Further Artificial Intelligence Report – Part 2

Module Code	**:**	CT046-3-2-FAI Further Artificial Intelligence
Intake Code	**:**	APU3F2311CS(AI)
Lecturer Name	**:**	Dr. Kuan Yik Junn
Hand in Date	**:**	23 March 2024, before 12:00 AM
Tutorial No.	**:**	6
Group No.	**:**	4

Student ID	**Student Name**
TP065697	LOW SIM CHUAN
TP061839	LIAW YU JAY
TP062192	NIVETHAN A/L RAMESH

Contents

List of Figures

Abstract

This research explores the construction of a prediction model to address the urgent problem of credit card fraud. It does this by using sophisticated machine learning methods to an unbalanced dataset that contains a disproportionate number of fraudulent transactions compared to valid ones. This stands for cardholder transactions in Europe with data coming mostly from Kaggle.

This research takes on the problem of algorithmic bias, which is a problem in fraud detection since it benefits the majority group. The technique, which is based on the CRISP-DM process model, takes a holistic view of the problem by first gaining a thorough comprehension of the company's goals. After that, EDA (exploratory data analysis) follows data preparation and gathering.

An important part of the project's approach is the data preparation step, which involves using methods such as oversampling, undersampling, and the Synthetic Minority Over-sampling Technique (SMOTE) to address imbalances in the dataset and regulate dimensionality using Principal Component Analysis (PCA). At the end of this step, you should have a dataset that is ready for better modelling and analysis.

Building and training a Random Forest model is the meat and potatoes of the undertaking. Because of how well it deals with the intricacies of an unbalanced dataset, it was selected. By tweaking the model's hyperparameters and validating it against a split dataset, this step is painstakingly intended to maximise the model's performance. The model's dependability and robustness are guaranteed by this.

A more comprehensive evaluation of the model's prediction accuracy and generalizability outside the training data might be achieved with the use of confusion matrices and Area Under the ROC Curve (AUC-ROC) as measures. In addition to its stated goal of improving fraud detection accuracy, this research offers helpful insights on dealing with unbalanced datasets. The

significance of meticulous data preparation and the strategic use of machine learning algorithms in the field of financial safety is highlighted by this.

1. Introduction

1.1 Introduction

The internet has completely altered the banking industry in the last few years. We do this by bringing forth a new age of ease and productivity. Having said that, there have been some problems associated with this change. New opportunities for fraud have emerged with the explosion of internet financial operations. This is especially problematic for credit card fraud since it causes problems for both consumers and banks. The development of sophisticated fraud techniques that make use of large and often skewed databases of real and fake transactions has made this problem even more complicated.

An effective prediction model using machine learning methods is the goal of this research, which explores the field of Further Artificial Intelligence (FAI). This approach tries to solve the problems caused by the uneven structure of transactional datasets to detect fraudulent credit card transactions accurately. This effort aims to greatly reduce the risk of credit card theft by using AI and machine learning. The safety of online money transfers is therefore improved.

1.2 Problems

Since there are many more valid transactions than fraudulent ones, credit card fraud is an uneven issue. By incorrectly labelling fraudulent activity as real, traditional detection methods often fail to catch fraud. Big bucks might go if this happens. Moreover, traditional detection methods produce many false positives, which increase operating expenses and undermine client confidence. The difficulty comes from trying to train a machine learning model to accurately differentiate between normal and unusual transaction patterns in a dataset that is unbalanced. In addition to

helping financial institutions achieve their operational and customer service objectives, this also offers dependable fraud detection.

1.3 Objectives

The objectives of the project:

1) To achieve a detection accuracy rate of at least 95% by reducing false negatives by 50% and ensure the model's precision and recall scores are above 90%.

2) To tackle the imbalanced nature of the dataset by incorporating and comparing the efficiency of oversampling, undersampling and SMOTE methods.

3) To quantify the model's performance through a cross-validation process by ensuring that the results are reliable plus the model is robust against overfitting.

1.4 Summary

By using the potential of more AI and ML, this initiative seeks to tackle the urgent problem of credit card theft in the digital age. By delving into the problem's intricacies and comprehending the obstacles presented by things like unbalanced datasets and complicated fraud schemes. An extensive strategy for creating a state-of-the-art prediction model is laid out in this project. Both the detection accuracy and the model's flexibility and efficiency in real-time fraud detection situations are ensured by the carefully established goals. This effort aims to make a substantial difference in the battle against credit card fraud by thorough research, data analysis, and model review. The result is an increase in confidence in digital financial ecosystems and a decrease in fraudulent financial activities.

2. Coding Implementation

2.1 Introduction

This section is about coding implementation of detecting fraudulent transactions, and the coding implementation is broken down into six sections as proposed in Part 1 of this assignment: Exploratory Data Analysis (EDA), Data Preprocessing, Handling Imbalanced Dataset, and Training and Evaluating Machine Learning Models.

2.2 Exploratory Data Analysis (EDA)

```
import pandas as pd
path_ds = "/content/drive/MyDrive/FAI assignment/creditcard.csv"
df = pd.read_csv(path_ds)
print("Shape of the dataset = ", df.shape)
df.head()
```

Figure 1: Loading the dataset from Google Drive

Firstly, the dataset is loaded in the form of ***Pandas dataframe***, and its shape is printed out.

```
import matplotlib.pyplot as plt
df["Class"].value_counts()
```

Figure 2: Getting Distribution Over Classes

Then, the distribution over classes is gotten to inspect if the dataset is imbalanced or not. If the dataset is imbalanced, then further operations are needed to preprocess the dataset.

```
df.hist(figsize=(20,15))
```

Figure 3: Histogram of each feature

For each feature, a histogram is plotted for observing the pattern.

```
[ ]  # for each class, generate the histogram for each feature
     df.groupby('Class').hist(figsize=(20,15))
```

Figure 4: Histogram of each feature by class

The data pattern might differ from class to class, hence in each class, a histogram is plotted for each feature for observing the pattern.

```
# dropping Time attribute as we are not working on time-series forecasting
from pandas.plotting import scatter_matrix
scatter_matrix(df.drop(columns=["Time", "Class"]), alpha=0.2, figsize=(20, 15), diagonal='kde')
```

Figure 5: Scatterplot of each pair of attributes excluding attributes "Time" and "Class"

Then, for each pair of attributes (excluding the attributes “Time” and “Class”), a scatterplot is plotted for bivariate analysis. The attribute “Time” is excluded because the task our team is working on is fraud detection, instead of time-series forecasting, hence the factor is not relevant to the task. The attribute “Class” is excluded because it is either 0 or 1 (discrete variable), for any attribute that pairs with it is not going to create a histogram, but a bar chart.

```
import seaborn as sn
import matplotlib.pyplot as plt
corrMatrix = df.corr()
plt.subplots(figsize=(25,20))
sn.heatmap(corrMatrix, annot=True)
plt.show()
```

Figure 6: Plotting heatmap for each pair of attributes

Heatmap is a good visualization tool for the correlation between each pair of attributes, it encodes different intensity of colors to different magnitude of the correlation. Hence, the correlation matrix of the dataset is first computed, then a heatmap is plotted based on the correlation matrix. In our assumption, highly correlated variables are redundant in this task, hence they will be removed from this dataset.

```
threshold = 0.75  # threshold for highly correlated variables
count_highly_correlated = 0
corrMatrix_numpy = corrMatrix.to_numpy()
for row_index in range(len(corrMatrix_numpy)):
    for col_index in range(len(corrMatrix_numpy[0])):
        if row_index != col_index:
            if corrMatrix_numpy[row_index][col_index] >= threshold:
                print(f"The variables at ({str(row_index)}, {str(col_index)}) are highly correlated!")
                count_highly_correlated = count_highly_correlated + 1
if count_highly_correlated == 0:
    print("There is no variables that are highly correlated!")
```

Figure 7: Checking if there is any highly correlated attributes

The threshold is set to 0.75 as the normally the data science industry will set it at the range from 0.70 to 0.80 (Dorsainvil, 2019). Then, some codes are written to check if there is any highly correlated variables inside the dataset.

```
# check for NULL values and data types
df.info()
```

Figure 8: Check for missing data and data type for each attribute

For each attribute, the data type and the information about missing data is checked. This is for determining whether there is a need for corresponding data preprocessing technique.

```
# Check for NULL values/missing values
round((df.isnull().sum()/df.shape[0]) * 100, 2)
```

Figure 9: Computing percentage of missing data for each attribute

For each attribute, the percentage of missing data is also computed to see which method best suits the dataset. If the percentage of missing data is low, simply removing the rows with missing data is feasible. Otherwise, other methods such as mean imputation and median imputation should be considered based on the data distribution.

```
# check the duplicate rows
count = 0
for duplicated in df.duplicated():
    if duplicated == True:
        count += 1
print("Count of duplicated rows = ", count)
```

Figure 10: Compute the number of duplicated rows

Then, the number of duplicated rows is computed for the entire dataset. If there is duplicated rows, they must be removed from the dataset for shortening the training time of the machine learning algorithm.

```
df = df.drop(columns=['Time'])
df.head(5)
```

Figure 11: Dropping the attribute "Time"

The attribute “Time” is dropped from the dataset as the task is about fraud detection. The task is not about time-series forecasting; hence the attribute “Time” is not relevant.

```
# boxplot to compare the range of each feature (excluding `Amount`) becoz it will affect the other boxplots
# to check for outliers
df.loc[:, ~df.columns.isin(["Amount"])].boxplot(figsize=(18, 20))
```

Figure 12: Boxplot excluding the attribute "Amount"

```
# boxplot for the attribute `Amount`
df.loc[:, df.columns.isin(["Amount"])].boxplot(figsize=(18, 20))
```

Figure 13: Boxplot of the attribute "Amount"

A boxplot is plotted for each attribute of the dataset, the reason of separating the attribute "Amount" from the rest is because of its high variance, high variance can "squeeze" the boxplots of the other attributes. Hence, for better-looking visualization purposes, it is better to separate the attribute "Amount" from the rest.

```
def analyze_outliers_IQR(df, feature_name):
    q1=df[feature_name].quantile(0.25)
    q3=df[feature_name].quantile(0.75)
    IQR=q3-q1
    outliers_samples = df[(df[feature_name]<(q1-1.5*IQR))| (df[feature_name]>(q3+1.5*IQR))]
    print("number of outliers = ", len(outliers_samples[feature_name]))
    print("max outlier value: ", outliers_samples[feature_name].max())
    print("min outlier value:", outliers_samples[feature_name].min())
    class0_outliers_count = len(outliers_samples[outliers_samples["Class"] == 0])
    class1_outliers_count = len(outliers_samples[outliers_samples["Class"] == 1])
    print("% of Fraudulent transactions in outliers = ", class1_outliers_count/(class0_outliers_count+class1_outliers_count))
    print("% of Normal transactions in outliers = ", class0_outliers_count/(class0_outliers_count+class1_outliers_count))
```

```
features = ["V" + str(i) for i in range(1, 29)]
for i in range(28):
    print("For ", features[i], ": ")
    analyze_outliers_IQR(df[[features[i], "Class"]], features[i])
    print()

print("For Amount:")
analyze_outliers_IQR(df[["Amount", "Class"]], "Amount")
```

Figure 14: Analyzing the outliers for each attribute

The outliers are determined based on the first quantile and third quantile, which is any data that falls outside the range (Q1 – 1.5 * IQR, Q3 + 1.5 * IQR). For each attribute, the number of the outliers, the maximum value of the outliers, the minimum value of the outliers, the percentage of class 0 among the outliers, and the percentage of class 1 among the outliers are computed. All these information are important to determine the method of handling these outliers. The outliers in

this case could be beneficial in the case of fraud detection hence simply removing them is not a wise choice (Team, 2020).

2.3 Data Preprocessing

```
# removal of duplicated rows
df = df.drop_duplicates()
print("Shape of df after dropping duplicated rows = ", df.shape)
df["Class"].value_counts()
```

Figure 15: Removing all the duplicated rows

Here comes the stage of data preprocessing, and it can be divided into data cleaning, feature selection, and feature extraction. First, the duplicated rows will be removed from the dataset. Then, the distribution over classes will be checked again to see if the removal of duplicated rows makes the dataset more imbalanced. The shape of the dataset will be checked as well for ensuring the duplicated rows have been removed completely from the dataset.

```
def drop_Class0_outliers_IQR(df, feature_name):
    q1=df[feature_name].quantile(0.25)
    q3=df[feature_name].quantile(0.75)
    IQR=q3-q1
    outliers_samples = df[(df[feature_name]<(q1-1.5*IQR))| (df[feature_name]>(q3+1.5*IQR))]
    outliers_class0_index = outliers_samples[outliers_samples["Class"] == 0].index
    dropped_df = df.drop(outliers_class0_index)
    print(f"Dropped {str(len(df) - len(dropped_df))} rows!")
    return dropped_df
```

```
for attribute in df.columns:
    print(f"Dropping the outliers that belong to class 0 for the attribute {attribute}...")
    df = drop_Class0_outliers_IQR(df, attribute)
```

Figure 16: Remove outliers that belong to class 0 for each attribute

For each attribute, the outliers that belong to class 0 will be removed from the dataset. The outliers that belong to class 1 is kept because they might reflect the occurrence of some rare events, in this case it is fraudulent transactions.

```
from sklearn.feature_selection import VarianceThreshold
X = df.drop(columns=["Class"])
y = df["Class"]
selector = VarianceThreshold(threshold=0.1)
X_variance_thresholded = selector.fit_transform(X)
print("Shape of feature matrix before variance thresholding = ", X.shape)
print("Shape of feature matrix after variance thresholding = ", X_variance_thresholded.shape)
X_variance_thresholded
```

Figure 17: Variance thresholding by setting the threshold as 0.1

Feature selection on the dataset is then performed. In this task, variance thresholding is performed on the dataset. If the attribute has a variance lower than 0.1, then the attribute is dropped. Then, the shape of feature matrix before and after applying the variance thresholding is printed to see how many attributes have been dropped.

