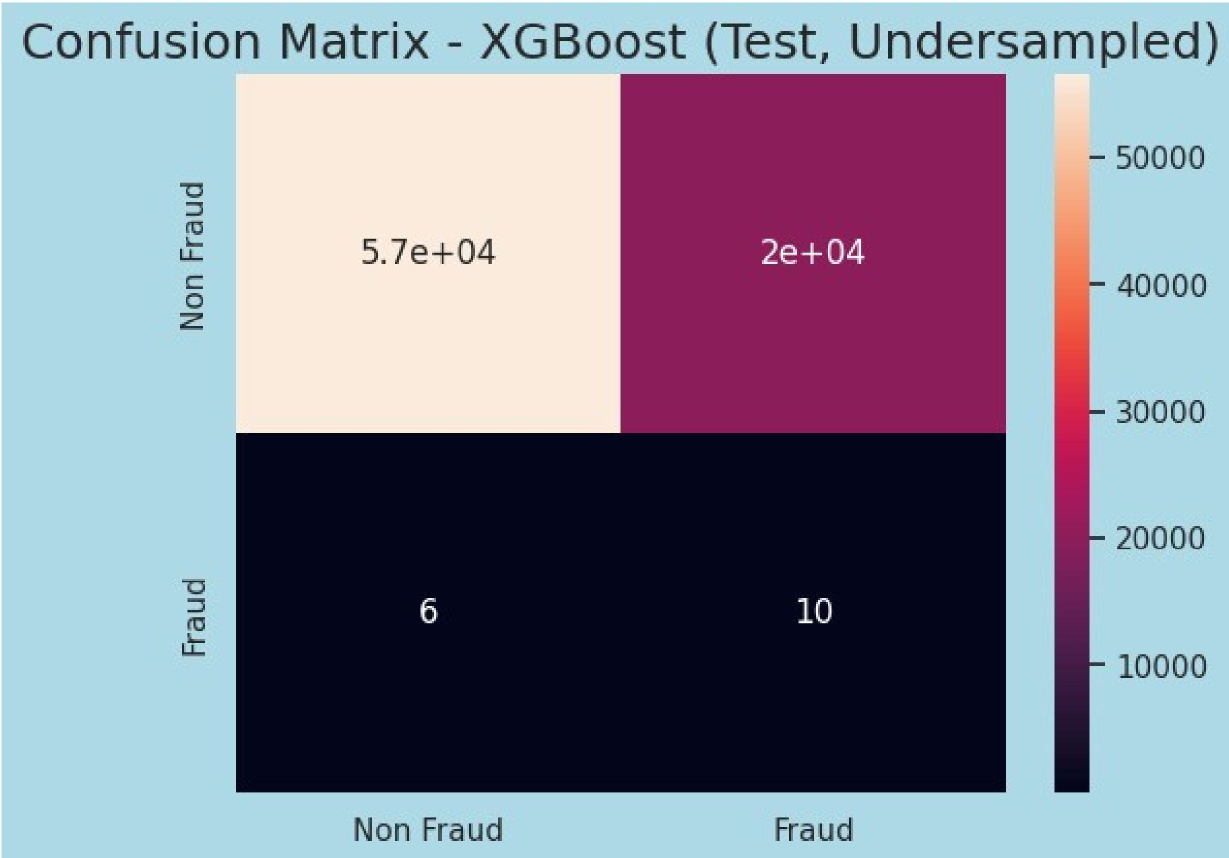


XGBoost Classifier

```
# Instantiate the XGBoost classifier
xgb = XGBClassifier(objective='binary:logistic',random_state=42)

xgb.fit(X_resampled,y_resampled)

# Use the model to predict on train data
xgb_train_resampled_preds = xgb.predict(X_resampled)

# Use the model to predict on test data
xgb_test_preds = xgb.predict(X_test)

xgb_test_results = get_test_scores('XGB (test)',xgb_test_preds,y_test)
xgb_test_results
```


```
{"summary":"{\n  \"name\": \"xgb_test_results\",\n  \"rows\": 1,\n
\"fields\": [\n    {\n      \"column\": \"model\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
```

```
\"num_unique_values\": 1,\n        \"samples\": [\n          \"XGB
(test)\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"precision\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.5001965325595095,\n        \"max\": 0.5001965325595095,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.5001965325595095\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"recall\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": null,\n        \"min\": 0.6817714453584018,\n        \"max\":
0.6817714453584018,\n        \"num_unique_values\": 1,\n
\"samples\": [\n          0.6817714453584018\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"F1\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": null,\n        \"min\":
0.42528526484153406,\n        \"max\": 0.42528526484153406,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.42528526484153406\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.7385191760436519,\n        \"max\": 0.7385191760436519,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.7385191760436519\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"xgb_test_results"}


# Generate array of values for confusion matrix
cm = confusion_matrix(y_test,xgb_test_preds,labels=xgb.classes_)

ax = sns.heatmap(cm,annot=True)
ax.xaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.yaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.set_title('Confusion Matrix - XGBoost (Test,
Undersampled)',fontsize=18)

plt.gcf().patch.set_facecolor('#ADD8E6')
```



Oversampling

```
from imblearn.over_sampling import SMOTE

smote = SMOTE()

# Balancing the data
X_oversampled,y_oversampled = smote.fit_resample(X_train,y_train)
```

Random Forest ***Classifier***

```
random_forest.fit(X_oversampled,y_oversampled)

# Use the model to predict on train data
rf_train_oversampled_preds = random_forest.predict(X_oversampled)

