ILLINOIS INSTITUTE OF TECHNOLOGY

CSP 571 – Data Preparation and Analysis

Final Project Report

Enhancing Predictive Analytics with Random Forest: A Comprehensive Machine Learning Pipeline for Class Prediction

Team Members:

Sai Teja Reddy Kotha - A20547859

Tharun Vikas Belagala - A20531643

Gowthami Kamatam - A20555738

Sai Pranay Yada - A20553636

Under the Guidance Of

Prof. Jawahar J. Panchal

Table of Contents

List of Figures

Abstract

The project's objective is to develop a robust machine learning pipeline that can predict a categorical target variable from a given dataset containing more than 1.2 million records across 15 numerical features. The pipeline will integrate comprehensive preprocessing, including handling missing values, scaling, encoding, and outlier detection, to ensure high-quality input data. EDA will be used to uncover data patterns, visualize feature relationships, and guide feature selection.

A Random Forest Classifier is used for training the model, which was chosen because it works quite well on high-dimensional data. Hyperparameter tuning improves the performance of the model, and the evaluation metrics used are accuracy, precision, recall, and F1-score, which give a full overview, especially on imbalanced datasets. After training, the model is exported to the ONNX (Open Neural Network Exchange) format. ONNX is a cross-platform standard that enables easy integration of machine learning models across different environments, including edge devices and cloud-based platforms. The ONNX export ensures that the model can be seamlessly deployed for real-time predictions in both production environments and edge devices, providing a scalable solution for various applications.

Future work will involve class imbalance challenges using techniques such as oversampling and cost-sensitive learning. Other algorithms like Gradient Boosting and XGBoost will be explored in pursuit of better performance. Model interpretability techniques using SHAP and LIME will be integrated to explain the feature contributions, enhancing transparency. Monitoring systems will be set up after deployment to track performance and detect model drift, with subsequent automated retraining to adapt to shifting data distributions, ensuring long-term scalability and relevance.

This project underlines the capability of Random Forest classifiers in the presence of high-dimensional predictive, putting scalable deployment, interpretability, and continuous performance monitoring into the modern real-world application perspective.

1. Overview:

1.1 Problem Statement:

Predicting categorical outcomes based on complex numerical data is a common challenge across domains such as healthcare, finance, and marketing. The classification problem solved in this project involves the development of a model for assigning categorical labels (Class) to observations characterized by multiple numerical features. This predictive capability has wide-ranging applications, from customer segmentation and fraud detection to risk assessment and resource optimization. However, it requires handling the challenge of achieving high accuracy without overfitting, good generalization ability on unseen data, and considering computational challenges for such a huge dataset.

Success with predictive models of this kind opens doors to automation of decision-making processes, increased value through decision insights, and a considerable reduction in expenses. This work puts in as much consideration the attainment of high predictive performance as it ensures the scalability, interpretability, and adaptability of the methodology to various applications.

1.2 Related Works

Classification tasks, therefore, have widely been studied using machine learning; ensemble methods are being given considerable attention due to their effectiveness and versatility, of which Random Forests are part. Bierman's pioneering work on Random Forests introduced the strengths of this method, which included its capability of handling huge datasets with high generalization and without overfitting via bagging. Recent studies further point out their utility in high-dimensional feature space where interactions among variables are nonlinear and complex.

In parallel, scaling, normalization, and imputation of data have been identified as steps that are crucial to guarantee the quality of the input into the machine learning models. Géron (2019) underlines that EDA helps to detect hidden patterns and informs about feature selection. Advanced metrics include F1-score and AUC-ROC, both of which are recommended for the assessment of model performance in an imbalanced dataset.

This project takes inspiration from such studies, integrating best practices into a structured pipeline for developing a reliable classification model. The literature suggests that though powerful, comparing the performance of Random Forests with alternative algorithms like Gradient Boosting or Neural Networks provides further insight, which is proposed for future work.

1.3 Proposed Methodology

Data Preprocessing:

The raw dataset is cleaned for inconsistencies, missing values, and anomalies. Since the data provided is complete and does not contain any missing values, the focus will be on the scaling of numerical features to enable the model to learn effectively. Features are scaled or normalized using Min-Max scaling or standardization to bring them to comparable scales.

Exploratory Data Analysis:

Visualization of the feature distributions with histograms, box plots, and scatter plots was done during the EDA to highlight outliers, skewness, and relationships between variables. The correlation matrix plotted helps understand the interrelations among different variables, indicating which to select or generate. A check on class distribution within the target variable also is considered to identify cases of imbalance. In any case, identified imbalance can easily be tackled with oversampling methods such as SMOTE and under-sampling.

Feature Engineering and Selection:

Feature selection will be done based on EDA findings; the features which contribute much toward the target prediction will be selected. Initial models or feature importance scores from Random Forest can be used to select features. Dimensionality reduction techniques, such as PCA, are considered if the computational efficiency is an issue. Domain knowledge could be another way of creating composite features or removing redundant ones.

Model Training:

A Random Forest classifier is selected because it can support high-dimensional data and present interpretable results. The dataset is then divided into a training set, usually 80%, and a test set, usually 20%, to ensure that the model sees enough diversity in training but is tested on observations it has never seen. This includes the tuning of hyperparameters such as the number of trees, maximum depth, and minimum samples per leaf by methods like grid search or randomized search.

Model Validation:

The validation procedure is meant to check the generalization ability of the model. Overall performance can be viewed using accuracy, while precision, recall, and F1-score provide more fine-grained analyses, especially on imbalanced datasets. A confusion matrix was analyzed to spot the exact areas of misclassifications with a view to refining the model further. Cross-validation was performed to ensure the robustness of the results across different splits.

Deployment Plan

The deployment strategy is based on exporting the trained Random Forest model in ONNX format, ensuring portability and interoperability across different systems and programming environments. The ONNX format was selected because it allows easy deployment of models with a high degree of efficiency during inference.

Model Export:

After having the trained model tested and its performance verified, it was converted into an ONNX file. This is a lightweight file that contains the model architecture and parameters and is easily deployable across platforms.

Inference Capabilities:

The model is loaded through the ONNX runtime environment in order to perform real-time predictions. This runtime assures optimized use of resources for good performance of the model during demanding cases, like the decision-making systems.

Systems Integration:

The ONNX model can be integrated into various applications, such as RESTful APIs or standalone microservices that allow client systems to interact with the model via standardized interfaces. This approach supports scalability and enables multiple users or devices to access the predictions concurrently.

Platform Versatility:

The ONNX format assures compatibility with a wide variety of hardware and software environments. It can be utilized to deploy models in Python-based servers, integrate them with C++ applications, or even execute the models on edge devices where low-latency predictions are essential. This makes ONNX very ideal for diverse deployment needs.

Performance Monitoring:

A monitoring framework is to be implemented to track the deployed model's predictions over time and ensure its continued performance. Metrics will be around prediction accuracy and latency while systems also check for data drift. When issues have been identified, the ONNX format has eased the update of models; with new versions, seamless deployment is possible without significant reconfiguration.

This deployment approach ensures that the model is not only effective but also adaptable to the constantly evolving requirements and environments.

2. Data Processing

2.1 Pipeline Details

The data preprocessing pipeline was developed to prepare the dataset for the best performance of machine learning models. This pipeline is presented herein; each step of this stage was chosen with respect to model training efficiency and in model results, taking care that they are interpretable and robust. Below is a more detailed look at the steps involved:

2.2 Data Loading and Preliminary Examination:

The first step in any data processing is to load the dataset and do an initial inspection. This will help in understanding the structure of the data and in identifying any potential problems that could influence the analysis. The dataset was read into the environment using Pandas and basic operations were performed to inspect the dimensions of the dataset and its first rows.

This allowed us to identify any obvious issues, such as type errors in the data, missing columns, or formatting errors, quickly.