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)  # for visualization purposes, in terms of 2D scatterplot
X_pca = pca.fit_transform(X=X_variance_thresholded)

# Store as dataframe and print
X_pca = pd.DataFrame(X_pca)
print(X_pca.shape)  #> ((110052, 2))
X_pca.round(2).head()
```

Figure 18: Creating 2 principal components

Then, feature extraction is performed, specifically Principal Component Analysis (PCA) for reducing the dimension of the dataset. In this case, the two principal components with largest and second largest variance are kept, two principal components are picked for visualization purposes. Then, the shape of the dataset is checked to confirm that the PCA operation is successful.

```
X_pca.var()
```

```
0    1529.153246
1       2.667573
dtype: float64
```

```
print(pca.explained_variance_ratio_.round(4)[:10])
```

Figure 19: Checking variance and contributions of each principal component to the total variance

The variance of each principal component is printed, and the contributions of each principal component to the total variance are printed in terms of percentage.

```
import numpy as np
colormap = np.array(['r', 'g'])
plt.scatter(X_pca[0], X_pca[1], s=5, c=colormap[y])
plt.show()
```

Figure 20: Plotting scatterplot using the 2 principal components

Finally, the scatterplot is used to plot the dataset using the 2 principal components. The points are coloured based on their classes to see if there is clear distinction between the classes.

2.4 Handle Imbalanced Dataset

Handling Imbalanced Dataset

```
[29] from IPython.display import display
     import pandas as pd
     from imblearn.over_sampling import RandomOverSampler
     from imblearn.under_sampling import RandomUnderSampler
     from imblearn.over_sampling import SMOTE

     # Print the shape of the original dataset
     print("Original dataset shape:", X_pca.shape)

     # Initialize the RandomOverSampler object
     ros = RandomOverSampler(random_state=0)
     X_ros, y_ros = ros.fit_resample(X_pca, y)
     print("Oversampled dataset shape:", X_ros.shape)

     # Display the oversampled dataset
     print("Oversampled dataset (first 5 samples):")
     display(pd.DataFrame(X_ros).head())
     display(pd.Series(y_ros).value_counts())

     # Initialize the RandomUnderSampler object
     rus = RandomUnderSampler(random_state=0)
     X_rus, y_rus = rus.fit_resample(X_pca, y)
     print("Undersampled dataset shape:", X_rus.shape)
```

Figure 21

```
# Display the undersampled dataset
print("Undersampled dataset (first 5 samples):")
display(pd.DataFrame(X_rus).head())
display(pd.Series(y_rus).value_counts())

# Initialize the SMOTE object
smote = SMOTE(random_state=0)
X_smote, y_smote = smote.fit_resample(X_pca, y)
print("SMOTE dataset shape:", X_smote.shape)

# Display the SMOTE dataset
print("SMOTE dataset (first 5 samples):")
display(pd.DataFrame(X_smote).head())
display(pd.Series(y_smote).value_counts())

# Now, X_ros, y_ros are the oversampled features and labels,
# X_rus, y_rus are the undersampled features and labels,
# X_smote, y_smote are the features and labels after applying SMOTE.

# The "API" for oversampling is represented by the use of RandomOverSampler.
# The "API" for undersampling is represented by the use of RandomUnderSampler.
# The "API" for SMOTE is represented by the use of SMOTE.
```

Figure 22

To deal with an unbalanced dataset, the supplied code is a Python script that makes use of the imbalanced-learn module. This is a typical problem with machine learning, and it is especially prevalent in areas like fraud detection. Python for data manipulation and ‘Imblearn’ for resampling approaches are two of the key libraries that are first loaded. Starting with a brief description of the dataset's size after dimensionality reduction by PCA, the script prints the form of the pre-processed data.

Using the 'RandomOverSampler' class, the first resampling approach is Random Oversampling. This strategy increases the minority group's strength by randomly increasing its sample size until it reaches the same level as the majority group. This may assist in giving the model a more accurate picture of the classes overall. However, it also increases the likelihood of overfitting if data from underrepresented classes are repeated.

The next step involves utilizing the 'RandomUnderSampler' class to execute Random Undersampling. The count of the minority class is equal to the number of samples discarded at

random, which decreases the majority class. The problem of imbalance is successfully addressed, although the majority class may have valuable data that is lost as a result.

Finally, the script uses SMOTE, which stands for Synthetic Minority Over-sampling Technique, to create new minority class samples. This method creates artificial samples by combining characteristics of the minority class. The goal is to make this class's feature space representation broader.

Showing the resampled datasets and giving the counts for each class is the last thing the code does. This checks that the resampling procedures were carried out correctly.

2.5 Model Training and Evaluation

Model Training and Evaluation (Random Forest Classifier)

Import Relevant Packages

```
# Data Processing
import pandas as pd
import numpy as np

# Modelling
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, ConfusionMatrixDisplay
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from scipy.stats import randint

# Tree Visualisation
from sklearn.tree import export_graphviz
from IPython.display import Image
import graphviz
```

Figure 23 Importing Library

The figure above shows the steps of importing relevant packages that are used to train and evaluate the random forest classifier model credit fraud detection. It can see that, libraries such as pandas which is important for data manipulation, analysis and computation, and NumPy that is important for numerical operations are imported (W3schools, n.d.). Most importantly, sklearn learn libraries which is a free machine learning library that assists in performing machines learning processes is imported. It can be seen that ***RandomForestClassifier*** and relevant operations like metrics, algorithm for perform hyperparameter tuning and ***train_test_split*** are also imported.

```
[ ] # Split the data into training and test sets for oversample dataset
    X_ros_train, X_ros_test, y_ros_train, y_ros_test = train_test_split(X_ros, y_ros, test_size=0.2)

    # Split the data into training and test sets for undersample dataset
    X_rus_train, X_rus_test, y_rus_train, y_rus_test = train_test_split(X_rus, y_rus, test_size=0.2)

    # Split the data into training and test sets for SMOTE dataset
    X_smote_train, X_smote_test, y_smote_train, y_smote_test = train_test_split(X_smote, y_smote, test_size=0.2)
```

Figure 24 Performing data splitting

The above figure shows the process of performing splitting of dataset into training split and testing split using the using the ***train_test_split*** function provided in the sklearn API. In this process, ***train_test_split*** is performed on three types of datasets which are the oversample dataset, undersample dataset and the SMOTE dataset that are obtained from the previous process of

handling imbalanced dataset. The ratio is set to 0.2 for test_size which means 20 percent of the dataset will be used to testing and the other 80 percent will be used for training.

Fitting and Evaluating the Model

```
[ ]
    rf_ros = RandomForestClassifier()
    rf_ros.fit(X_ros_train, y_ros_train)

    rf_rus = RandomForestClassifier()
    rf_rus.fit(X_rus_train, y_rus_train)

    rf_smote = RandomForestClassifier()
    rf_smote.fit(X_smote_train, y_smote_train)
```

Figure 25 Fitting and Evaluating the model

The figure above shows the code to train the Random Forest Classifier on the three different datasets. In the code, an instance of the RandomForestClassifier class is created and stored into a variable. This variable now represents the classifier itself. After that, the ***.fit()*** method found in the classifier class is used to train the model where the X dataset that contains the training features and Y that contains the target variable are supplied to the method. This process is repeated three times to generate a trained classifier for all three datasets. After this process, the ***rf_ros*** represents a trained classifier for the oversampled dataset, the ***rf_rus*** represents a trained classifier for the undersampled dataset, and the ***rf_smote*** represents a classifier for the SMOTE dataset.

Making predictions using the trained models

```
[ ]
    y_pred_ros = rf_ros.predict(X_ros_test)

    y_pred_rus = rf_rus.predict(X_rus_test)

    y_pred_smote = rf_smote.predict(X_smote_test)
```

Figure 26 Making predictions

The code above represents the process of making predictions on the three different test datasets using the trained model from above. In the code, the predict method of the classifier object is provided with the test datasets of the three different categories. Then, it will perform prediction on the test datasets when executed. The results of prediction that stores the predicted class are stored in a variable for all three predictions.

Evaluating the results

```
# Create the confusion matrix for oversampling dataset
cm_ros = confusion_matrix(y_ros_test, y_pred_ros)

disp_ros = ConfusionMatrixDisplay(confusion_matrix= cm_ros).plot();

# Set the title
title = "Confusion Matrix of Oversampling dataset"
disp_ros.ax_.set_title(title)

# Create the confusion matrix for undersampling dataset
cm_rus = confusion_matrix(y_rus_test, y_pred_rus)

disp_rus = ConfusionMatrixDisplay(confusion_matrix= cm_rus).plot();

# Set the title
title = "Confusion Matrix of Undersampling dataset"
disp_rus.ax_.set_title(title)


# Create the confusion matrix for smote dataset
cm_smote = confusion_matrix(y_smote_test, y_pred_smote)

disp_smote = ConfusionMatrixDisplay(confusion_matrix= cm_smote).plot();

# Set the title
title = "Confusion Matrix of SMOTE dataset"
disp_smote.ax_.set_title(title)
```

Figure 27 Evaluating the results

The code above shows the process of model evaluation for the trained classifiers based on the predictions that they made above. The evaluation metrics chosen for this project is confusion matrix due to its suitability in handling imbalanced datasets (Jayawardena, 2020). In the code above, the confusion-matrix function form scikit-learn is deployed. The variable that stored the predicted result from the trained model and the actual labels of each dataset are passed into the ***confusion_matrix*** method which will calculate the true positives, true negative, false positives and false negatives predictions. Then, a series of codes is deployed to visualize the result of confusion matrix.

Visualisation of trees

1. Oversampling dataset

```
# Export the first three decision trees from the oversampling forest

for i in range(3):
    tree = rf_ros.estimators_[i]
    dot_data = export_graphviz(tree,
                               feature_names=X_ros_train.columns,
                               filled=True,
                               max_depth=2,
                               impurity=False,
                               proportion=True)
    graph = graphviz.Source(dot_data)
    display(graph)
```

Figure 28 code to display first three decision trees (oversampling)

2. Undersampling dataset

```
# Export the first three decision trees from the undersampling forest

for i in range(3):
    tree = rf_rus.estimators_[i]
    dot_data = export_graphviz(tree,
                               feature_names=X_rus_train.columns,
                               filled=True,
                               max_depth=2,
                               impurity=False,
                               proportion=True)
    graph = graphviz.Source(dot_data)
    display(graph)
```

Figure 29 code to display first three decision trees (undersampling)

3. SMOTE dataset

```
# Export the first three decision trees from the smote forest

for i in range(3):
    tree = rf_smote.estimators_[i]
    dot_data = export_graphviz(tree,
                               feature_names=X_smote_train.columns,
                               filled=True,
                               max_depth=2,
                               impurity=False,
                               proportion=True)
    graph = graphviz.Source(dot_data)
    display(graph)
```

Figure 30 code to display first three decision trees (smote)

The code above is a loop that exports and displays the first three decision trees from the three decision tree forests which are oversampling, undersampling, and smote. The collection of decision trees in the random forest classifier is represented by the rf_ros.estimators_ attribute, from which a decision tree object is fetched in each iteration of the loop. The decision tree is then represented as a graph in the DOT language format using the export_graphviz function. This code

aims to illustrate the first three decision trees' structure in the three random forest classifiers, shedding light on the classifier's decision-making process and the relative importance of various characteristics.

Hyperparameter tuning

```
from sklearn.metrics import make_scorer, precision_score
param_dist = {'n_estimators': [50, 100, 150, 250, 300],
              'max_depth':[4, 8, 12, 16 ,20]}

# Create a random forest classifier
rf = RandomForestClassifier()


# Define the scoring metric
scoring_metric = make_scorer(precision_score)

# Use random search to find the best hyperparameters with precision as the scoring metric
rand_search = RandomizedSearchCV(rf,
                                 param_distributions=param_dist,
                                 n_iter=5,
                                 cv=3,
                                 scoring=scoring_metric)

# Fit the random search object to the data
rand_search.fit(X_ros_train, y_ros_train)
```

Figure 31: Hyperparameter tuning (oversampling dataset)

```
# Create a variable for the best model
best_rf_ros = rand_search.best_estimator_

# Print the best hyperparameters
print('Best hyperparameters:',  rand_search.best_params_)
```

Figure 32: Storing of best model

```
from sklearn.metrics import make_scorer, precision_score
param_dist = {'n_estimators': [50, 100, 150, 250, 300],
              'max_depth':[4, 8, 12, 16 ,20]}

# Create a random forest classifier
rf = RandomForestClassifier()


# Define the scoring metric
scoring_metric = make_scorer(precision_score)

# Use random search to find the best hyperparameters with precision as the scoring metric
rand_search = RandomizedSearchCV(rf,
                                 param_distributions=param_dist,
                                 n_iter=5,
                                 cv=3,
                                 scoring=scoring_metric)

# Fit the random search object to the data
rand_search.fit(X_rus_train, y_rus_train)
```

Figure 33 Hyperparameter tuning (undersampling dataset)

```
from sklearn.metrics import make_scorer, precision_score
param_dist = {'n_estimators': [50, 100, 150, 250, 300],
              'max_depth':[4, 8, 12, 16 ,20]}

# Create a random forest classifier
rf = RandomForestClassifier()


# Define the scoring metric
scoring_metric = make_scorer(precision_score)

# Use random search to find the best hyperparameters with precision as the scoring metric
rand_search = RandomizedSearchCV(rf,
                                 param_distributions=param_dist,
                                 n_iter=5,
                                 cv=3,
                                 scoring=scoring_metric)

# Fit the random search object to the data
rand_search.fit(X_smote_train, y_smote_train)
```

Figure 34 Hyperparameter tuning (smote dataset)

The above figures show the code for hyperparameter tuning for the three classifiers on the oversampling dataset. A random search algorithm is deployed in this project for hyperparameter tuning of the random forest classifier. A total of 25 combinations were provided for this case, where 5 values are defined for n_estimators parameter and another 5 values are defined for the max_depth parameter. The precision_score is used as the evaluation matrix in this case instead of accuracy because it is better in handling imbalanced datasets, The n_iter parameter is set to 5 which means only 5 combinations from the 25 combinations provided will be tested. The best model obtained from the hyperparameter tuning is stored in a variable. Then, the best combination obtained for each dataset is recorded and printed using the rand_search.best_params_.