rf_train_oversampled_results = get_test_scores('RF (train,
oversampled)',rf_train_oversampled_preds,y_oversampled)
rf_train_oversampled_results
```

{"summary":"{\n \"name\": \"rf_train_oversampled_results\",\n
\"rows\": 1,\n \"fields\": [\n   {\n   \"column\": \"model\",\n
\"properties\": {\n   \"dtype\": \"string\",\n
\"num_unique_values\": 1,\n   \"samples\": [\n   \"RF

(train, oversampled)\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 1.0,\n   \"max\": 1.0,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   1.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 1.0,\n   \"max\": 1.0,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   1.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"F1\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 1.0,\n   \"max\": 1.0,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   1.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"accuracy\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 1.0,\n   \"max\": 1.0,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   1.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe","variable_name":"rf_train_oversampled_results"}

```
# Use the model to predict on test data
rf_test_preds = random_forest.predict(X_test)

rf_test_results = get_test_scores('RF (test)',rf_test_preds,y_test)
rf_test_results
```

{"summary":"{\n \"name\": \"rf_test_results\",\n \"rows\": 1,\n \"fields\": [\n   {\n   \"column\": \"model\",\n   \"properties\": {\n   \"dtype\": \"string\",\n   \"num_unique_values\": 1,\n   \"samples\": [\n   \"RF (test)\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 0.4998955654478284,\n   \"max\": 0.4998955654478284,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   0.4998955654478284\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 0.4999804151978065,\n   \"max\": 0.4999804151978065,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   0.4999804151978065\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"F1\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 0.49993798672263096,\n   \"max\": 0.49993798672263096,\n   \"num_unique_values\": 1,\n   \"samples\": [\n

```
0.49993798672263096\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.9997519776518811,\n        \"max\": 0.9997519776518811,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.9997519776518811\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"rf_test_results"}
```