Code:

```
file_path = 'data_public.csv'
data = pd.read_csv(file_path)

data_info = {
    "head": data.head(),
    "info": data.info(),
    "description": data.describe(),
    "class_distribution": data['Class'].value_counts() if 'Class' in data.columns else "Target column missing."
}
data_info
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1200000 entries, 0 to 1199999
Data columns (total 16 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   A       1200000 non-null  float64
 1   B       1200000 non-null  float64
 2   C       1200000 non-null  float64
 3   D       1200000 non-null  float64
 4   E       1200000 non-null  float64
 5   F       1200000 non-null  float64
 6   G       1200000 non-null  float64
 7   H       1200000 non-null  float64
 8   I       1200000 non-null  float64
 9   J       1200000 non-null  float64
 10  K       1200000 non-null  float64
 11  L       1200000 non-null  float64
 12  M       1200000 non-null  float64
 13  N       1200000 non-null  float64
 14  O       1200000 non-null  float64
 15  Class   1200000 non-null  int64
dtypes: float64(15), int64(1)
memory usage: 146.5 MB
{'head':            A          B           C          D           E          F  \
 0  231.420023 -12.210984  217.624839 -15.611916  140.047185  76.904999
 1  -38.019270 -14.195695    9.583547  22.293822  -25.578283 -18.373955
 2  -39.197085 -20.418850   21.023083  19.790280  -25.902587 -19.189004
 3  221.630408  -5.785352  216.725322  -9.900781  126.795177  85.122288
 4  228.558412 -12.447710  204.637218 -13.277704  138.930529  91.101870
```

```
           G          H          I          J          K        L  \
0  131.591871  198.160805  82.873279  127.350084  224.592926 -5.992983
1   -0.094457  -33.711852  -8.356041   23.792402    4.199023  2.809159
2   -2.953836  -25.299219  -6.612401   26.285392    5.911292  6.191587
3  108.857593  197.640135  82.560019  157.105143  212.989231 -3.621070
4  115.598954  209.300011  89.961688  130.299732  201.795100 -1.573922

           M          N          O  Class
0 -14.689648  143.072058  153.439659      3
1 -59.330681  -11.685950    1.317104      2
2 -56.924996   -4.675187   -1.027830      2
3 -15.469156  135.265859  149.212489      3
4 -15.128603  148.368622  147.492663      3  ,
'info': None,
'description':                  A             B             C             D             E  \
count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
mean   5.068656e+01 -1.883373e+01  7.162152e+01 -1.355120e+01  2.944177e+01
std    1.292492e+02  1.446355e+01  1.052808e+02  4.689774e+01  7.282278e+01
min   -7.308940e+01 -8.322357e+01 -5.972853e+01 -1.375818e+02 -3.829826e+01
25%   -3.793679e+01 -1.786669e+01  7.553164e+00 -1.471337e+01 -2.436286e+01
50%   -3.197847e+01 -1.369876e+01  1.348796e+01 -8.004308e+00 -1.897058e+01
75%    2.280020e+02 -1.055606e+01  2.123439e+02  1.955806e+01  1.289018e+02
max    2.687738e+02  4.460108e+00  2.561698e+02  3.263799e+01  1.579843e+02

                 F             G             H             I             J  \
count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
mean  -6.185189e+00  3.174186e+01  5.112504e+01  3.300077e+01  4.092546e+01
std    7.309100e+01  6.660329e+01  1.034053e+02  4.217119e+01  7.694386e+01
min   -1.485917e+02 -6.654137e+01 -4.246089e+01 -1.818542e+01 -1.123844e+02
25%   -3.072492e+01 -3.484185e+00 -2.629661e+01 -7.594991e+00  2.108044e+01
50%   -2.475391e+01  1.491431e+00 -1.817028e+01  3.769369e+01  2.717432e+01
75%    7.834417e+01  1.151840e+02  1.915891e+02  7.984842e+01  1.253846e+02
max    1.229186e+02  1.660534e+02  2.329496e+02  1.112970e+02  1.755397e+02

                 K             L             M             N             O  \
count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
mean   7.938340e+01 -6.746540e+00 -4.232290e+01  4.949012e+01  5.980333e+01
std    9.484003e+01  1.557490e+01  1.791142e+01  6.728231e+01  6.677712e+01
min   -1.415233e+01 -6.271828e+01 -8.144988e+01 -2.057979e+01 -1.283059e+01
25%    2.419273e+00 -8.875128e+00 -5.567326e+01 -7.131906e+00  1.628438e-01
50%    2.652955e+01 -1.079123e+00 -5.297585e+01  1.462293e+01  4.689262e+01
75%    2.046458e+02  3.334451e+00 -2.208504e+01  1.363603e+02  1.451293e+02
max    2.598003e+02  2.159496e+01  1.032828e+01  1.789303e+02  1.807011e+02

              Class
count  1.200000e+06
mean   2.324106e+00
std    7.211461e-01
min    1.000000e+00
25%    2.000000e+00
50%    2.000000e+00
75%    3.000000e+00
max    3.000000e+00  ,
'class_distribution': Class
3    569521
2    449885
1    180594
Name: count, dtype: int64}
```

- The above output suggest that the data frame contains 15 numerical features from A to O, having than 1.2 million entries and a Target column: “Class” which has 3 categories: 1,2,3.
- It also says that all the columns have non-null values. And the class distribution shows a imbalance that will be handled later.

2.3 Handling Missing Values and Duplicates:

Other than data normalization, checking for missing values is usually among the first tasks performed as part of any preprocessing step. Although our dataset did not have nulls in it, including the logic that would handle such values in the future was an important action. Application of the isnull() function checked for missing data across all columns; though no missing values were found, this logic imputed missing values to make the model robust.

Code:

```
#Test for missing values
print("Missing values:\n", data.isnull().sum())
```

output:

```
Missing values:

 A       0
B        0
C        0
D        0
E        0
F        0
G        0
H        0
I        0
J        0
K        0
L        0
M        0
N        0
O        0
Class    0
dtype: int64
```

We considered different imputation methods, like mean or median imputation for numerical data, or mode imputation for categorical data.

```
## Handle missing values
num_imputer = SimpleImputer(strategy='mean')  # For numerical columns
num_cols = data.select_dtypes(include=['float64', 'int64']).columns

data[num_cols] = num_imputer.fit_transform(data[num_cols])
```

Duplicate rows in a dataset might skew the analysis, especially if they are not removed. No duplicates were found in the dataset, but for the sake of ensuring that in future data, duplicates are taken care of, this step was included. This ensured that the dataset was kept unique and clean using the function drop_duplicates().

```
## Remove duplicates
data = data.drop_duplicates()
```

2.4 Outlier Removal

Outlier Detection and Handling with Z-Scores

Outliers, which are data points significantly different from the rest of the data, can distort the learning process and reduce model accuracy. Identifying and handling outliers is essential for robust machine learning models. In this project, the outliers were identified by using Z-Scores, one of the commonly used methods for identifying extreme values. The Z-score indicates how many standard deviations a data point is away from the mean. A Z-score greater than a threshold identifies a data point as an outlier. In this case, the threshold value was set to be 3, which meant that any data point having a Z-score greater than 3 was considered an outlier.

How to Handle Outliers

The following is the procedure followed for handling outliers:

Calculating Z-Score:

The Z-scores for each feature were calculated using Scikit-learn's zscore function from the scipy.stats module. This function standardizes each feature by subtracting the mean and dividing by the standard deviation. In this dataset, for every column, excluding the target variable Class, we have computed the Z-scores.

Outlier Removal:

Those data points were then considered outliers for which the absolute Z-score was greater than our threshold of 3. We filter the dataset to keep only those rows where all features had a Z-score less than our threshold. This will prevent extreme values from warping the model training process.

```
z_scores = np.abs(zscore(data[num_cols.drop('Class', errors='ignore')]))
outlier_threshold = 3
data = data[(z_scores < outlier_threshold).all(axis=1)]
```

- **output: Data frame after outliner**

```
<class 'pandas.core.frame.DataFrame'>
Index: 1193716 entries, 0 to 1199999
Data columns (total 16 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   A       1193716 non-null  float64
 1   B       1193716 non-null  float64
 2   C       1193716 non-null  float64
 3   D       1193716 non-null  float64
 4   E       1193716 non-null  float64
 5   F       1193716 non-null  float64
 6   G       1193716 non-null  float64
 7   H       1193716 non-null  float64
 8   I       1193716 non-null  float64
 9   J       1193716 non-null  float64
 10  K       1193716 non-null  float64
 11  L       1193716 non-null  float64
 12  M       1193716 non-null  float64
 13  N       1193716 non-null  float64
 14  O       1193716 non-null  float64
 15  Class   1193716 non-null  float64
dtypes: float64(16)
memory usage: 154.8 MB
{'head':             A          B          C          D          E          F  \
 0  231.420023 -12.210984  217.624839 -15.611916  140.047185  76.904999
 1  -38.019270 -14.195695    9.583547  22.293822  -25.578283 -18.373955
 2  -39.197085 -20.418850   21.023083  19.790280  -25.902587 -19.189004
 3  221.630408  -5.785352  216.725322  -9.900781  126.795177  85.122288
 4  228.558412 -12.447710  204.637218 -13.277704  138.930529  91.101870

             G          H          I          J          K         L  \
 0  131.591871  198.160805  82.873279  127.350084  224.592926 -5.992983
 1   -0.094457  -33.711852  -8.356041   23.792402    4.199023  2.809159
 2   -2.953836  -25.299219  -6.612401   26.285392    5.911292  6.191587
 3  108.857593  197.640135  82.560019  157.105143  212.989231 -3.621070
 4  115.598954  209.300011  89.961688  130.299732  201.795100 -1.573922

            M          N          O  Class
 0 -14.689648  143.072058  153.439659    3.0
 1 -59.330681  -11.685950    1.317104    2.0
 2 -56.924996   -4.675187   -1.027830    2.0
 3 -15.469156  135.265859  149.212489    3.0
 4 -15.128603  148.368622  147.492663    3.0  ,
 'info': None,
 'description':                   A             B             C             D             E  \
 count  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06
 mean   5.127006e+01 -1.859638e+01  7.221468e+01 -1.302937e+01  2.967397e+01
 std    1.293377e+02  1.412193e+01  1.052384e+02  4.646286e+01  7.294319e+01
 min   -7.308940e+01 -6.222383e+01 -5.972853e+01 -1.375818e+02 -3.829826e+01
 25%   -3.780827e+01 -1.775918e+01  7.685009e+00 -1.465145e+01 -2.438129e+01
 50%   -3.191680e+01 -1.366546e+01  1.354735e+01  1.208039e+01 -1.906021e+01
 75%    2.280929e+02 -1.053829e+01  2.124468e+02  1.957742e+01  1.289626e+02
 max    2.687738e+02  4.460108e+00  2.561698e+02  3.263799e+01  1.579843e+02

                  F             G             H             I             J  \
 count  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06
 mean  -5.556763e+00  3.218744e+01  5.143688e+01  3.289812e+01  4.163375e+01
 std    7.276581e+01  6.649355e+01  1.035869e+02  4.225687e+01  7.652201e+01
 min   -1.485917e+02 -6.654137e+01 -4.246089e+01 -1.818542e+01 -1.123844e+02
 25%   -3.059424e+01 -3.374245e+00 -2.631824e+01 -7.609854e+00  2.121195e+01
 50%   -2.469506e+01  1.541504e+00 -1.869131e+01 -1.980466e+00  2.723772e+01
 75%    7.845013e+01  1.153048e+02  1.916882e+02  7.992832e+01  1.254983e+02
 max    1.229186e+02  1.660534e+02  2.329496e+02  1.112970e+02  1.755397e+02

                  K             L             M             N             O  \
 count  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06  1.193716e+06
```

```
mean   7.955735e+01 -6.572324e+00 -4.226676e+01  4.956960e+01  5.980252e+01
std    9.505787e+01  1.542274e+01  1.793629e+01  6.744867e+01  6.695195e+01
min   -1.415233e+01 -5.346733e+01 -8.144988e+01 -2.057979e+01 -1.283059e+01
25%    2.397390e+00 -8.684320e+00 -5.566943e+01 -7.151169e+00  1.440800e-01
50%    1.103891e+01 -1.025141e+00 -5.297566e+01  4.363521e-01  7.384988e+00
75%    2.047679e+02  3.357204e+00 -2.200822e+01  1.364647e+02  1.452034e+02
max    2.598003e+02  2.159496e+01  1.032828e+01  1.789303e+02  1.807011e+02

              Class
count  1.193716e+06
mean   2.325839e+00
std    7.189936e-01
min    1.000000e+00
25%    2.000000e+00
50%    2.000000e+00
75%    3.000000e+00
max    3.000000e+00  ,
'class_distribution': Class
3.0    566395
2.0    449885
1.0    177436
Name: count, dtype: int64}
```