```
# Split the data into training and test sets for original
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create a Random Forest classifier for original dataset
rf_ros = RandomForestClassifier(n_estimators=250, max_depth = 20)

rf_rus = RandomForestClassifier(n_estimators=250, max_depth = 4)

rf_smote = RandomForestClassifier(n_estimators=100, max_depth = 12)

rf_ori_ros = RandomForestClassifier(n_estimators=250, max_depth = 20)

rf_ori_rus = RandomForestClassifier(n_estimators=250, max_depth = 4)

rf_ori_smote = RandomForestClassifier(n_estimators=100, max_depth = 12)

rf_ori_ros.fit(X_train, y_train)

rf_ori_rus.fit(X_train, y_train)

rf_ori_smote.fit(X_train, y_train)

rf_rus.fit(X_ros_train, y_ros_train)

rf_ros.fit(X_rus_train, y_rus_train)

rf_smote.fit(X_smote_train, y_smote_train)
```

Figure 35 Model training using hyperparameters from obtained from hyperparameter tuning

The code above shows that an instance of RandomForestClassifier is created for each of the datasets based on the best combination of hyperparameters obtained for each of the datasets above. An additional of three instances are also declared with the same hyperparameter combinations of the three datasets to train the model with those parameters on the original dataset. It can be seen that, the models are trained with the respective datasets using the .fit() method.

```
# Generate predictions with the best model for oversampling on oversampling dataset
y_pred_ros = rf_ros.predict(X_ros_test)

# Generate predictions with the best model for oversampling on undersampling dataset
y_pred_rus = rf_rus.predict(X_rus_test)

# Generate predictions with the best model for oversampling on smote dataset
y_pred_smote = rf_smote.predict(X_smote_test)

# Generate predictions with the best model for oversampling on original dataset
y_pred_ros_ori = rf_ori_ros.predict(X_test)

# Generate predictions with the best model for oversampling on original dataset
y_pred_rus_ori = rf_ori_rus.predict(X_test)

# Generate predictions with the best model for oversampling on original dataset
y_pred_smote_ori = rf_ori_smote.predict(X_test)
```

Figure 36 Generate prediction with the trained model from hyperparameter tuning

The code above shows again the process of generating predictions on the test set for the three different datasets but this time using the classifiers that have been trained with hyperparameters settings obtained from the hyperparameter tuning processes above. The first lines of code show the code for predicting on the test set for undersampling, oversampling, and smote dataset. The last three lines of code, show the use of classifiers that are trained with the same hyperparameter settings as the previous three lines of code but on the original training dataset.

```
[ ]  # Create the confusion matrix for oversampling dataset
     cm_ros = confusion_matrix(y_ros_test, y_pred_ros)

     disp_ros = ConfusionMatrixDisplay(confusion_matrix= cm_ros).plot();

     # Set the title
     title = "Confusion Matrix of Oversampling dataset"
     disp_ros.ax_.set_title(title)

     # Create the confusion matrix for undersampling dataset
     cm_rus = confusion_matrix(y_rus_test, y_pred_rus)

     disp_rus = ConfusionMatrixDisplay(confusion_matrix= cm_rus).plot();

     # Set the title
     title = "Confusion Matrix of Undersampling dataset"
     disp_rus.ax_.set_title(title)


     # Create the confusion matrix for smote dataset
     cm_smote = confusion_matrix(y_smote_test, y_pred_smote)

     disp_smote = ConfusionMatrixDisplay(confusion_matrix= cm_smote).plot();

     # Set the title
     title = "Confusion Matrix of SMOTE dataset"
     disp_smote.ax_.set_title(title)


     # Create the confusion matrix for original dataset using hyparameter of ros
     cm_ori = confusion_matrix(y_test, y_pred_ros_ori)

     disp_smote = ConfusionMatrixDisplay(confusion_matrix= cm_ori).plot();

     # Set the title
     title = "Confusion Matrix of original dataset using hyparameter of ros "
     disp_smote.ax_.set_title(title)

     # Create the confusion matrix for original dataset using hyparameter of rus
     cm_ori = confusion_matrix(y_test, y_pred_rus_ori)

     disp_smote = ConfusionMatrixDisplay(confusion_matrix= cm_ori).plot();

     # Set the title
     title = "Confusion Matrix of original dataset using hyparameter of rus "
     disp_smote.ax_.set_title(title)

     # Create the confusion matrix for original dataset using hyparameter of smote
     cm_ori = confusion_matrix(y_test, y_pred_smote_ori)

     disp_smote = ConfusionMatrixDisplay(confusion_matrix= cm_ori).plot();

     # Set the title
     title = "Confusion Matrix of original dataset sing hyparameter of smote"
     disp_smote.ax_.set_title(title)
```

Figure 37 Confusion Matrix for all six predictions

The code above shows the last line part of the code where the same process is repeated as above where the result of predictions made by the respective classifiers are evaluated against the actual labels of the dataset using the confusion matrix. The results of each confusion matrix are displayed for clear visualization of the researchers.

3.Results

3.1 Introduction

The execution results of all the coding implementations are shown by different phase in the pipeline of machine learning.

3.2 Results and Analysis

Exploratory Data Analysis (EDA)

```
import pandas as pd
path_ds = "/content/drive/MyDrive/FAI assignment/creditcard.csv"
df = pd.read_csv(path_ds)
print("Shape of the dataset = ", df.shape)
df.head()
```

```
Shape of the dataset =  (284807, 31)
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

5 rows × 31 columns

Figure 38: Shape of the dataset and first 5 rows

The dataset has been loaded successfully from Google Drive. Then, the shape of the dataset showed that the dataset consists of 30 features and 1 target variable, and there are 284807 rows.

```
import matplotlib.pyplot as plt
df["Class"].value_counts()
```

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

Figure 39: Imbalanced Dataset

Then, the results of the distribution over classes showed that there are 284315 normal transactions, while there are only 492 fraudulent transactions. This kind of big gap shows that there is a need for techniques to handle the imbalanced dataset, such as oversampling, undersampling, and SMOTE, otherwise the machine learning algorithm cannot perform well at detecting fraudulent transactions due to its relatively little amount of data.

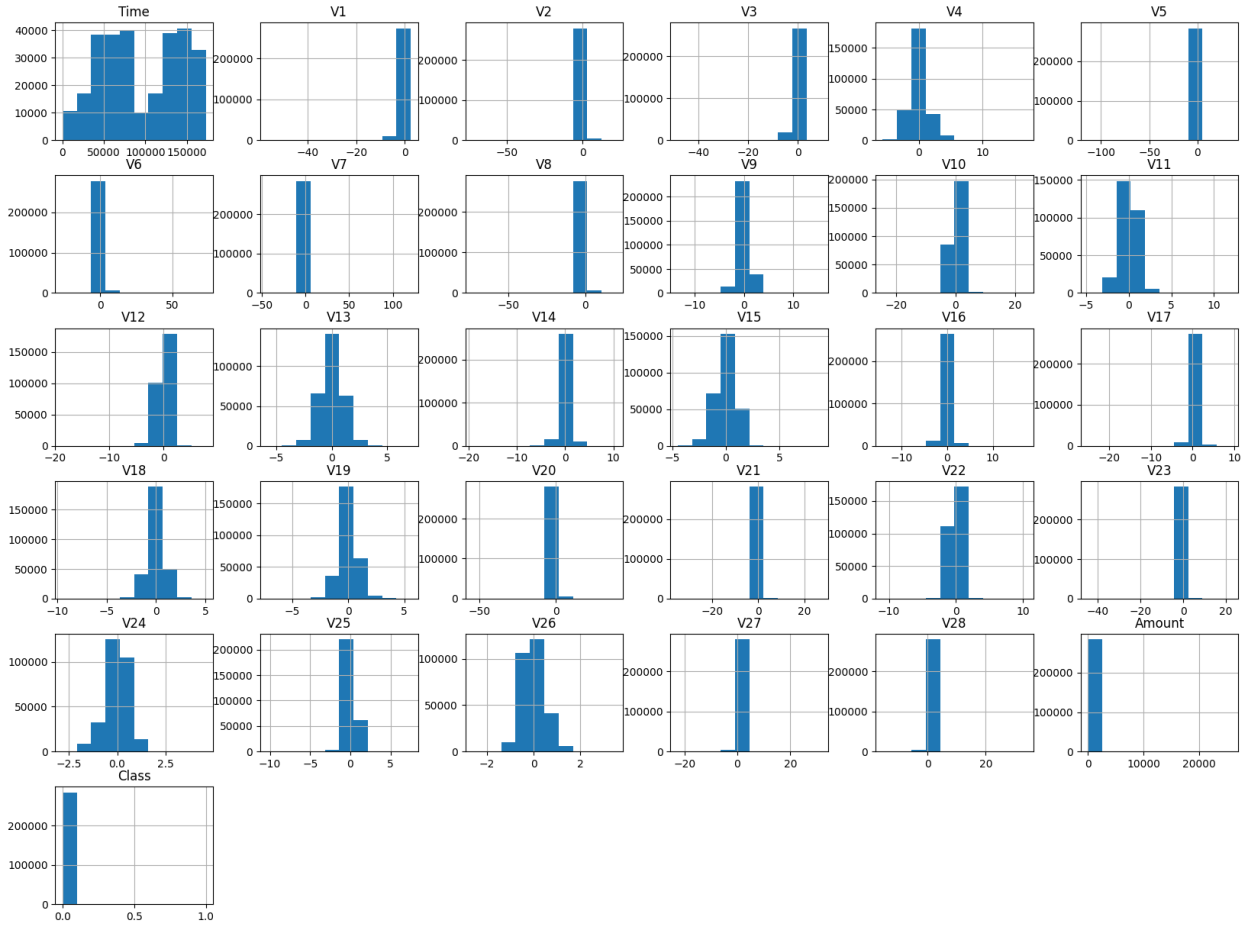


Figure 40: Histogram for each feature

From the histograms above (excluding the columns “Time” and “Class”), “V1”, “V2”, “V3”, “V5”, “V8”, “V12”, “V17”, “V18”, and “V23” are showing left-skewed distribution. Other than that, “V4”, “V6”, “V7”, “V11”, “V15”, “V24”, “V26”, “V28” and “Amount” are showing right-skewed distribution. To deal with this kind of skewed distributions, several transformation methods can be applied such as square root, natural log, and log base 10, so that they can be transformed into nearly normal distributions (Turney, 2022). But our team proposed Random Forest for fraud detection, and Random Forest does not assume the type of data distribution to be a normal distribution, hence there is no need to apply the data transformations on the skewed features (kingJulian, 2019).