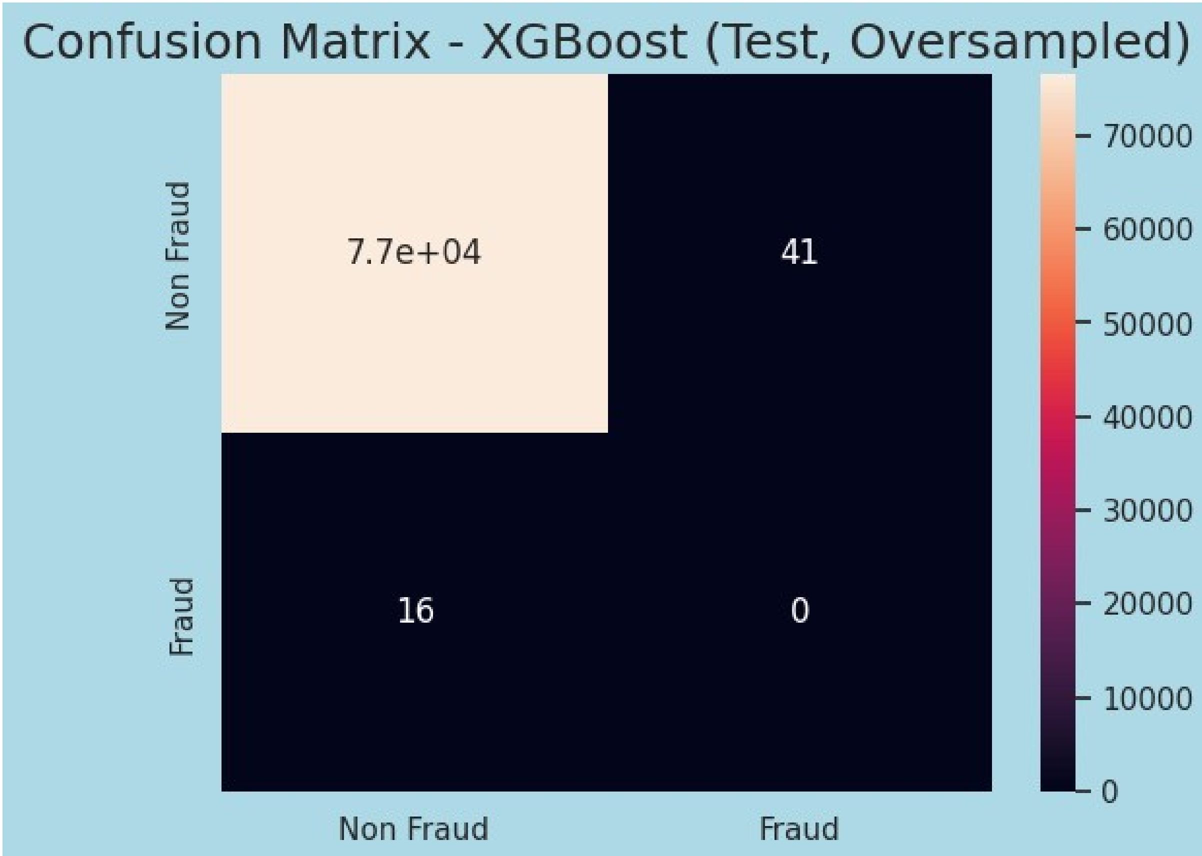

XGBoost Classifier

```
# 1. Instantiate the XGBoost classifier
xgb = XGBClassifier(objective='binary:logistic',random_state=42)

xgb.fit(X_oversampled,y_oversampled)

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=42, ...)

# Use the model to predict on test data
xgb_test_preds = xgb.predict(X_test)

xgb_test_results = get_test_scores('XGB (test)',xgb_test_preds,y_test)
xgb_test_results
```



```
{"summary":"{\n  \"name\": \"xgb_test_results\",\n  \"rows\": 1,\n
\"fields\": [\n    {\n      \"column\": \"model\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 1,\n        \"samples\": [\n          \"XGB
(test)\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"precision\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.4998955136158819,\n        \"max\": 0.4998955136158819,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.4998955136158819\n        ],\n        \"semantic_type\": \"\",\n
```

```
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"recall\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": null,\n        \"min\": 0.49973234103668884,\n
\"max\": 0.49973234103668884,\n        \"num_unique_values\": 1,\n
\"samples\": [\n          0.49973234103668884\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"F1\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": null,\n        \"min\":
0.499813914008684,\n        \"max\": 0.499813914008684,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.499813914008684\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"accuracy\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": null,\n        \"min\":
0.9992559329556432,\n        \"max\": 0.9992559329556432,\n
\"num_unique_values\": 1,\n        \"samples\": [\n
0.9992559329556432\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"xgb_test_results"}

# Generate array of values for confusion matrix
cm = confusion_matrix(y_test,xgb_test_preds,labels=xgb.classes_)

ax = sns.heatmap(cm,annot=True)
ax.xaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.yaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.set_title('Confusion Matrix - XGBoost (Test,
Oversampled)',fontsize=18)

plt.gcf().patch.set_facecolor('#ADD8E6')
```



It seems that undersampling improves the classification of the minority class (frauds), while oversampling tends to do better on the majority class (non frauds, i.e. regular transactions).