- From the Above output we can say that the outliners has been removed and we can see the change in the number of entries in the data frame.

Effect of removing Outliers:

By cleaning the outliers, the dataset became more consistent and didn't contain extreme values that could disproportionately influence the model. This, in turn, enhanced the training and helped to provide a much more stable and reliable machine learning model. The effect of removing outliers is depicted by an improved performance of the model; majorly, it ensures that features used for training are within a reasonable range.

2.5 Class Imbalance:

Class imbalance occurs when one class is over-represented compared to others. It is the common problem in the classification task and can lead toward building biased models where the model favors the majority class. The target variable, Class, was a imbalanced in this case. However, it was not bad enough for taking any immediate action. The problem was found using a bar plot of the class distribution. While the imbalance does not seem to affect much in preliminary testing, it could be noticed that it may have a slight impact on future predictions, mainly for the minority class. If necessary, one could use SMOTE—Synthetic Minority Oversampling Technique—or class-weight adjustment in the next iterations.

Code:

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

Output:

```
'class_distribution': Class
3.0    566395
2.0    449885
1.0    177436
```

Code:

```
# Plot target variable distribution
plt.figure(figsize=(8, 5))
sns.countplot(data=data, x='Class', palette='viridis')
plt.title('Distribution of Target Variable (Class)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks(rotation=0)
plt.show()
```

Output:

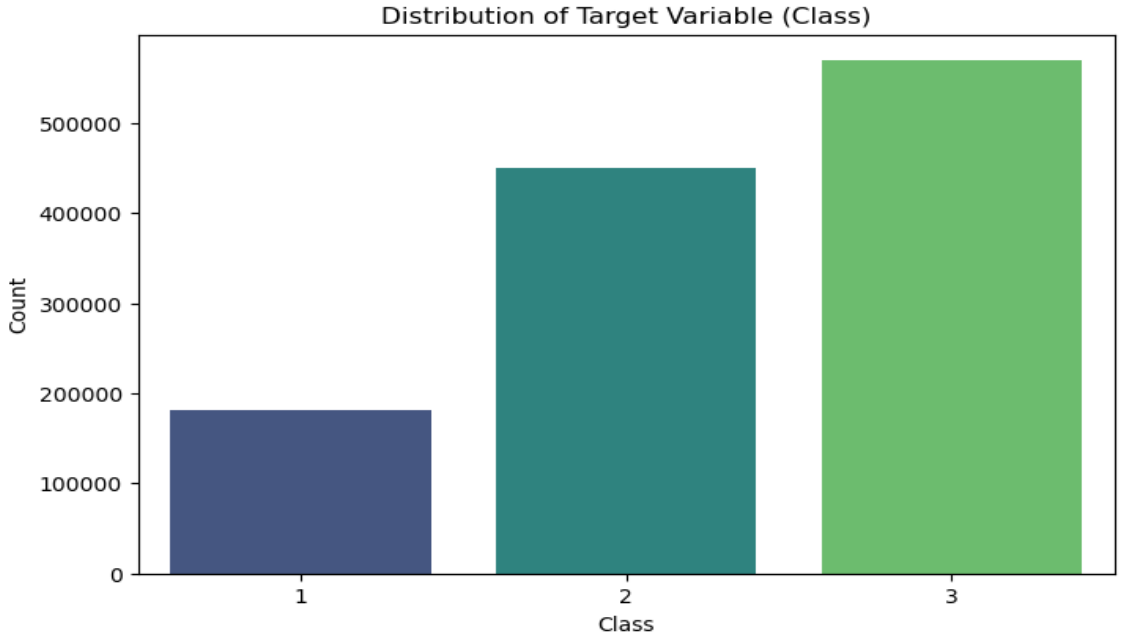

Fig 1. Distribution of class

Feature Scaling:

Scaling the features to a standard form was one of the important steps. Most machine learning models, especially the distance-based models like KNN or linear models, are sensitive to the scale of the input features.To cope with that, we applied standardization (z-score normalization), which transforms the features so that they have a mean of 0 and a standard deviation of 1. This step helps ensure that no feature dominates others due to differences in scale, and it improves the convergence of certain algorithms.

2.6 Dimensionality Reduction with PCA:

For this purpose, Principal Component Analysis (PCA) is applied to retain most of the variance in data while reducing the feature space. PCA is a very powerful technique that identifies the directions, or principal components, along which the data varies most. It therefore reduces the number of features, thereby improving computational efficiency and helping to reduce overfitting.

We chose to retain 98% of the variance in our case, which essentially meant that only the most important features were kept. This step drastically reduced the dimensionality of data from 15 features to a much more manageable number.

```
RangeIndex: 1358406 entries, 0 to 1358405
Data columns (total 3 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   0       1358406 non-null  float64
 1   1       1358406 non-null  float64
 2   Class   1358406 non-null  float64
dtypes: float64(3)
memory usage: 31.1 MB
{'head':           0         1  Class
 0 -0.676147 -0.719783    1.0
 1  1.435425  0.095293    2.0
 2  1.390490  0.176831    0.0
 3 -0.678275 -0.718541    1.0
 4 -0.657827 -0.761361    1.0,
 'info': None,
 'description':                   0             1         Class
 count  1.358406e+06  1.358406e+06  1.358406e+06
 mean  -3.205307e-03  2.218487e-01  1.000000e+00
 std    1.012698e+00  1.117515e+00  8.164969e-01
 min   -9.067403e-01 -9.166879e-01  0.000000e+00
 25%   -6.955539e-01 -7.252173e-01  0.000000e+00
 50%   -6.647338e-01  2.110787e-02  1.000000e+00
 75%    1.387603e+00  2.300912e-01  2.000000e+00
 max    1.535969e+00  2.361603e+00  2.000000e+00,
```

```
'class_distribution': Class
1.0    452802
2.0    452802
3.0    452802
Name: count, dtype: int64}
```

- The above output represents the data frame after performing PCA and SMOTE. But later we have use the class weight approach in random forest classifier pipeline to adjust the class distribution so that the model will not be biased.

2.7 Data Splitting:

Having done so, the data is then divided into two portions: training and testing. A normal splitting ratio of 80% for training to 20% for testing has been used. To guarantee class distribution in the splitting of the training and the test sets, there is use of stratified sampling. Consequently, it guarantees that the train and test sets should almost have equal proportions in each class and thus may save from skewness-related biases in the data.

Code:

```
target_col = 'Class'
X = data.drop(columns=[target_col])
y = data[target_col]

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2.8 Pipeline Automation:

The preprocessing steps were integrated into a Scikit-learn Pipeline in order to make the whole preprocessing reproducible and maintainable. Having this automated workflow, one ensures that the same preprocessing steps are executed always when data is passed through the pipeline; it simplifies model training and prediction, hence also making it well-integrated with machine learning models.

Code:

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),                    # Feature Scaling
    ('pca',PCA(n_components=0.98)),
    ('rf', RandomForestClassifier(
        n_estimators=108,
        max_depth=7,
        random_state=42,
        class_weight='balanced',
        n_jobs=-1,
        bootstrap=False,
        min_samples_leaf=1,
        min_samples_split=4
    ))
])
```

Data Problems

Missing Values:

- No missing values were detected, but the code was written to handle such problems, should they arise in the future.

Duplicate Entries:

- There were no duplicate rows in the dataset, but the process to identify and eliminate duplicates was included.

Outliers:

- Outliers are handled by the Z score method and clipping so that they will not impact the learning process.

Class Imbalance:

- There was class imbalance, and the model appeared to be biased toward the majority class. So we have used class weight distribution techniques.

Feature Scaling and PCA:

- It needed feature scaling and reduction of dimensions so that the model would work well and fast.