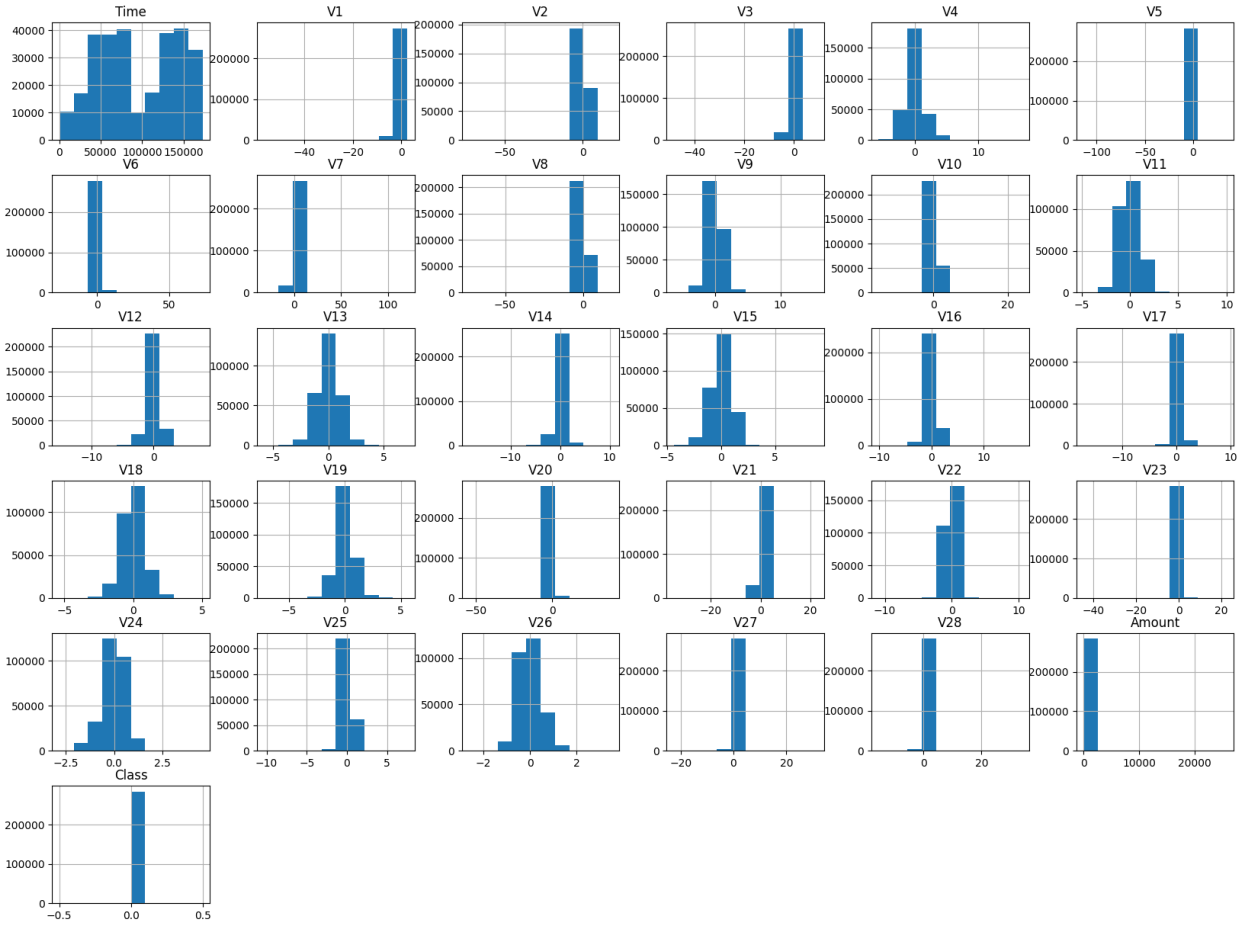


Figure 41: Histogram for each feature in class 0

From the histograms above (excluding the columns “Time” and “Class”), most of the features are having low variances as their histograms are not spread out. For the feature “Amount” among normal transactions, most of them are lower than around 2000. But there are still some extreme values among normal transactions because the maximum amount can go up to 20000.

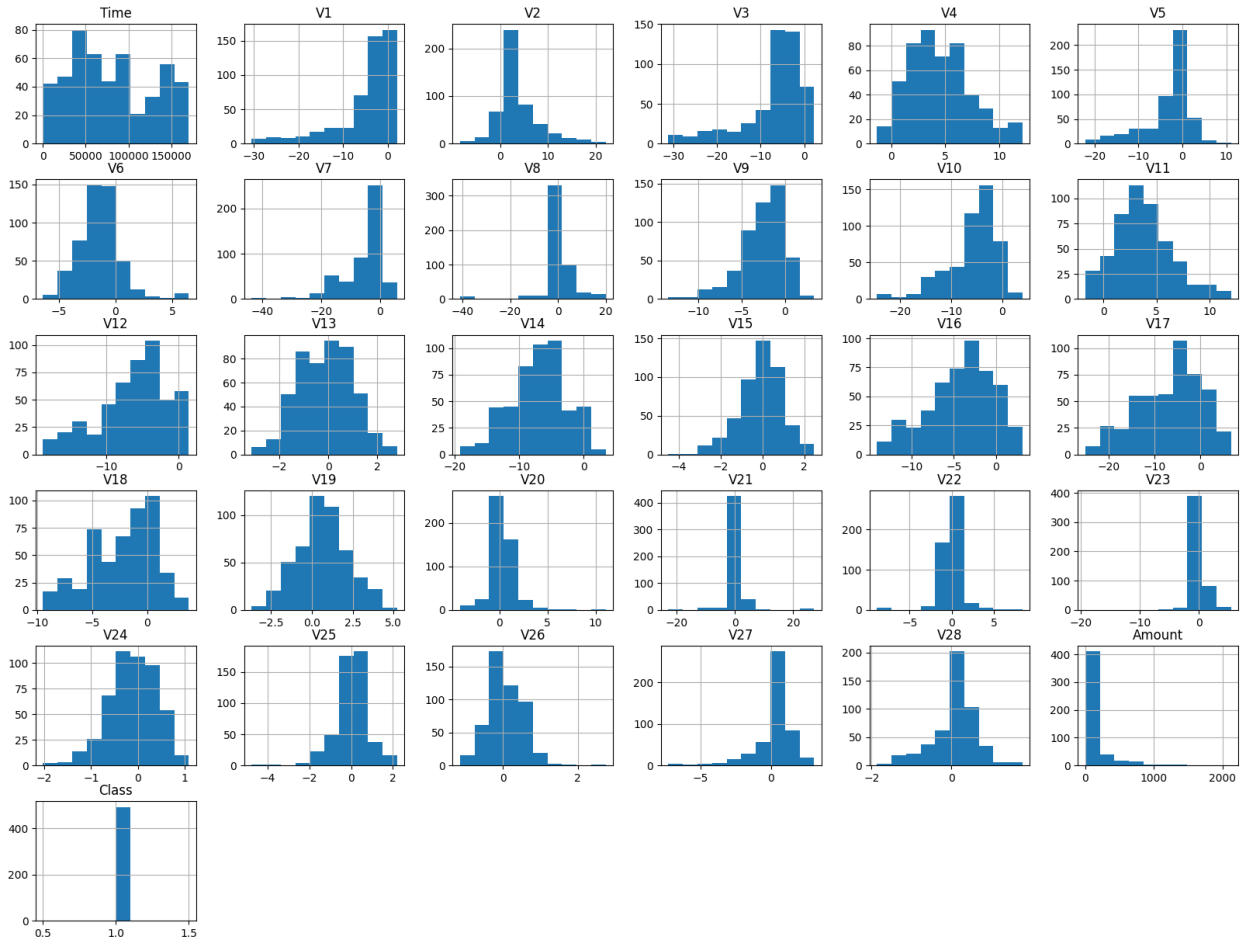


Figure 42: Histogram for each feature in class 1

From the histograms above, fraudulent transactions seem to have much higher variances than normal transactions as the histograms for most of the features are quite spread out. For the transaction amount, most of the fraudulent transactions have an amount lower than around 200, and the extreme values here are less exaggerated than normal transactions. It seems that the general logic "Fraudulent transactions generally involve high amount" does not apply in this dataset.

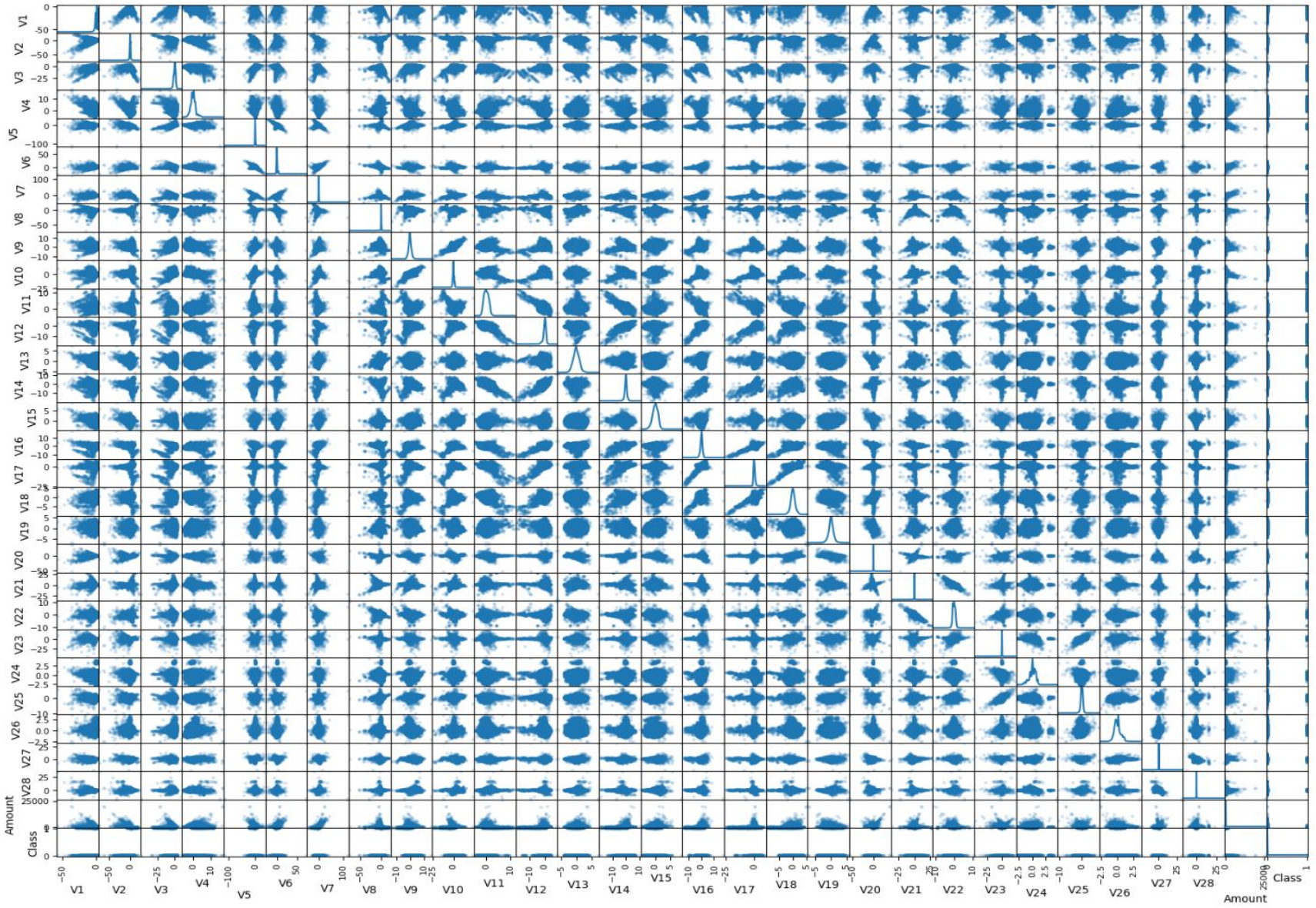


Figure 43: Bivariate Analysis – Scatterplot

From the histograms of each pair of features, there are several positive correlations such as between “V17” and “V18”, between “Amount” and “V7” etc. There are also several negative correlations such as between “V4” and “V1”, between “V5” and “V6” etc. Since all the feature names are already hidden due to confidentiality purposes, there is not much analysis can be done here.