In light of this, I will keep the undersampling strategy and try to improve its results by also implementing other strategies.

since undersampling random forest give best recall score so we cross validate to this and check score

```
# K-fold stratified cross validation
kfold = StratifiedKFold(n_splits=10)
kfold
```

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

```
#GRID SEARCH CV TAKE TOO MUCH TIM SO I COMMENT OUT IF U HAVE 6 CORE
PROCESSOR THEN IT WILL BE DONE IN 20 MIN. ACCURACY IS NEAR TO -->0.77

# RFC Parameters tuning
# RFC = RandomForestClassifier(random_state=42)

# rf_param_grid = {
#     'max_depth': [2,3,4,5,None],
#     'max_features': [1.0],
#     'max_samples': [1.0],
```

```
#     'min_samples_leaf': [2,3,4],
#     'min_samples_split': [2,2,4],
#     'n_estimators': [200,300,400]
#     }

# gsRFC =
GridSearchCV(RFC,param_grid=rf_param_grid,cv=kfold,scoring="f1")

# gsRFC.fit(X_resampled,y_resampled)

#FOR TIME TAKING I USE RANDOM SEARCH CV
from sklearn.model_selection import RandomizedSearchCV
RFC = RandomForestClassifier(random_state=42)
# Define the Stratified K-Folds cross-validator
kfold = StratifiedKFold(n_splits=10)

# Define the parameter grid for RandomizedSearchCV
rf_param_dist = {
    'max_depth': [2, 3, 4, 5, None],
    'max_features': [1.0],
    'max_samples': [1.0],
    'min_samples_leaf': [2, 3, 4],
    'min_samples_split': [2, 4],
    'n_estimators': [200, 300, 400]
}

# Initialize RandomizedSearchCV
rsRFC = RandomizedSearchCV(RFC, param_distributions=rf_param_dist,
n_iter=100, cv=kfold, scoring="f1", random_state=42)

# Fit the model
rsRFC.fit(X_resampled, y_resampled)

# Optionally, you can retrieve the best parameters
best_params = rsRFC.best_params_
print("Best parameters found: ", best_params)


# Use the model to predict on test data
rf_test_preds = rsRFC.predict(X_test)

rf_test_results = get_test_scores('RF (test)',rf_test_preds,y_test)
rf_test_results

# Generate array of values for confusion matrix
cm = confusion_matrix(y_test,rf_test_preds,labels=rsRFC.classes_)

ax = sns.heatmap(cm,annot=True)
ax.xaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.yaxis.set_ticklabels(['Non Fraud','Fraud'])
ax.set_title('Confusion Matrix - Random Forest (Test, Undersampled &
```

```
CV)',fontsize=18)

plt.gcf().patch.set_facecolor('#ADD8E6')
```

Till now Best Recall Score is --> under sampling , random forest -->recall score of 0.788003 for your credit card fraud detection model using Random Forest with undersampling! A high recall is crucial in fraud detection, as it indicates that your model is effectively identifying fraudulent transactions.

```
# !pip install catboost

# from sklearn.ensemble import VotingClassifier
# from sklearn.ensemble import RandomForestClassifier
# from lightgbm import LGBMClassifier
# from catboost import CatBoostClassifier
# from xgboost import XGBClassifier

# # Instantiate individual classifiers
# rf = RandomForestClassifier(n_estimators=100, random_state=42)
# lgbm = LGBMClassifier(objective='binary', random_state=42)
# xgb = XGBClassifier(objective='binary:logistic', random_state=42)
# catboost = CatBoostClassifier(iterations=1000, learning_rate=0.1,
depth=6, random_seed=42, silent=True)

# # Instantiate the Voting Classifier
# voting_clf = VotingClassifier(
#     estimators=[
#         ('rf', rf),
#         ('lgbm', lgbm),
#         ('xgb', xgb),
#         ('catboost', catboost)
#     ],
#     voting='hard'  # Use 'soft' for soft voting
# )

# # Fit the Voting Classifier on the resampled training data
# voting_clf.fit(X_resampled, y_resampled)

# # Use the model to predict on train data
```

```
# voting_train_resampled_preds = voting_clf.predict(X_resampled)

# # Use the model to predict on test data
# voting_test_preds = voting_clf.predict(X_test)

# # Get test results using your scoring function
# voting_test_results = get_test_scores('Voting Classifier (test)',
voting_test_preds, y_test)
# voting_test_results
```