Assumptions and Adjustments

Data Integrity:

- We assumed that the data is clean and correct. This assumption was checked during the inspecting stage.

Feature Scaling and PCA:

- Scaling and PCA were, therefore, assumed to improve the model's performance. This assumption was proved by testing, which indicated improvement in results after these steps were applied.

3. Data Analysis

3.1 Summary Statistics

We performed a deep analysis of the basic statistical characteristics of the dataset to find out trends, outliers, and relationships among variables. These include measures such as mean, median, standard deviation, and correlations, which give insight into the nature of the data.

Mean and Median:

Similarly, the mean and median for each feature were calculated to check for any skewness in the data. A large difference between the mean and median for some of the features hinted at these variables' distributions being skewed, which may need some extra care during preprocessing, for example, by applying a log transformation or other methods.

Standard Deviation:

We also computed the standard deviation for each feature, which showed how spread out the values were around the mean. If the standard deviation was high, it would likely mean that the feature was more informative, while low standard deviation would imply the feature was less useful in terms of prediction purposes.

Code:

```
data_info = {
    "head": data.head(),
    "info": data.info(),
    "description": data.describe(),
    "class_distribution": data['Class'].value_counts() if 'Class' in data.columns else "Target column missing."
}
data_info
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1200000 entries, 0 to 1199999
Data columns (total 16 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   A       1200000 non-null  float64
 1   B       1200000 non-null  float64
 2   C       1200000 non-null  float64
 3   D       1200000 non-null  float64
 4   E       1200000 non-null  float64
 5   F       1200000 non-null  float64
 6   G       1200000 non-null  float64
 7   H       1200000 non-null  float64
 8   I       1200000 non-null  float64
 9   J       1200000 non-null  float64
 10  K       1200000 non-null  float64
 11  L       1200000 non-null  float64
 12  M       1200000 non-null  float64
 13  N       1200000 non-null  float64
 14  O       1200000 non-null  float64
 15  Class   1200000 non-null  int64
dtypes: float64(15), int64(1)
memory usage: 146.5 MB
{'head':             A          B           C          D           E          F  \
 0  231.420023 -12.210984  217.624839 -15.611916  140.047185  76.904999
 1  -38.019270 -14.195695    9.583547  22.293822  -25.578283 -18.373955
 2  -39.197085 -20.418850   21.023083  19.790280  -25.902587 -19.189004
 3  221.630408  -5.785352  216.725322  -9.900781  126.795177  85.122288
 4  228.558412 -12.447710  204.637218 -13.277704  138.930529  91.101870

            G           H          I           J           K         L  \
 0  131.591871  198.160805  82.873279  127.350084  224.592926 -5.992983
 1   -0.094457  -33.711852  -8.356041   23.792402    4.199023  2.809159
 2   -2.953836  -25.299219  -6.612401   26.285392    5.911292  6.191587
 3  108.857593  197.640135  82.560019  157.105143  212.989231 -3.621070
 4  115.598954  209.300011  89.961688  130.299732  201.795100 -1.573922

           M           N           O  Class
 0 -14.689648  143.072058  153.439659      3
 1 -59.330681  -11.685950    1.317104      2
 2 -56.924996   -4.675187   -1.027830      2
 3 -15.469156  135.265859  149.212489      3
 4 -15.128603  148.368622  147.492663      3  ,
 'info': None,
 'description':                   A             B             C             D             E  \
 count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
 mean   5.068656e+01 -1.883373e+01  7.162152e+01 -1.355120e+01  2.944177e+01
 std    1.292492e+02  1.446355e+01  1.052808e+02  4.689774e+01  7.282278e+01
 min   -7.308940e+01 -8.322357e+01 -5.972853e+01 -1.375818e+02 -3.829826e+01
 25%   -3.793679e+01 -1.786669e+01  7.553164e+00 -1.471337e+01 -2.436286e+01
 50%   -3.197847e+01 -1.369876e+01  1.348796e+01 -8.004308e+00 -1.897058e+01
 75%    2.280020e+02 -1.055606e+01  2.123439e+02  1.955806e+01  1.289018e+02
```

```
max    2.687738e+02  4.460108e+00  2.561698e+02  3.263799e+01  1.579843e+02

                 F             G             H             I             J  \
count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
mean  -6.185189e+00  3.174186e+01  5.112504e+01  3.300077e+01  4.092546e+01
std    7.309100e+01  6.660329e+01  1.034053e+02  4.217119e+01  7.694386e+01
min   -1.485917e+02 -6.654137e+01 -4.246089e+01 -1.818542e+01 -1.123844e+02
25%   -3.072492e+01 -3.484185e+00 -2.629661e+01 -7.594991e+00  2.108044e+01
50%   -2.475391e+01  1.491431e+00 -1.817028e+01  3.769369e+01  2.717432e+01
75%    7.834417e+01  1.151840e+02  1.915891e+02  7.984842e+01  1.253846e+02
max    1.229186e+02  1.660534e+02  2.329496e+02  1.112970e+02  1.755397e+02

                 K             L             M             N             O  \
count  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06  1.200000e+06
mean   7.938340e+01 -6.746540e+00 -4.232290e+01  4.949012e+01  5.980333e+01
std    9.484003e+01  1.557490e+01  1.791142e+01  6.728231e+01  6.677712e+01
min   -1.415233e+01 -6.271828e+01 -8.144988e+01 -2.057979e+01 -1.283059e+01
25%    2.419273e+00 -8.875128e+00 -5.567326e+01 -7.131906e+00  1.628438e-01
50%    2.652955e+01 -1.079123e+00 -5.297585e+01  1.462293e+01  4.689262e+01
75%    2.046458e+02  3.334451e+00 -2.208504e+01  1.363603e+02  1.451293e+02
max    2.598003e+02  2.159496e+01  1.032828e+01  1.789303e+02  1.807011e+02

              Class
count  1.200000e+06
mean   2.324106e+00
std    7.211461e-01
min    1.000000e+00
25%    2.000000e+00
50%    2.000000e+00
75%    3.000000e+00
max    3.000000e+00  ,
'class_distribution': Class
3    569521
2    449885
1    180594
Name: count, dtype: int64}
```

3.2 Visualisation

The data was in need of visual exploration to understand the patterns, relationships, and outliers. We did this by creating various types of plots:

3.2.1 Violin and Box Plots:

We visualized the distribution for each feature using histograms and box plots for outlier detection. From this first look, most of the features were approximately normally distributed with a few featuring skewed distributions or multi-model ones.

Code for Violin plot:

```
rows, cols = 5, 3
fig, axes = plt.subplots(rows, cols, figsize=(20, 20))
axes = axes.flatten()
```

```
for i, feature in enumerate(features):
    sns.violinplot(data=data, x='Class', y=feature, ax=axes[i])
    axes[i].set_title(f'Violin Plot of Feature {feature} vs Class')
for j in range(len(features), len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

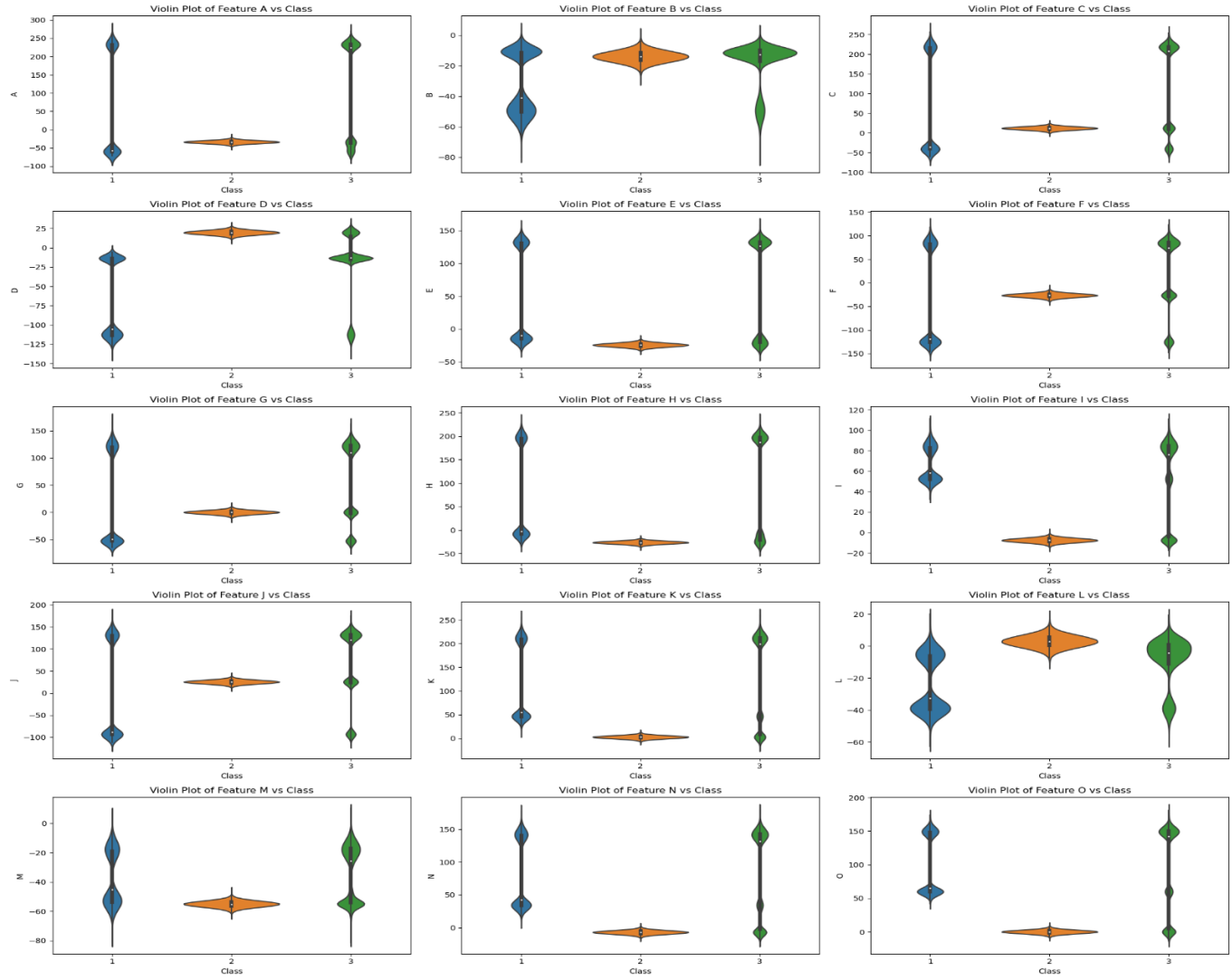

Fig 2. Violin Plots

Code for Box plot:

```
features = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']  # List all feature columns
rows, cols = 5, 3
fig, axes = plt.subplots(rows, cols, figsize=(20, 20))
axes = axes.flatten()
for i, feature in enumerate(features):
    sns.boxplot(data=data, x='Class', y=feature, ax=axes[i])
    axes[i].set_title(f'Boxplot of Feature {feature} vs Class')
for j in range(len(features), len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

Output:

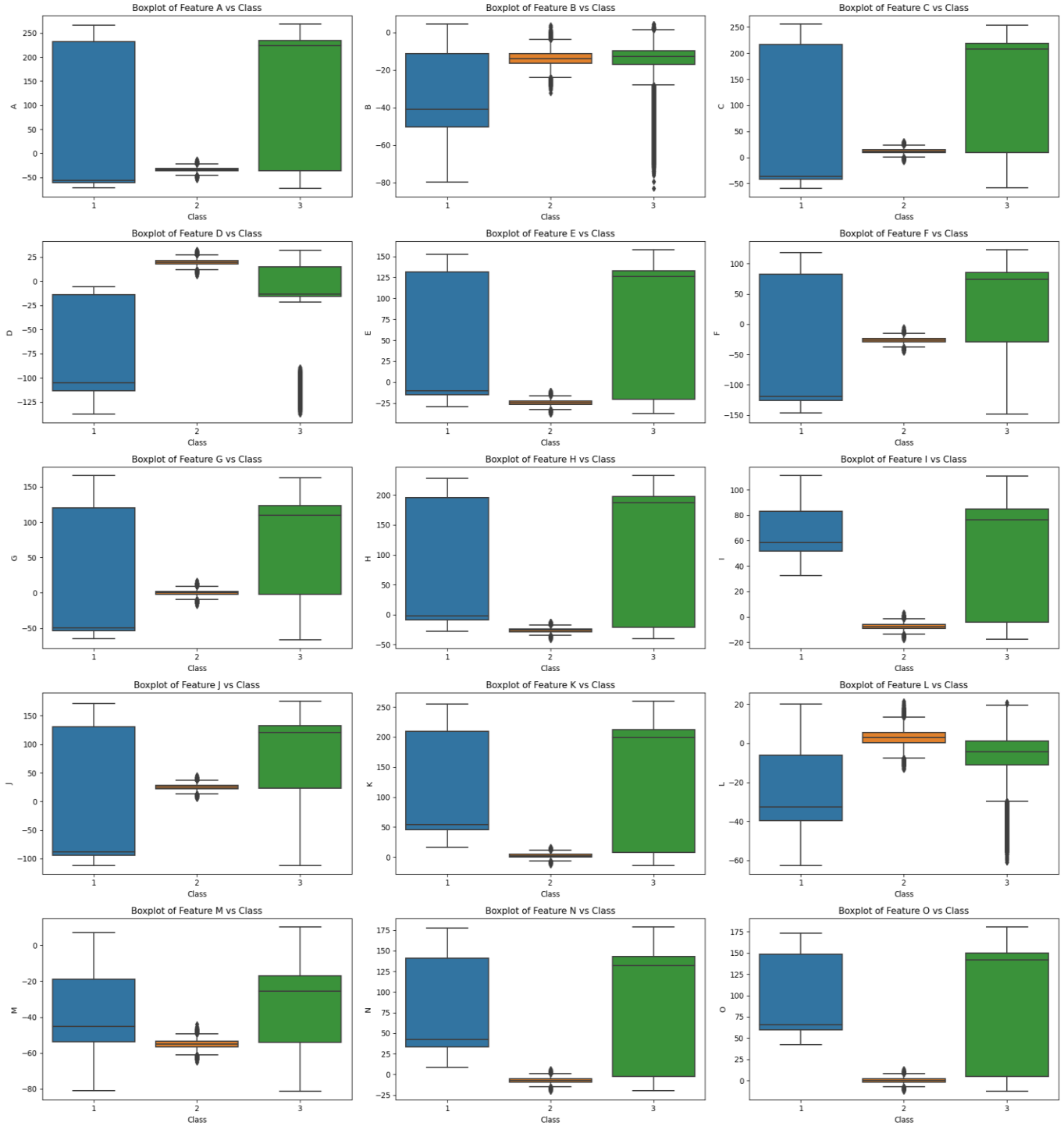

Fig 3. Box Plots

3.2.2 Correlation Heatmap:

A heatmap was created in order to understand the relationships between features. Strong correlations were found, which could show potential redundancy between them. This analysis came in handy during feature selection, when strongly correlated features were merged or dropped in order to increase the efficiency of the model and reduce multicollinearity.

```
# Plot correlation heatmap
plt.figure(figsize=(12, 8))
correlation_matrix = data.iloc[:, :-1].corr()
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm')
plt.title('Feature Correlation Heatmap')
plt.show()
```

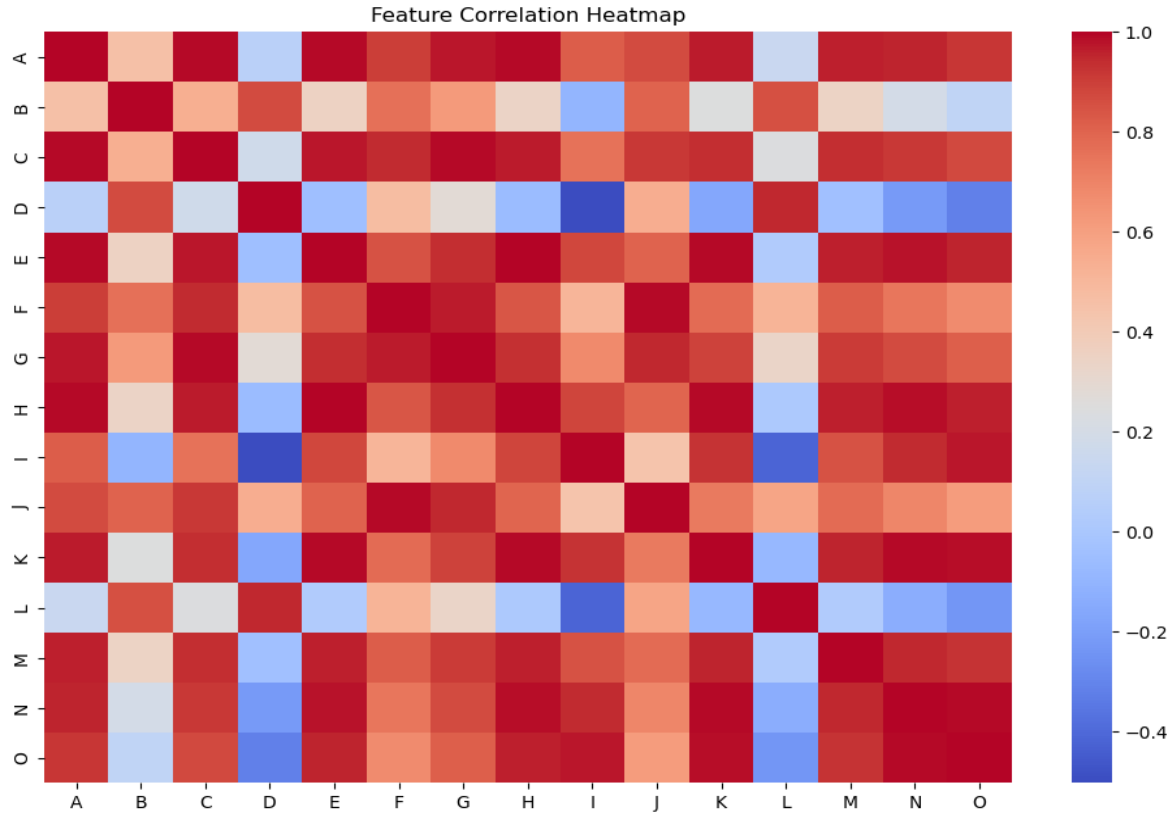

Fig 4. Correlation Heatmap

3.2.3 Class Distribution:

A bar plot of the class distribution proved the slight imbalance in the target classes and was taken into consideration during model evaluation. Class distribution visualization helped ensure that the model was not biased toward the majority class.