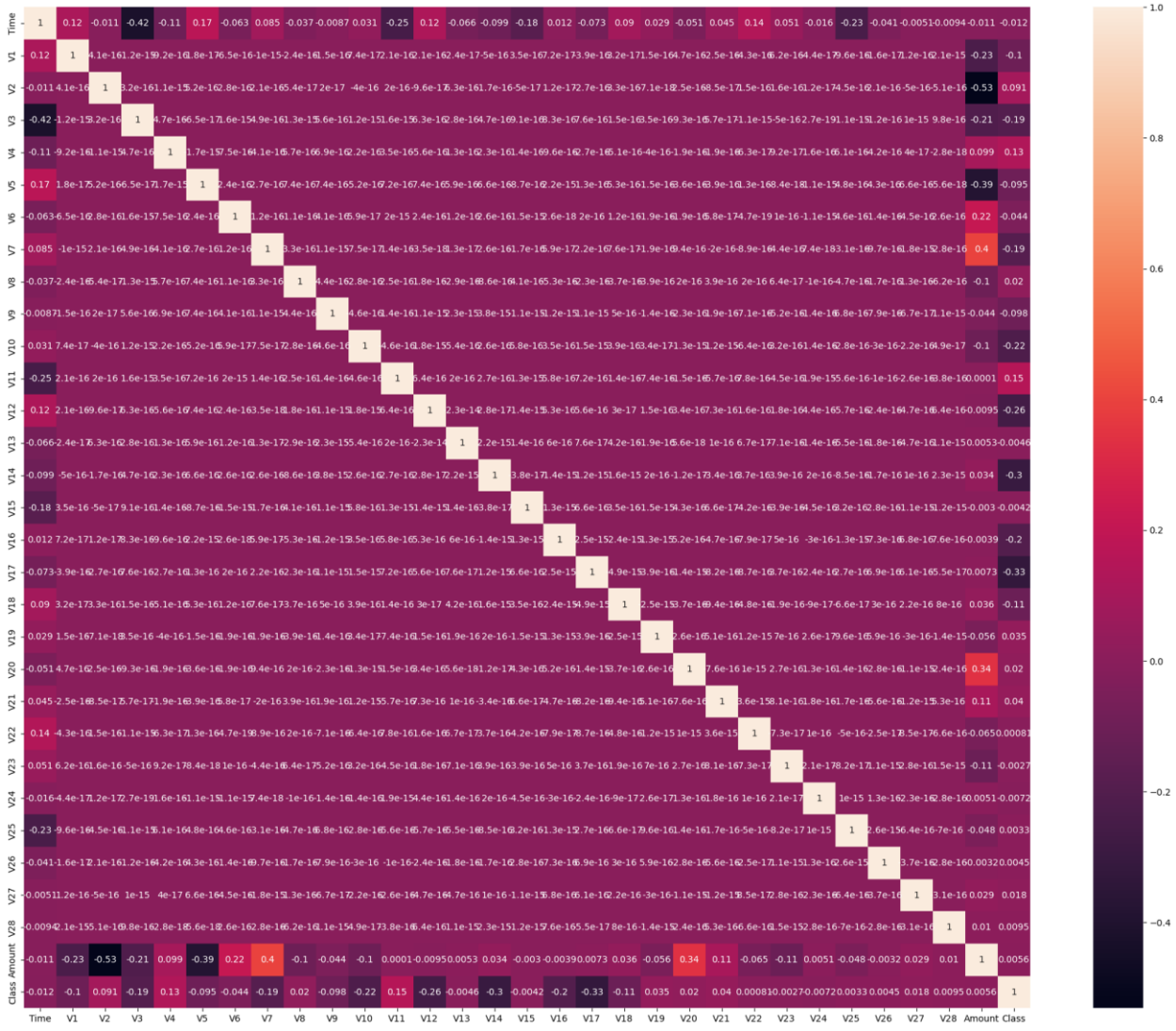


Figure 44: Heatmap

From the heatmap above, there are no boxes in white other than the diagonal of the matrix. It simply means that the correlations of each pair of attributes in this dataset are not high.

```
threshold = 0.75  # threshold for highly correlated variables
count_highly_correlated = 0
corrMatrix_numpy = corrMatrix.to_numpy()
for row_index in range(len(corrMatrix_numpy)):
    for col_index in range(len(corrMatrix_numpy[0])):
        if row_index != col_index:
            if corrMatrix_numpy[row_index][col_index] >= threshold:
                print(f"The variables at ({str(row_index)}, {str(col_index)}) are highly correlated!")
                count_highly_correlated = count_highly_correlated + 1
if count_highly_correlated == 0:
    print("There is no variables that are highly correlated!")
```

```
There is no variables that are highly correlated!
```

Figure 45: No highly correlated variables

For thorough checking, a piece of code is written to check the correlations of each pair of attributes whether they exceed the threshold our team has set, which is 0.75. The final result shows that there is no highly correlated variables in this dataset.

```
# check for NULL values and data types
df.info()
```

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

Figure 46: Data types and information about missing data in each feature

From the results above, the data types of all columns are floating point numbers, except for the target column “Class”, which is integer. All of the columns do not have any missing data.

```
# Check for NULL values/missing values
round((df.isnull().sum()/df.shape[0]) * 100, 2)
```

```
Time      0.0
V1        0.0
V2        0.0
V3        0.0
V4        0.0
V5        0.0
V6        0.0
V7        0.0
V8        0.0
V9        0.0
V10       0.0
V11       0.0
V12       0.0
V13       0.0
V14       0.0
V15       0.0
V16       0.0
V17       0.0
V18       0.0
V19       0.0
V20       0.0
V21       0.0
V22       0.0
V23       0.0
V24       0.0
V25       0.0
V26       0.0
V27       0.0
V28       0.0
Amount    0.0
Class     0.0
dtype: float64
```

Figure 47: Percentages of missing data in each feature

After computing the percentage of missing data for each column, all the percentages are 0%. Hence, there is no need to handle missing data by applying various methods such as removing missing data, mean imputation, and median imputation.

```
# check the duplicate rows
count = 0
for duplicated in df.duplicated():
    if duplicated == True:
        count += 1
print("Count of duplicated rows = ", count)
```

```
Count of duplicated rows =  1081
```

Figure 48: Count of duplicated rows

After counting the number of duplicated rows, there are 1081 duplicated rows in this dataset. Hence, they need to be removed from the dataset to speed up the training of the Random Forest model.

```
df = df.drop(columns=['Time'])
df.head(5)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

5 rows × 30 columns

Figure 49: Dropping the column "Time"

The column “Time” is dropped because this task is about fraud detection, instead of time-series forecasting. The column “Time” is not relevant to the task.

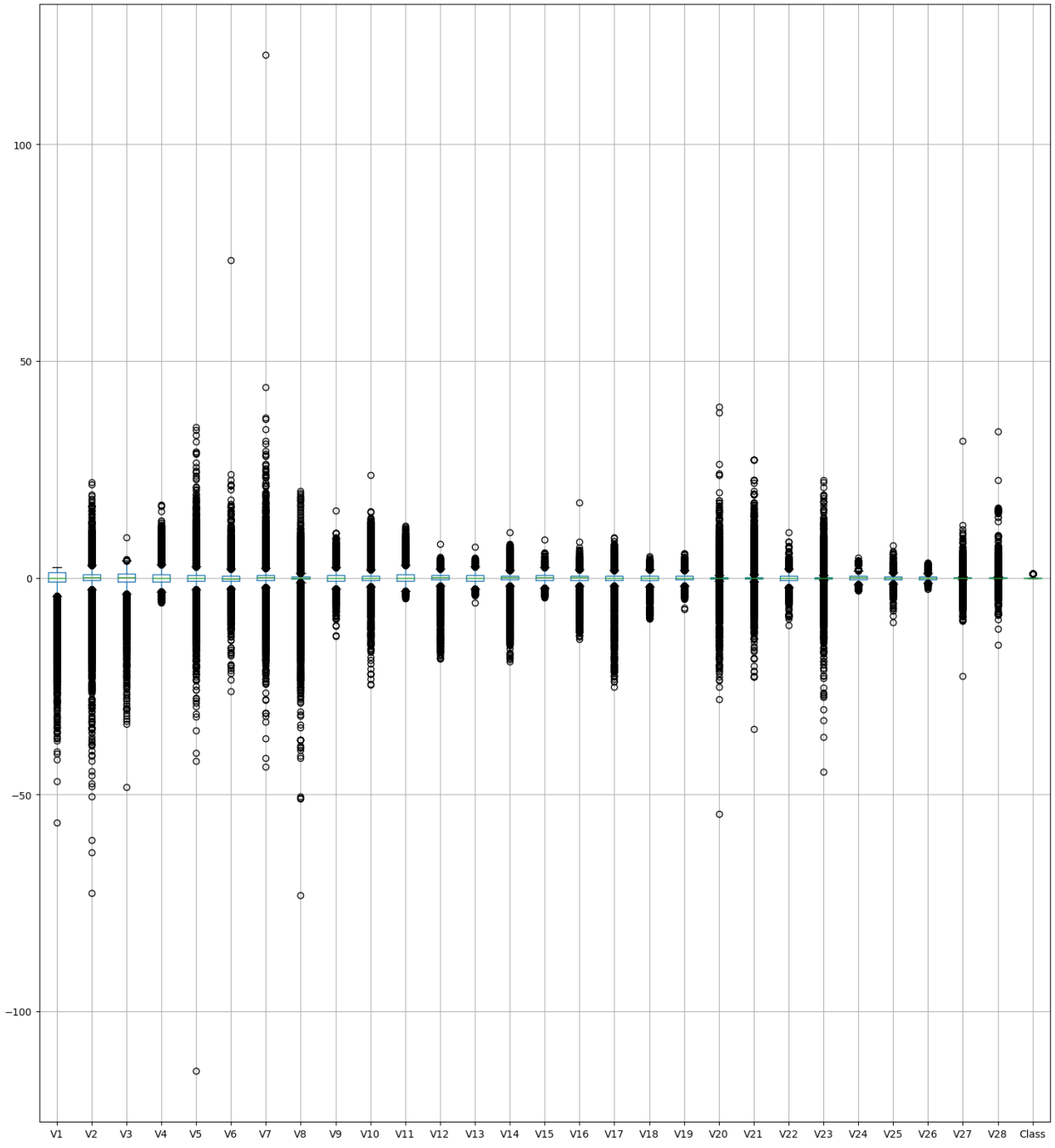


Figure 50: Boxplot excluding the column "Amount"

For the boxplots of the columns except the column “Amount”, there are a lot of outliers for all the columns. From the boxplots, all columns do not have too much difference in terms of the range of the data.

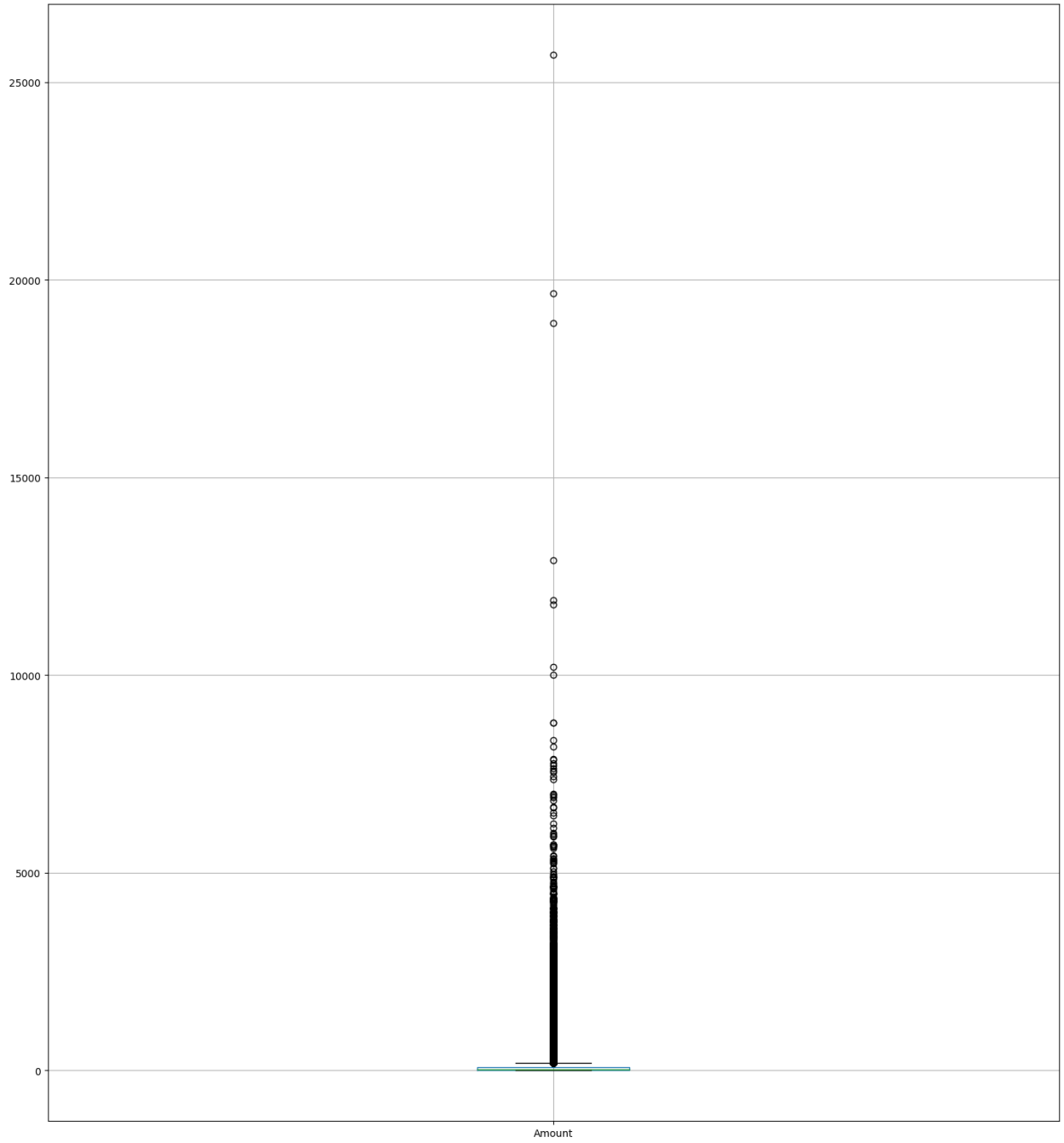


Figure 51: Boxplot for the column "Amount"

For the boxplot of the column “Amount”, there are also a lot of outliers, and the range of the data is too large until the box representing the Interquartile Range appears a lot smaller on the boxplot. The huge amount of the outliers indicates that our team needs to handle the outliers gracefully to prevent the Random Forest from overfitting.