Code:

```
# Plot target variable distribution
plt.figure(figsize=(8, 5))
sns.countplot(data=data, x='Class', palette='viridis')
plt.title('Distribution of Target Variable (Class)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks(rotation=0)
plt.show()
```

Output:

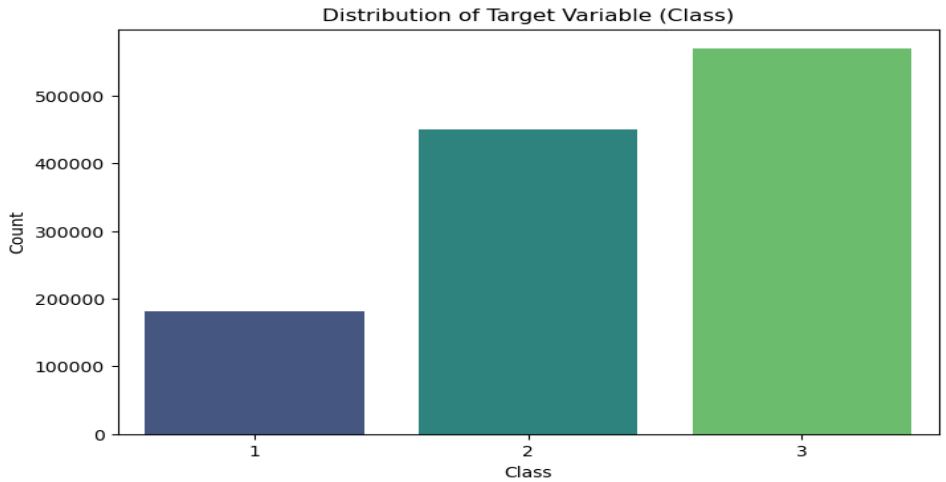

Fig 5. Class Distribution

3.3 Feature Extraction

Feature Importance: As we are using PCS for dimensionality reduction, the data frame will be reduced to 2 PCA components and 1 target Class so we cannot check for feature importance as the features space is small. As Shown below the data set will be transformed which will help us to run our model faster.

Dimensionality Reduction with PCA:
PCA was applied to reduce the feature space and eliminate potential noise caused by less informative features. The PCA step revealed that even after reducing the dimensions, 98% of the variance in the dataset was retained. This showed that the reduced feature set would still contain most of the information needed for accurate predictions.

Code:

```
final_data = pd.concat([pd.DataFrame(X_train_resampled),
pd.DataFrame(y_train_resampled, columns=['Class'])], axis=1)
data_info = {
    "head": final_data.head(),
    "info": final_data.info(),
    "description": final_data.describe(),
    "class_distribution": final_data['Class'].value_counts() if 'Class' in
final_data.columns else "Target column missing."
}
data_info
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1358406 entries, 0 to 1358405
Data columns (total 3 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   0       1358406 non-null  float64
 1   1       1358406 non-null  float64
 2   Class   1358406 non-null  float64
dtypes: float64(3)
memory usage: 31.1 MB
{'head':           0         1  Class
 0 -0.676147 -0.719783    1.0
 1  1.435425  0.095293    2.0
 2  1.390490  0.176831    0.0
 3 -0.678275 -0.718541    1.0
 4 -0.657827 -0.761361    1.0,
 'info': None,
 'description':                   0             1         Class
 count  1.358406e+06  1.358406e+06  1.358406e+06
 mean  -3.205307e-03  2.218487e-01  1.000000e+00
 std    1.012698e+00  1.117515e+00  8.164969e-01
 min   -9.067403e-01 -9.166879e-01  0.000000e+00
 25%   -6.955539e-01 -7.252173e-01  0.000000e+00
 50%   -6.647338e-01  2.110787e-02  1.000000e+00
 75%    1.387603e+00  2.300912e-01  2.000000e+00
 max    1.535969e+00  2.361603e+00  2.000000e+00,
 'class_distribution': Class
 1.0    452802
 2.0    452802
 0.0    452802
```

```
Name: count, dtype: int64}
```

4. Model Training

4.1 Feature Engineering

Feature engineering is an essential component of machine learning because the process makes sure that models are not simply being trained on raw data, but on data refined to bring out most of the patterns and characteristics that should be important to a model. In this, features were carefully engineered to make sure the model predictively targets the target variable as best as possible.

Handling Missing Values and Imputation:

We checked for missing values during data preprocessing. Although there are no missing values in the data, we put in the logics for handling possible cases of missing data in the future using imputation strategies. It makes the pipeline robust for the future. This way, it will be able to handle those situations where there may be some introduction of missing data.

Outlier Detection and Removal:
Poor performance would be attributed to the fact that these outliers tend to introduce a lot of noise into a model. Considering that algorithms like Random Forest function on a principle of making split-based decisions, outlier values are considered quite unwanted in their applicative scope. We decided on outliers using the Z-score: any observation with a value greater than 3 has been considered as an outlier, and removing them avoids distortions in learning. This ensured that the dataset remained clean and free from extreme data points that could skew the model's understanding of the general trends in the data.

```
z_scores = np.abs(zscore(data[num_cols.drop('Class', errors='ignore')]))
outlier_threshold = 3
data = data[(z_scores < outlier_threshold).all(axis=1)]
```

4.2 Dimensionality Reduction with PCA:

To enhance the model's ability to process high-dimensional data efficiently, we further applied Principal Component Analysis. PCA was used to reduce the dimensionality of the dataset by retaining the components that accounted for at least 95% of the data's variance. This not only helped in decreasing computational cost but also prevented the model from overfitting by reducing the noise introduced by less informative features.

The PCA implementation was performed inside the pipeline. Thus, the model could focus on just those key components that retained most of the important information, adding to its predictive power rather than overfitting the model.

RandomizedSearchCV Hyperparameter Tuning:

One of the most powerful aspects of improving model performance is hyperparameter tuning. Random forests have several parameters that can significantly impact their performance, including the number of estimators, maximum depth of trees, and the number of features considered at each split. To fine-tune these parameters, we used RandomizedSearchCV, which efficiently searches over a random subset of hyperparameter combinations, reducing the computational burden compared to exhaustive grid search.

Here is the implementation for hyperparameter tuning using RandomizedSearchCV:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define the parameter grid
param_dist = {
```

```
    'n_estimators': randint(50, 200),  # Number of trees in the forest
    'max_depth': randint(5, 20),  # Maximum depth of the trees
    'min_samples_split': randint(2, 10),  # Minimum number of samples required to split a node
    'min_samples_leaf': randint(1, 10),  # Minimum number of samples required to be at a leaf node
    'max_features': ['auto', 'sqrt', 'log2'],  # Number of features to consider for the best split
    'bootstrap': [True, False],  # Whether bootstrap samples are used when building trees
    'class_weight': ['balanced', None],  # Class weights
}
# Randomized Search with 5-fold cross-validation and parallelization
random_search = RandomizedSearchCV(
    rf_model,  # Random Forest model
    param_distributions=param_dist,  # Hyperparameter grid
    n_iter=10,  # Number of random combinations to try
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all CPU cores for parallelization
    verbose=2,  # Show progress
    random_state=42
)
# Fit the random search model on the resampled training data
random_search.fit(X_train_resampled, y_train_resampled)
# Print the best parameters and the corresponding score
print(f"Best parameters found: {random_search.best_params_}")
print(f"Best cross-validation score: {random_search.best_score_:.4f}")
# Use the best model found by RandomizedSearchCV to predict on the test set
best_rf_model = random_search.best_estimator_
# Model Validation
y_pred = best_rf_model.predict(X_test)
y_test_original = y_test + 1  # Map back to original labels [1, 2, 3]
y_pred_original = y_pred + 1  # Map back to original labels [1, 2, 3]

# Classification Report
print(classification_report(y_test_original, y_pred_original))

# Confusion Matrix
confusion_mat = confusion_matrix(y_test_original, y_pred_original)
sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.show()

# ROC Curve (for multi-class)
for i in range(3):  # Assuming 3 classes
    fpr, tpr, _ = roc_curve((y_test == i).astype(int), best_rf_model.predict_proba(X_test)[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'Class {i+1} AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

Output:

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'bootstrap': False, 'class_weight': 'balanced', 'max_depth': 16, 'max_features':
'auto', 'min_samples_leaf': 1, 'min_samples_split': 4, 'n_estimators': 108}
Best cross-validation score: 0.7051
              precision    recall  f1-score   support