```
For  V1 :
number of outliers =  7062
max outlier value:  -4.27462768510267
min outlier value: -56.407509631329
% of Fraudulent transactions in outliers =  0.02463891248937978
% of Normal transactions in outliers =  0.9753610875106202

For  V2 :
number of outliers =  13526
max outlier value:  22.0577289904909
min outlier value: -72.7157275629303
% of Fraudulent transactions in outliers =  0.018187195031790625
% of Normal transactions in outliers =  0.9818128049682093

For  V3 :
number of outliers =  3363
max outlier value:  9.38255843282114
min outlier value: -48.3255893623954
% of Fraudulent transactions in outliers =  0.0927743086529884
% of Normal transactions in outliers =  0.9072256913470116

For  V4 :
number of outliers =  11148
max outlier value:  16.8753440335975
min outlier value: -5.68317119816995
% of Fraudulent transactions in outliers =  0.02852529601722282
% of Normal transactions in outliers =  0.9714747039827771

For  V5 :
number of outliers =  12295
max outlier value:  34.8016658766686
min outlier value: -113.743306711146
% of Fraudulent transactions in outliers =  0.016917446116307442
% of Normal transactions in outliers =  0.9830825538836926

For  V6 :
number of outliers =  22965
max outlier value:  73.3016255459646
min outlier value: -26.1605059358433
% of Fraudulent transactions in outliers =  0.005965599825821903
% of Normal transactions in outliers =  0.9940344001741781

For  V7 :
number of outliers =  8948
max outlier value:  120.589493945238
min outlier value: -43.5572415712451
% of Fraudulent transactions in outliers =  0.033638801966919984
% of Normal transactions in outliers =  0.9663611980330801

For  V8 :
number of outliers =  24134
max outlier value:  20.0072083651213
min outlier value: -73.2167184552674
% of Fraudulent transactions in outliers =  0.010234523908179333
% of Normal transactions in outliers =  0.9897654760918206

For  V9 :
number of outliers =  8283
max outlier value:  15.5949946071278
min outlier value: -13.4340663182301
% of Fraudulent transactions in outliers =  0.0269226125799831
% of Normal transactions in outliers =  0.9730773874200169

For  V10 :
number of outliers =  9496
max outlier value:  23.7451361206545
min outlier value: -24.5882624372475
% of Fraudulent transactions in outliers =  0.04201769165964617
% of Normal transactions in outliers =  0.9579823083403538

For  V11 :
number of outliers =  780
max outlier value:  12.0189131816199
min outlier value: -4.79747346479757
% of Fraudulent transactions in outliers =  0.3769230769230769
% of Normal transactions in outliers =  0.6230769230769231

For  V12 :
number of outliers =  15348
max outlier value:  7.8483920756446
min outlier value: -18.6837146333443
% of Fraudulent transactions in outliers =  0.02664842324732864
% of Normal transactions in outliers =  0.9733515767526714
```

Figure 52: Information about the outliers in each column - Part 1

```
For  V13 :
number of outliers =  3368
max outlier value:  7.12688295859376
min outlier value: -5.79188120632084
% of Fraudulent transactions in outliers =  0.0023752969121140144
% of Normal transactions in outliers =  0.997624703087886

For  V14 :
number of outliers =  14149
max outlier value:  10.5267660517847
min outlier value: -19.2143254902614
% of Fraudulent transactions in outliers =  0.030390840342073646
% of Normal transactions in outliers =  0.9696091596579264

For  V15 :
number of outliers =  2894
max outlier value:  8.87774159774277
min outlier value: -4.49894467676621
% of Fraudulent transactions in outliers =  0.00414651002073255
% of Normal transactions in outliers =  0.9958534899792675

For  V16 :
number of outliers =  8184
max outlier value:  17.3151115176278
min outlier value: -14.1298545174931
% of Fraudulent transactions in outliers =  0.04288856304985337
% of Normal transactions in outliers =  0.9571114369501467

For  V17 :
number of outliers =  7420
max outlier value:  9.25352625047285
min outlier value: -25.1627993693248
% of Fraudulent transactions in outliers =  0.053504043126684636
% of Normal transactions in outliers =  0.9464959568733153

For  V18 :
number of outliers =  7533
max outlier value:  5.04106918541184
min outlier value: -9.49874592104677
% of Fraudulent transactions in outliers =  0.0314615690959777
% of Normal transactions in outliers =  0.9685384309040223

For  V19 :
number of outliers =  10205
max outlier value:  5.59197142733558
min outlier value: -7.21352743017759
% of Fraudulent transactions in outliers =  0.013228809407153356
% of Normal transactions in outliers =  0.9867711905928467

For  V20 :
number of outliers =  27770
max outlier value:  39.4209042482199
min outlier value: -54.497720494566
% of Fraudulent transactions in outliers =  0.006985956067698956
% of Normal transactions in outliers =  0.9930140439323011

For  V21 :
number of outliers =  14497
max outlier value:  27.2028391573154
min outlier value: -34.8303821448146
% of Fraudulent transactions in outliers =  0.0168310684969304
% of Normal transactions in outliers =  0.9831689315030696

For  V22 :
number of outliers =  1317
max outlier value:  10.5030900899454
min outlier value: -10.933143697655
% of Fraudulent transactions in outliers =  0.02126044039483675
% of Normal transactions in outliers =  0.9787395596051632

For  V23 :
number of outliers =  18541
max outlier value:  22.5284116897749
min outlier value: -44.8077352037913
% of Fraudulent transactions in outliers =  0.008036243999784263
% of Normal transactions in outliers =  0.9919637560002157

For  V24 :
number of outliers =  4774
max outlier value:  4.58454913689817
min outlier value: -2.83662691870341
% of Fraudulent transactions in outliers =  0.0006284038542103058
% of Normal transactions in outliers =  0.9993715961457897
```

Figure 53: Information about the outliers in each column - Part 2

```
For  V25 :
number of outliers =  5367
max outlier value:  7.51958867870916
min outlier value: -10.2953970749851
% of Fraudulent transactions in outliers =  0.01024781069498789
% of Normal transactions in outliers =  0.9897521893050121

For  V26 :
number of outliers =  5596
max outlier value:  3.5173456116238
min outlier value: -2.60455055280817
% of Fraudulent transactions in outliers =  0.0017869907076483202
% of Normal transactions in outliers =  0.9982130092923517

For  V27 :
number of outliers =  39163
max outlier value:  31.6121981061363
min outlier value: -22.5656793207827
% of Fraudulent transactions in outliers =  0.008783801036692797
% of Normal transactions in outliers =  0.9912161989633071

For  V28 :
number of outliers =  30342
max outlier value:  33.8478078188831
min outlier value: -15.4300839055349
% of Fraudulent transactions in outliers =  0.008964471689407422
% of Normal transactions in outliers =  0.9910355283105926

For Amount:
number of outliers =  31904
max outlier value:  25691.16
min outlier value: 184.52
% of Fraudulent transactions in outliers =  0.002852306920762287
% of Normal transactions in outliers =  0.9971476930792377
```

Figure 54: Information about the outliers in each column - Part 3

Based on the information about the outliers in each column, the feature “V11” seems like the most important feature in determining fraudulent transactions as it has the highest percentage of fraudulent transactions among the other features.

Data Preprocessing

```
# before removing duplicated rows
print(df.shape)
df["Class"].value_counts()
```

```
(284807, 30)
0    284315
1       492
Name: Class, dtype: int64
```

```
# removal of duplicated rows
df = df.drop_duplicates()
print("Shape of df after dropping duplicated rows = ", df.shape)
df["Class"].value_counts()
```

```
Shape of df after dropping duplicated rows =  (275663, 30)
0    275190
1       473
Name: Class, dtype: int64
```

Figure 55: Inspection of shape before and after removing duplicated rows

Before and after removing the duplicated rows, the distribution over classes is printed to check whether it will make the dataset more imbalanced. After that process, it can be seen that the duplicated rows have been removed successfully based on the shape and distribution over classes before and after.

```
Dropped 5071 rows!
Dropping the outliers that belong to class 0 for the attribute V10...
Dropped 6020 rows!
Dropping the outliers that belong to class 0 for the attribute V11...
Dropped 46 rows!
Dropping the outliers that belong to class 0 for the attribute V12...
Dropped 9636 rows!
Dropping the outliers that belong to class 0 for the attribute V13...
Dropped 1267 rows!
Dropping the outliers that belong to class 0 for the attribute V14...
Dropped 6151 rows!
Dropping the outliers that belong to class 0 for the attribute V15...
Dropped 2078 rows!
Dropping the outliers that belong to class 0 for the attribute V16...
Dropped 5165 rows!
Dropping the outliers that belong to class 0 for the attribute V17...
Dropped 1745 rows!
Dropping the outliers that belong to class 0 for the attribute V18...
Dropped 4954 rows!
Dropping the outliers that belong to class 0 for the attribute V19...
Dropped 5193 rows!
Dropping the outliers that belong to class 0 for the attribute V20...
Dropped 8427 rows!
Dropping the outliers that belong to class 0 for the attribute V21...
Dropped 848 rows!
Dropping the outliers that belong to class 0 for the attribute V22...
Dropped 44 rows!
Dropping the outliers that belong to class 0 for the attribute V23...
Dropped 3799 rows!
Dropping the outliers that belong to class 0 for the attribute V24...
Dropped 2337 rows!
Dropping the outliers that belong to class 0 for the attribute V25...
Dropped 1165 rows!
Dropping the outliers that belong to class 0 for the attribute V26...
Dropped 4808 rows!
Dropping the outliers that belong to class 0 for the attribute V27...
Dropped 17006 rows!
Dropping the outliers that belong to class 0 for the attribute V28...
Dropped 7443 rows!
Dropping the outliers that belong to class 0 for the attribute Amount...
Dropped 10396 rows!
Dropping the outliers that belong to class 0 for the attribute Class...
Dropped 0 rows!
```

```
print("The shape of the dataset after dropping the outliers that belong to class 0 for each attribute = ", df.shape)
```

```
The shape of the dataset after dropping the outliers that belong to class 0 for each attribute =  (110052, 30)
```

Figure 56: Shape of the dataset after dropping the outliers that belong to class 0 for each attribute

As the outliers could represent the fraudulent transactions, hence only the outliers that belong to normal transactions are removed from the dataset. The shape of the dataset is checked after removing all the outliers belonging to normal transactions, which is (110052, 30).

```
df["Class"].value_counts()
```

```
0    109579
1       473
Name: Class, dtype: int64
```

Figure 57: Distribution over classes after dropping the outliers that belong to class 0 for each attribute

The distribution over classes after dropping the outliers that belong to normal transactions is printed, and it can be seen that the number of fraudulent transactions remains the same, but not for the normal transactions.

```
from sklearn.feature_selection import VarianceThreshold
X = df.drop(columns=["Class"])
y = df["Class"]
selector = VarianceThreshold(threshold=0.1)
X_variance_thresholded = selector.fit_transform(X)
print("Shape of feature matrix before variance thresholding = ", X.shape)
print("Shape of feature matrix after variance thresholding = ", X_variance_thresholded.shape)
X_variance_thresholded
```

```
Shape of feature matrix before variance thresholding =  (110052, 29)
Shape of feature matrix after variance thresholding =  (110052, 24)
array([[ 1.19185711,  0.26615071,  0.16648011, ...,  0.1671704 ,
         0.12589453,  2.69      ],
       [-0.42596588,  0.96052304,  1.14110934, ..., -0.23279382,
         0.10591478,  3.67      ],
       [ 1.22965763,  0.14100351,  0.04537077, ...,  0.75013694,
        -0.25723685,  4.99      ],
       ...,
       [ 2.03955977, -0.1752331 , -1.19682531, ..., -0.3156099 ,
         0.20111437,  2.68      ],
       [ 0.12031638,  0.93100513, -0.54601214, ..., -0.43587009,
         0.12407888,  2.69      ],
       [-0.73278867, -0.05508049,  2.03502975, ..., -0.60662399,
        -0.39525507, 24.79      ]])
```

Figure 58: Check how many columns have been dropped after variance thresholding

The shape of the feature matrix is printed before and after performing the variance thresholding. Based on the results, 5 features have been dropped because they can't meet the threshold 0.1.

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)  # for visualization purposes, in terms of 2D scatterplot
X_pca = pca.fit_transform(X=X_variance_thresholded)

# Store as dataframe and print
X_pca = pd.DataFrame(X_pca)
print(X_pca.shape)  #> ((110052, 24))
X_pca.round(2).head()
```

```
(110052, 2)
```

	0	1
0	-28.47	-0.09
1	-27.50	1.50
2	-26.17	-0.39
3	-27.49	1.40
4	90.36	-0.69

Figure 59: Checking shape after applying Principal Component Analysis (PCA)

The PCA is performed on the feature matrix after variance thresholding to reduce the dimension of the dataset. After printing the shape, its dimension is reduced from (110052, 24) to (110052, 2).

```
X_pca.var()
```

```
0    1529.153246
1       2.667573
dtype: float64
```

```
print(pca.explained_variance_ratio_.round(4)[:10])
```

```
[0.9901 0.0017]
```

Figure 60: Getting variance and contribution of each principal component to the total variance in terms of percentage

The variances of each new feature are printed out, and the variances are greatly increased, which means there will be a clearer distinction between normal and fraudulent transactions. The contributions of each principal component to the total variance are printed in terms of percentage, which are 99.01% and 0.17% respectively.

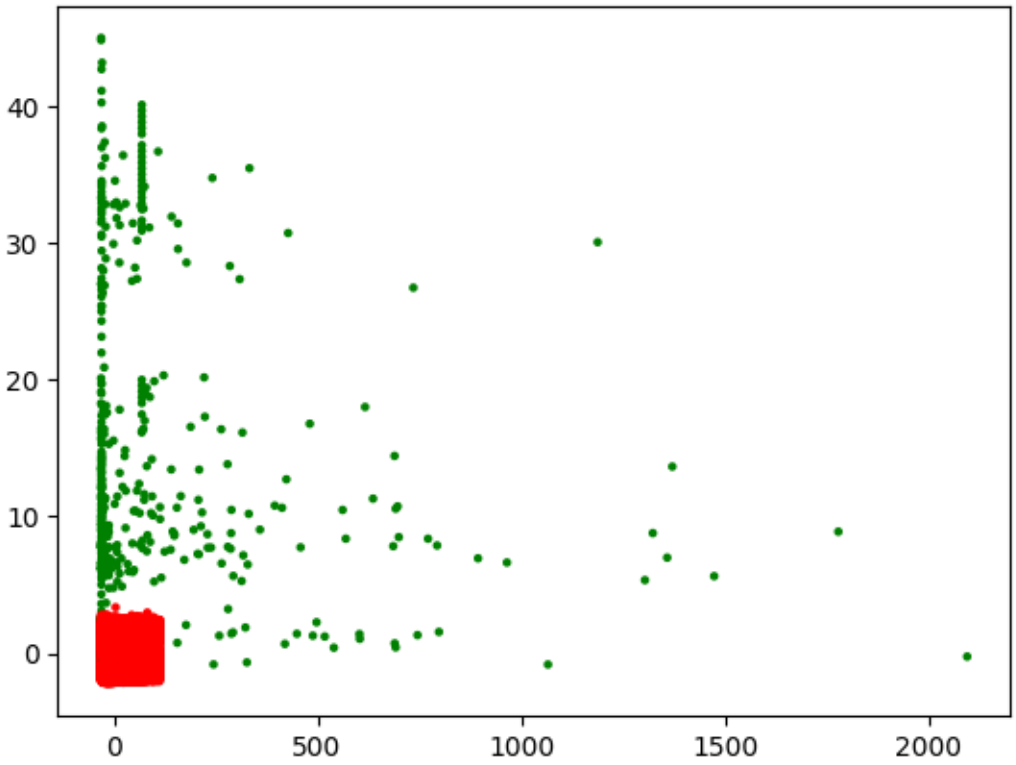
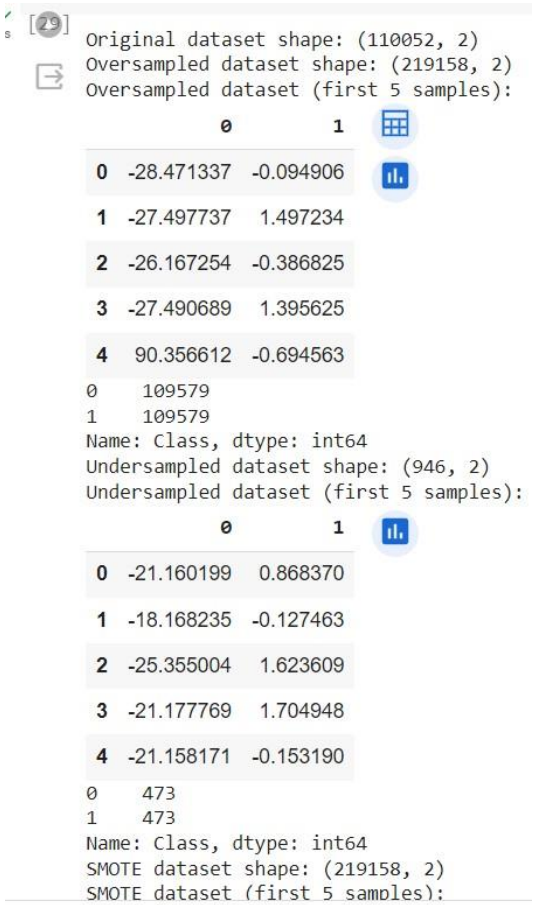


Figure 61: Scatterplot of the dataset using the 2 principal components

After getting the scatterplot of the dataset using the 2 principal components, there is a clear distinction between the normal transactions (points in green) and the fraudulent transactions (points in red). Hence it is easier for the Random Forest model to classify between different types of transactions.

Handling Imbalanced Dataset



```
[29]
Original dataset shape: (110052, 2)
Oversampled dataset shape: (219158, 2)
Oversampled dataset (first 5 samples):
```

	0	1
0	-28.471337	-0.094906
1	-27.497737	1.497234
2	-26.167254	-0.386825
3	-27.490689	1.395625
4	90.356612	-0.694563

```
0    109579
1    109579
Name: Class, dtype: int64
Undersampled dataset shape: (946, 2)
Undersampled dataset (first 5 samples):
```

	0	1
0	-21.160199	0.868370
1	-18.168235	-0.127463
2	-25.355004	1.623609
3	-21.177769	1.704948
4	-21.158171	-0.153190

```
0    473
1    473
Name: Class, dtype: int64
SMOTE dataset shape: (219158, 2)
SMOTE dataset (first 5 samples):
```

Figure 62

```
Name: Class, dtype: int64
SMOTE dataset shape: (219158, 2)
SMOTE dataset (first 5 samples):
```

	0	1
0	-28.471337	-0.094906
1	-27.497737	1.497234
2	-26.167254	-0.386825
3	-27.490689	1.395625
4	90.356612	-0.694563

```
0    109579
1    109579
Name: Class, dtype: int64
```

Figure 63

The results demonstrate that the resampling strategies suggested before were effective. The 110,052 samples across two features in the PCA-preprocessed data are shown by the original dataset form. In order to make the minority class as large as the majority class, Random Oversampling is used. The outcome is a dataset of 219,158 samples, which is larger than before. This proves that the dataset size has been doubled due to resampling, since it implies that each minority class sample was roughly repeated in order to attain balance.

Alternatively, Random Undersampling substantially decreases the dataset to 946 samples. A large amount of the data from the majority class was likely eliminated due to this drastic decrease. This equals the size of the minority group, which is relatively smaller. Despite the balanced dataset that this technique delivers, there seems to be a significant loss of data points that might be crucial for the model's success.

By replicating the size of the oversampled dataset, the resulting dataset, after applying SMOTE, once again displays a total of 219,158 samples. In contrast to Random Oversampling, SMOTE

creates brand new synthetic samples for the underrepresented class instead of just copying and pasting them. As a result of each resampling procedure, the feature space changes in the first five dataset samples. The modification to the structure of the dataset is therefore validated.

By confirming that the dataset is now balanced, we can say that the class counts of the resampled dataset are equal. In order to build a trustworthy prediction model, these findings are vital since they guarantee that the trained model will be fair and perform well on both classes. The requirements of the Part 1 assignment have been met.

Model Training and Evaluation

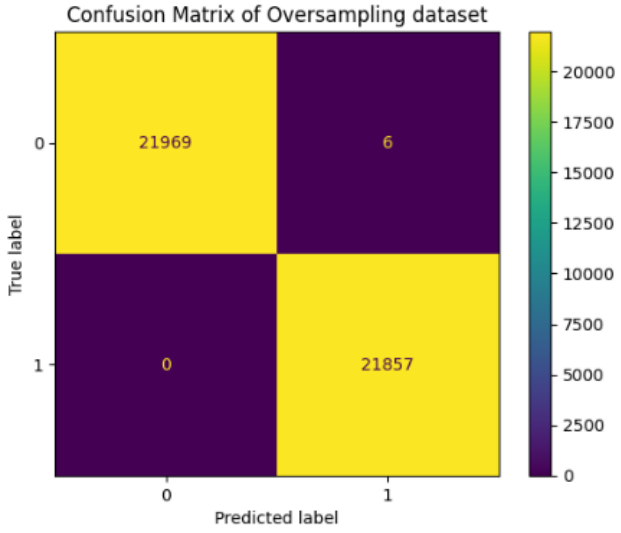


Figure 64

The figure above shows the result of the confusion matrix for the classification made by the Random Forest classifier trained on the oversampling dataset. In this confusion matrix, 0 is considered the negative class and 1 is considered the positive class. The top left corner of the confusion matrix shows the True Negative (TN) where the negative class or 0 label is being classified correctly, the top right corner shows the False Negative (FN) where the classification that belongs to positive class is wrongly classified to be negative class, the bottom left of the matrix shows the False Positive (FP) where the actual label is 0 or negative class but is classified as positive or 1, and lastly the bottom right is the True Positive (TP) where the actual label is 1 or positive and the classification is correct. The result shows the 21969 True Negative predictions were made, 6 False Negative predictions were made, 0 False Positive predictions were made and a total of 21857 True Positive predictions were made on the oversampling test dataset. In more detail, the accuracy of the model can be calculated using the formula shown below:

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

Figure 65

The calculation using this formula shows that the model is able to achieve an astonishing 0.9999 accuracy on the oversampling dataset.

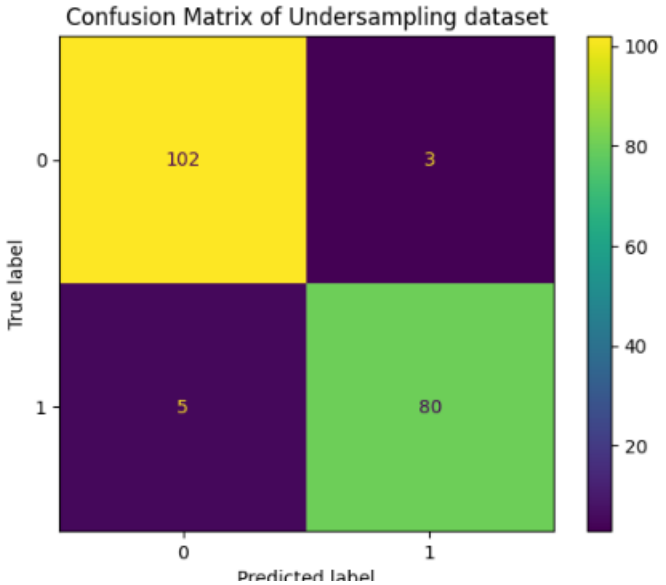


Figure 66

The figure above shows another confusion matrix for the result of classification made by model trained on the undersampling train dataset. The result showed that 102 classifications were True Negative, 3 were False Negative, 5 were False Positive and 80 were True Positive. The result obtained was undoubtedly not as good as the result above, but the accuracy of the model is still very high obtaining an accuracy of 0.9579.

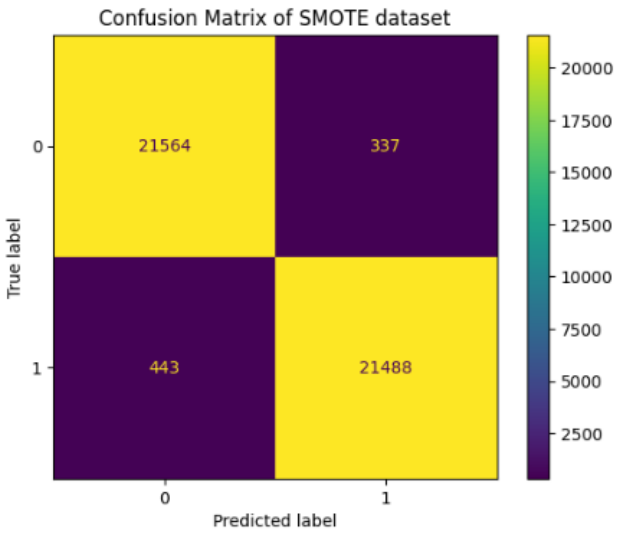


Figure 67

The figure above shows another confusion matrix for the result of classification made by the model trained on the undersampling train dataset. The result showed that 21564 classifications were True Negative, 337 were False Negative, 443 were False Positive and 21488 were True Positive. The result obtained was undoubtedly not as good as the result above, but the accuracy of the model is still very high obtaining an accuracy of 0.9579.

```
from sklearn.metrics import make_scorer, precision_score
param_dist = {'n_estimators': [50, 100, 150, 250, 300],
              'max_depth':[4, 8, 12, 16 ,20]}

# Create a random forest classifier
rf = RandomForestClassifier()


# Define the scoring metric
scoring_metric = make_scorer(precision_score)

# Use random search to find the best hyperparameters with precision as the scoring metric
rand_search = RandomizedSearchCV(rf,
                                 param_distributions=param_dist,
                                 n_iter=5,
                                 cv=3,
                                 scoring=scoring_metric)

# Fit the random search object to the data
rand_search.fit(X_ros_train, y_ros_train)
```

```
RandomizedSearchCV
estimator: RandomForestClassifier
RandomForestClassifier
```

```
# Create a variable for the best model
best_rf_ros = rand_search.best_estimator_

# Print the best hyperparameters
print('Best hyperparameters:',  rand_search.best_params_)
```

```
Best hyperparameters: {'n_estimators': 250, 'max_depth': 20}
```

Figure 68

The figure above shows the best combination of hyperparameters obtained by using a random search algorithm performed on the oversampling dataset. The best set of hyperparameters obtained were n_estimators with a value of 250 and max_depth with a value of 20.

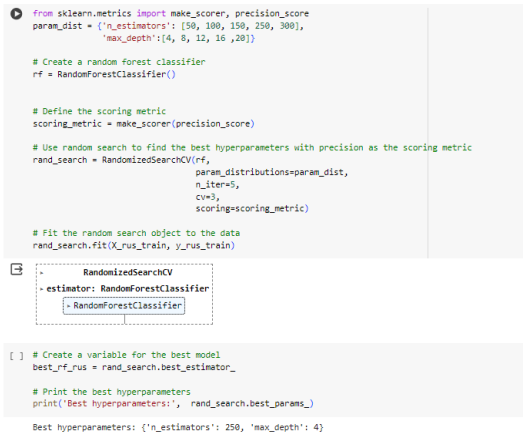


Figure 69

The figure above shows the best combination of hyperparameters obtained by using a random search algorithm performed on the undersampling dataset. The best set of hyperparameters obtained were n_estimators with a value of 250 and max_depth with a value of 4.