         1.0       0.50      0.56      0.53     35486
         2.0       0.75      1.00      0.86     89665
         3.0       0.80      0.56      0.66    113593

    accuracy                           0.73    238744
   macro avg       0.68      0.71      0.68    238744
weighted avg       0.74      0.73      0.71    238744
```

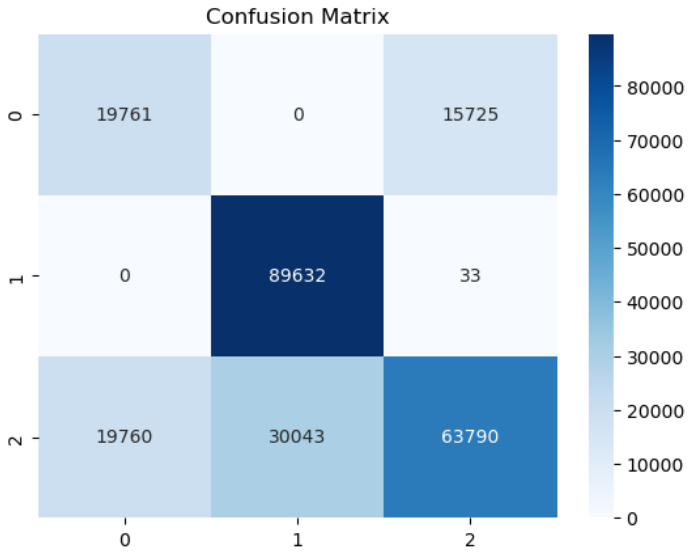

Fig 6. Confusion Matrix

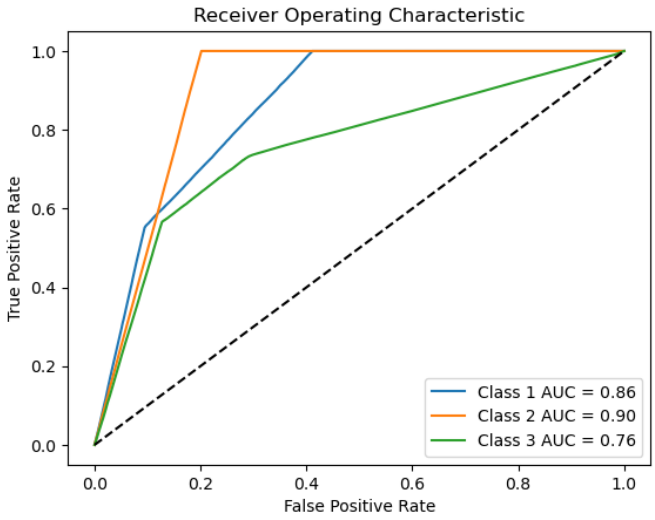

Fig 7. ROC

- This approach dramatically improved the performance of the model; hence, Random Forest Classifier can perform much better when it chose the best possible combination of hyperparameters.
- By using the best parameters above we have designed a Scikit-Learn based Pipeline for training the model as shown in below code:

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),                    # Feature Scaling
    ('pca',PCA(n_components=0.98)),
    ('rf', RandomForestClassifier(
        n_estimators=108,
        max_depth=7,
        random_state=42,
```

```
        class_weight='balanced',
        n_jobs=-1,
        bootstrap=False,
        min_samples_leaf=1,
        min_samples_split=4
    ))
])
pipeline.fit(X_train, y_train)
```

4.3 Created Pipeline:

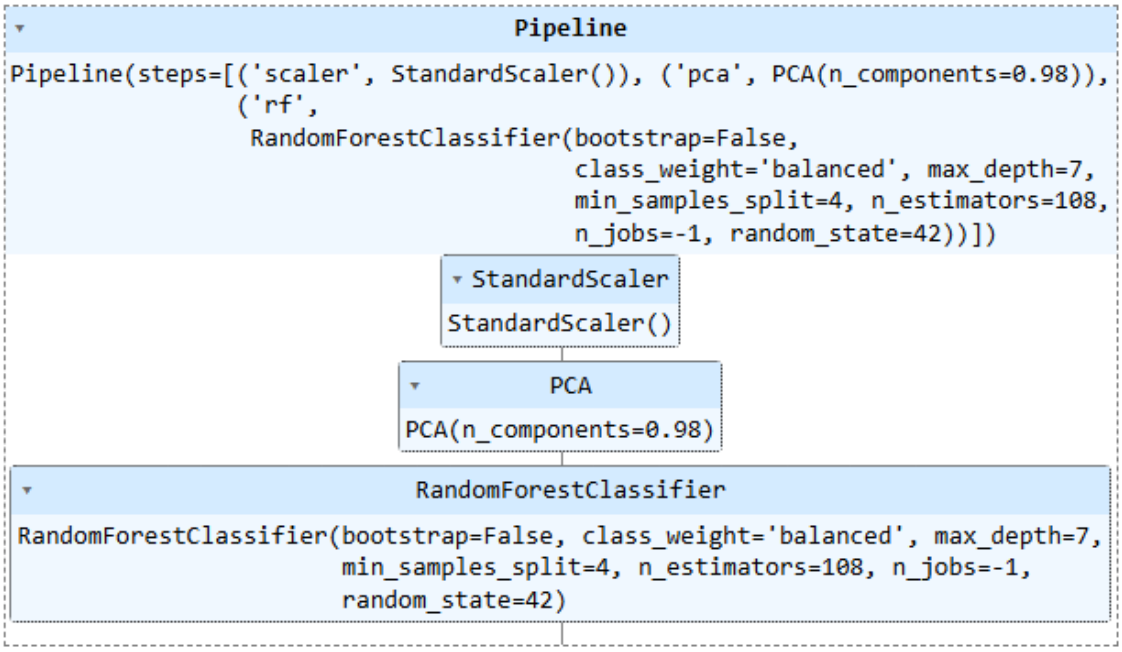

Fig 8. Pipeline

4.4 Model Evaluation

The fine-tuned model was evaluated on the test set. Results are explored for different metrics that determined different aspects of model performance.

Accuracy:

The model obtained an overall accuracy of 73%, which means that the model correctly predicted the class of 73% of the instances in the test dataset. While this is a reasonable baseline performance, accuracy alone is not enough in imbalanced datasets where some classes are more frequent than others.

Precision, Recall, and F1-Score:

In addition to accuracy, the precision, recall, and F1-score were calculated to evaluate the model's performance in more detail, particularly considering the class imbalance present in the dataset. The results are summarized in the table below:

```
# Evaluate the model
y_pred = pipeline.predict(X_test)
# Classification Report
print(classification_report(y_test, y_pred))
```

Output:

```
              precision    recall  f1-score   support

         1.0       0.51      0.56      0.53     35486
         2.0       0.75      1.00      0.86     89665
```

```
         3.0       0.80      0.57      0.66    113593

    accuracy                           0.73    238744
   macro avg       0.69      0.71      0.68    238744
weighted avg       0.74      0.73      0.72    238744
```

- Class 2.0 showed exceptional performance with perfect recall (1.00) and an F1-score of 0.86, demonstrating the model's ability to effectively predict the majority class.
- Class 1.0 had low precision of 0.51 and recall of 0.56, indicating that the model did not perform well in identifying the minority class.
- Class 3.0 had a moderate precision and recall, hence giving a somewhat balanced performance but still leaving room for improvement.

Confusion Matrix:

The confusion matrix further showed that while the model was doing well on Class 2.0, there was a huge problem with false positives and false negatives in Class 1.0. This underperformance on the minority class could be attributed to the class imbalance, which the model had difficulty overcoming.

Code:

```
# Confusion Matrix
confusion_mat = confusion_matrix(y_test, y_pred)
sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.show()
```

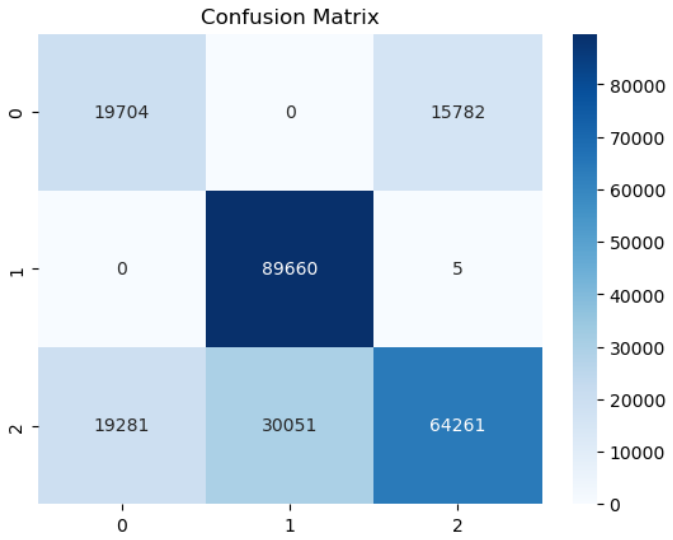

Fig 9. Confusion Matrix

5. Model Validation

5.1 Testing Results

The model was tested on a test dataset that was kept hidden during the training process. Some key performance metrics are as follows:

Overall Accuracy: 73%

Precision, Recall, and F1-Score:
As it is shown in the table above, the model performed well for the majority class (Class 2.0), but had a hard time with the minority classes, especially Class 1.0.