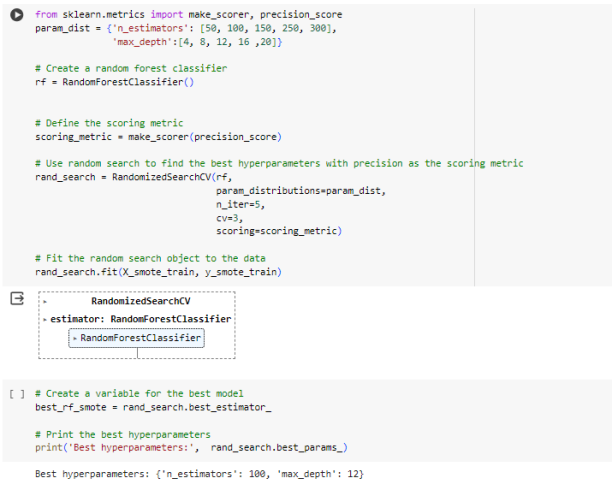


Figure 70

The figure above shows the best combination of hyperparameters obtained by using a random search algorithm performed on the SMOTE dataset. The best set of hyperparameters obtained were n_estimators with a value of 100 and max_depth with a value of 12.

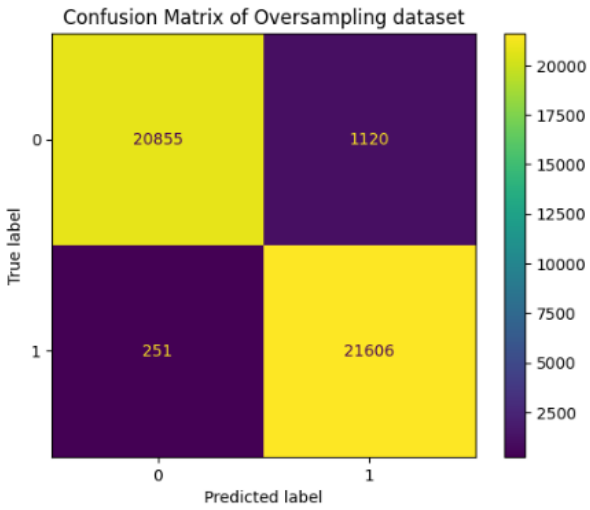


Figure 71

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the oversampling dataset classifier. From the result, it can be seen that 20855 were True Negative, 1120 were False Negative, 251 were False Positive and 21606 were True Positive. In terms of accuracy, the model achieved an accuracy of 0.9687 compared to the accuracy of 0.9999 obtained without hyperparameter tuning for the oversampling dataset.

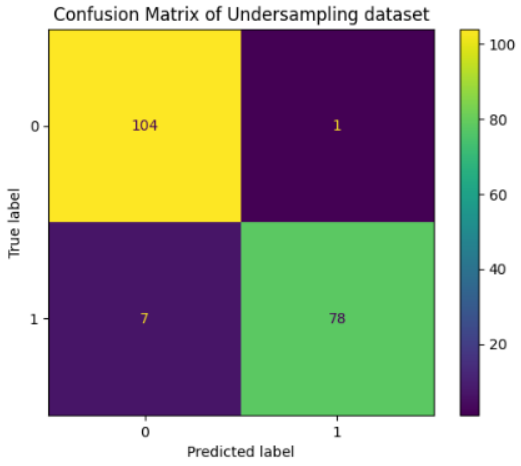


Figure 72

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the undersampling dataset classifier. From the result, it can be seen that 104 were True Negative, 1 were False Negative, 7 were False Positive and 78 were True Positive. In terms of accuracy, the model achieved an accuracy of 0.9589.

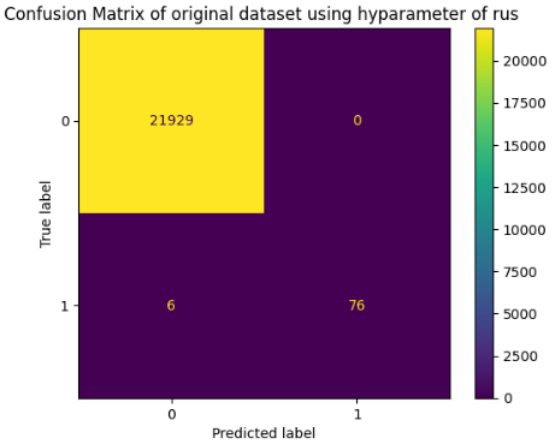


Figure 73

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the SMOTE dataset classifier. From the result, it can be seen that 21617 were True Negative, 284 were False Negative, 963 were False Positive and 20968 were True Positive. In terms of accuracy, the model achieved an accuracy of 0.9716.

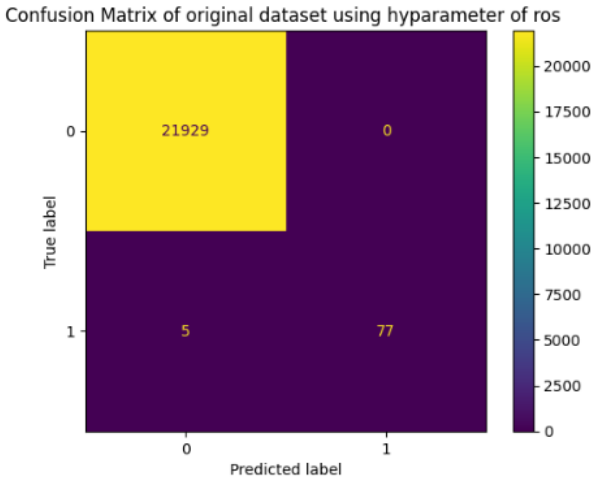


Figure 74

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the oversampling dataset on the test set of the original dataset, X_test. From the result, it can be seen that 21929 were True Negative, 0 were False Negative, 5 were False Positive and 77 were True Positive. In terms of accuracy, the model achieved an accuracy of 0.9998.

Figure 75

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the undersampling dataset on the test set of the original dataset, X_test. From the result, it can be seen that 21929 were True Negative, 0 were False Negative, 6 were False Positive and 76 were True Positive. In terms of accuracy, the model achieved an accuracy of 0.9997.

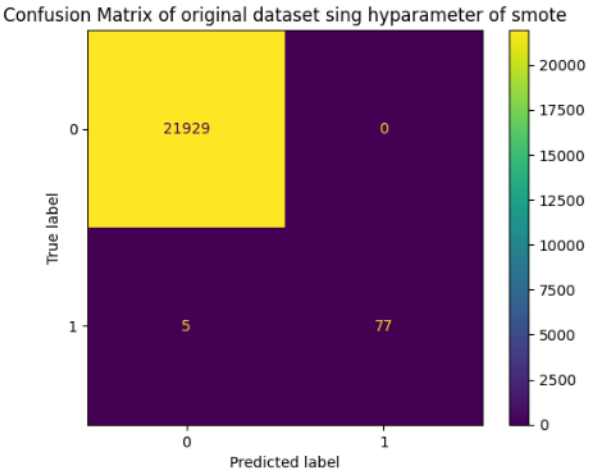


Figure 76

The above shows the result of the confusion matrix obtained for the classification result by the model trained using hyperparameter settings obtained from the hyperparameter tuning process for the SMOTE dataset on the test set of the original dataset, X_test. From the result, it can be seen that 21929 were True Negative, 0 were False Negative, 5 were False Postive and 77 were True Postive. In terms of accuracy, the model achieved an accuracy of 0.9998.

3.3 Summary

In summary, three methods are deployed for handling the imbalanced dataset which are random oversampling, random undersampling, and random SMOTE. In the first part, random forest classifiers are trained and evaluated based on the three datasets obtained. The second part of model training performs hyperparameter tuning for all three classifiers on the respective datasets using a random search algorithm. From there, the best hyperparameters for all three types of datasets is obtained. The next step taken by the researchers is to create three classifiers using the hyperparameters obtained from the three hyperparameter tuning stages to train a classifier on the original dataset. The result evaluated from the confusion matrix showed that all three types of hyperparameter combinations obtained from random oversampling, undersampling and SMOTE are quite similar and considered satisfactory.

Through observation of the result obtained, random oversampling paired with the hyperparameter of n_estimators with a value of 250 and max_depth with a value of 20 gives the best result overall before and after hyperparameter tuning. The classifier is able to achieve 21929 True Negative, 0 False Negative, 5 False Positive, and 77 True Positive when tested on the original dataset. Despite this, in the case of fraud detection, researchers should aim to minimize the number of false negatives as it might be fatal if even a single fraudulent transaction is missed out. Adding to that, customers will be very mad if they are warned that they are involved in fraud as they might need to go through many troublesome procedures and even cancellation of credit cards. Not only that, but it is also mandatory to minimize the false positives found in the classification of transactions because the detection of normal transactions as fraudulent requires the waste of company resources to inform customers and doing research on the root course of the incident. However, in the case of the models trained, there are no false negatives that are classified. Hence, there are two models performing equally well: the first model is with ***n_estimators=250*** and ***max_depth=20***, while the second model is with ***n_estimators=100*** and ***max_depth=12***.

According to Occam's Razor, if there are two machine learning models performing equally well, then a simpler model is preferred over a complex model (Mavuduru, 2022). Hence the model

with ***n_estimators=100*** and ***max_depth=12*** is selected by our team due to lesser number of trees and smaller maximum depth of each tree.

4. Conclusion

4.1 Conclusion

Credit card fraud detection was the focus of this research, which used cutting-edge machine learning and further AI capabilities. The goal is to create a model that can anticipate fraudulent transactions with a high degree of accuracy. Issues with real-time fraud detection, the increasing complexity of fraud schemes, and unbalanced transaction datasets were the most common throughout the research.

Important machine learning techniques for resolving dataset imbalance included oversampling, undersampling, and the Synthetic Minority Over-sampling Technique (SMOTE). The model's accuracy and its capacity to adapt to new fraudulent patterns were both improved by this. The groundwork for comprehending dataset subtleties was provided by the exploratory data analysis (EDA). By honing the model's inputs, this has guided the data preparation and feature engineering stages.

To further solidify the model's effectiveness, the Random Forest method was used in conjunction with thorough hyperparameter adjustment and validation. Thus, this accomplishes noteworthy standards in recall, precision, and accuracy. By providing a scalable and adaptable solution that can stay up with the ever-changing strategies used by criminals, this result demonstrated the feasibility of machine learning in addressing complicated situations such as credit card fraud.

Still, developing better detection algorithms is essential in the never-ending war against credit card fraud. Investigating the feasibility of using deep learning to build real-time detection systems

that can keep up with the speed of present transactions and reveal complex fraud patterns is an area that might need further investigation in the future.

Ultimately, this study has shown that machine learning is a powerful tool against credit card fraud and has shown how to improve the safety of online financial transactions. Digital financial ecosystems may be greatly enhanced using the approaches and insights obtained from this study. This leads to greater confidence and safety for financial institutions and cardholders alike.

4.2 Limitations

Here is the list of limitations of this assignment, and all of them are because of the nature of the dataset:

- Imbalanced dataset: only 0.172% of the transactions are fraudulent, hence it's difficult to build a model to correctly classify the minority class due to lack of data. Other than that, the trained model will be biased to the majority class due to the relatively more data.
- Confidentiality issues: the original features and additional background information are hidden due to confidentiality concerns. This can restrict the depth of Exploratory Data Analysis (EDA) and feature engineering on the dataset, leading to a bad-performing machine learning model.
- Temporal information: The feature "Time" in the dataset represents the seconds elapsed between each transaction and the first transaction. The feature does not provide additional context about seasonality hence the machine learning model can't predict the transactions based on the time when they happen.

4.3 Future Works

Here is the list of future works that can be done to improve the accuracy of detecting fraudulent transactions:

- Advanced modelling approaches: Deep-learning approaches might be considered because this dataset can be considered as huge as it consists of 284807 transactions. Deep-learning approaches are typically more suitable than traditional machine learning approaches when it comes to handling huge datasets.
- More feature engineering approaches: Experimenting with different combinations of feature engineering methods such as scaling methods or transformations to see which combination produces the best results.

References

Dorsainvil, K. (2019, November 14). *Dealing with Multicollinearity of Features*. Retrieved from medium: https://medium.com/@kdorsain/dealing-with-multicollinearity-of-features-2b13bc42fbe1

Jayawardena, N. (2020). *How to Deal with Imbalanced Data*. Retrieved from Medium: https://towardsdatascience.com/how-to-deal-with-imbalanced-data-34ab7db9b100

kingJulian. (2019, February 7). *A feature distribution is nearly normal: what does that imply for my ML model?* Retrieved from stackoverflow: https://stackoverflow.com/questions/54071893/a-feature-distribution-is-nearly-normal-what-does-that-imply-for-my-ml-model#:~:text=So%2C%20coming%20back%20to%20your,)%20models%2C%20LDA%20%2C%20QDA%20

Mavuduru, A. (2022, August 10). *What Occam's Razor Means in Machine Learning*. Retrieved from towards data science: https://towardsdatascience.com/what-occams-razor-means-in-machine-learning-53f07effc97c#:~:text=Occam%27s%20Razor%20is%20a%20philosophical,over%20a%20more%20complex%20model.

Team, D. A. (2020, December 31). *Outlier Detection and Its importance in Machine learning*. Retrieved from datamites: https://datamites.com/blog/outlier-detection-and-its-importance-in-machine-learning/

Turney, S. (2022, May 10). *Skewness | Definition, Examples & Formula*. Retrieved from Scribbr: https://www.scribbr.com/statistics/skewness/#:~:text=The%20three%20types%20of%20skewness,peak%20than%20on%20its%20right.

W3schools. (n.d.). *What is NumPy?* Retrieved from W3schools: https://w3schools.com/python/numpy/numpy_intro.asp