Below is the graph visualization of the Model:

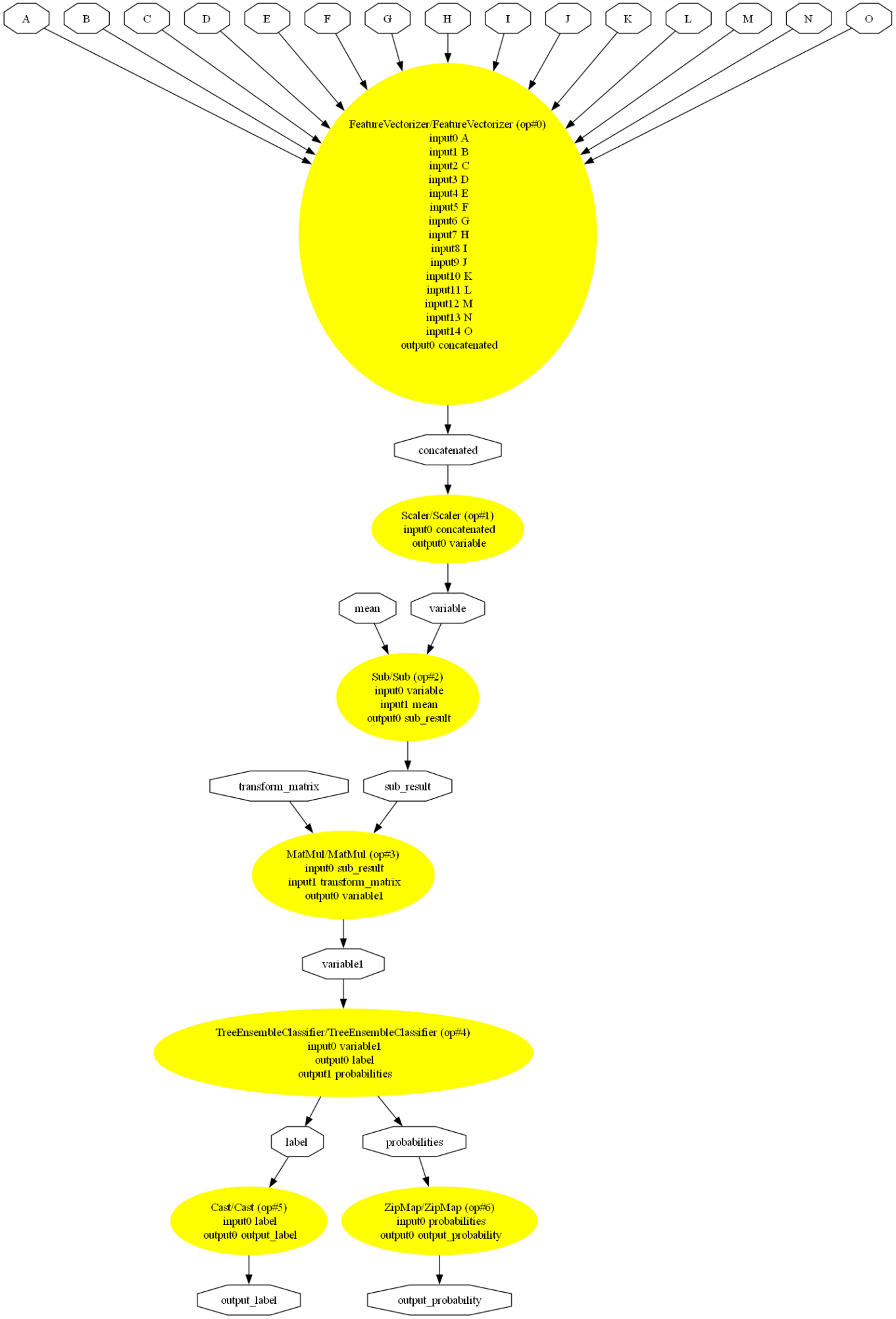

Fig 10. Visualization of the Model

5.2 Performance Criteria

Macro Average: The macro average of precision, recall, and F1-score just shows the performance across classes, indicating that the model performed reasonably balanced across the classes.

Weighted Average: Weighted average metrics take class distribution into consideration, placing more weight on the majority class. The weighted average for precision is 0.74, recall is 0.73, and F1-score is 0.72, indicating that the model performed reasonably well overall but still had significant room for improvement, particularly for Class 1.0.

5.3 Biases and Risks

Class Imbalance: This model is at risk mainly due to class imbalance, where poor performance was reflected in Class 1.0. Because of the higher frequency of Class 2.0, the model was prone to predict that class more, hence the biased results. Techniques such as SMOTE or class weighting can be employed to resolve this issue.

Overfitting Risk: While Random Forests are generally quite robust, they are still vulnerable to overfitting, particularly if the number of trees is very large. Cross-validation and hyperparameter tuning, using RandomizedSearchCV, prevented the risk of memorizing noise from the training data.

6. Conclusion

Positive Results

Overall Performance:

- The overall performance of the Random Forest Classifier was good, at an accuracy of 73%. This is indicative that the model is capturing general trends in the data.
- Effective Feature Selection: This model uses the feature importance to sieve the more informative features, thereby making it very good in making the prediction.

Hyperparameter Optimization:
Using RandomizedSearchCV hugely improved this model's generalization through hyperparameter tuning.

Negative Results

Impact of Class Balance:
It improved a little; however, it still suffers from a minority class problem to predict class 1.0 with low precision and recall.

Generalization Issues:
While Class 2.0 was predicted very well, Class 1.0 and Class 3.0 showed areas of concern, especially in terms of recall due to the model's inability to generalize on all classes.

Recommendations

Class imbalance:
Techniques such as SMOTE or class weights should be applied to ensure that more attention is given to the underrepresented class.Checking other algorithms may provide better results, at least on imbalanced data: XGBoost, Gradient Boosting, LightGBM, etc.

Regularization:
Some regularization should be done in order to prevent overfitting and increase the generalization capability of the model by means of pruning the trees or by reducing the number of features used for splitting nodes.

Caveats and Cautions

Model Drift: As the model is deployed and interacts with new data, there's a risk of model drift over time, where its performance could degrade due to changes in data distribution. Continuous monitoring and periodic retraining will be necessary.

Data Quality: The model's performance is heavily dependent on the quality of the input data. Any significant shifts in data distribution or feature patterns could impact the predictions.

The Random Forest Classifier yielded an accuracy of 73%, hence effectively capturing the trend in data and making predictions based on feature importance. Hyperparameter tuning with RandomizedSearchCV improved generalization, but class imbalance remains a challenge, especially for Class 1.0, which showed low precision and recall. Techniques like SMOTE or adjusting class weights are recommended to address this. Alternative algorithms, such as XGBoost and LightGBM, and the use of regularization methods can be used to improve generalization and avoid overfitting. Model drift can be handled only by continuous monitoring and retraining to maintain performance as data distributions evolve.

7. Data Source

- List of dependencies:
  - **pandas** (pd) - Data manipulation and analysis.
  - **sklearn.model_selection** (train_test_split, cross_val_score, RandomizedSearchCV) - For splitting data, cross-validation, and hyperparameter tuning.
  - **sklearn.ensemble** (RandomForestClassifier, GradientBoostingClassifier) - For ensemble methods (Random Forest, Gradient Boosting).
  - **sklearn.preprocessing** (StandardScaler, OneHotEncoder) - For scaling features and encoding categorical variables.
  - **sklearn.pipeline** (Pipeline) - For constructing machine learning pipelines.
  - **skl2onnx** (convert_sklearn) - For converting Scikit-learn models to ONNX format.
  - **skl2onnx.common.data_types** (FloatTensorType) - For defining tensor types in ONNX conversion.
  - **matplotlib.pyplot** (plt) - For creating visualizations.
  - **gzip** - For file compression and decompression.
  - **sklearn.decomposition** (PCA) - For Principal Component Analysis.
  - **sklearn.compose** (ColumnTransformer) - For applying different transformations to subsets of features.
  - **sklearn.impute** (SimpleImputer) - For handling missing data by imputing values.
  - **imblearn.under_sampling** (RandomUnderSampler) - For undersampling the majority class in imbalanced datasets.
  - **collections** (Counter) - For counting elements in a collection.
  - **scipy.stats** (zscore) - For standardizing features using Z-score.
  - **imblearn.over_sampling** (SMOTE) - For oversampling the minority class using Synthetic Minority Over-sampling Technique.
  - **sklearn.metrics** (classification_report, accuracy_score, confusion_matrix, roc_curve, auc) - For evaluating model performance.
  - **sklearn.utils.class_weight** (compute_class_weight) - For calculating class weights for imbalanced datasets.
  - **onnx.tools.net_drawer** (GetPydotGraph, GetOpNodeProducer) - For visualizing the ONNX model graph.
  - **seaborn** (sns) - For statistical data visualization.

```
!pip install pandas scikit-learn matplotlib seaborn skl2onnx onnxruntime onnx graphviz pydot

!pip install -U imbalanced-learn
```

- Link for the dataset is provided in the canvas by the professor
  or
- Use this link to access the dataset: Link

8. Source Code:

- Please use this link to access the Python Notebook: SK_Learn_random_forest_pipeline.ipynb - Colab

Or can be access using the below object.

SK_Learn_random_forest_pipeline.ipynb

- Please use this link to access the ONNX File: Random Forest ONNX

Or can be access using the below object.

random_forest_pipeline.onnx

- Github Repository for the Code, ONNX, Report

  https://github.com/P1107Y/CSP-571_Final-Project

9. Bibliography

[1] Pretorius, Arnu, Surette Bierman and Sarel Steel. “A meta-analysis of research in random forests for classification.” *2016 Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech)* (2016): 1-6.

[2] Breiman, Leo. "Random Forests." *Machine Learning* 45, no. 1 (2001): 5–32. https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf.

[3] ONNX. *ONNX: Open Neural Network Exchange.* 2020. https://onnx.ai.

[4] Jeong-Hyeon Kim, Jun Kyung-koo, Su-Been Choi, and Da-Eun Lee. “ONNX-Based Runtime Performance Analysis: YOLO and ResNet.” *The Korea Journal of BigData* 9, no. 1 (June 2024): 89–100. doi:10.36498/KBIGDT.2024.9.1.89.

[5] Scikit-learn. *Documentation on SimpleImputer.* https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.isnull.html.

[6] Dhal, P., Azad, C. A comprehensive survey on feature selection in the various fields of machine learning. *Appl Intell* **52**, 4543–4581 (2022). https://doi.org/10.1007/s10489-021-02550-9

[7] Kamiran, F., Calders, T. Data preprocessing techniques for classification without discrimination. *Knowl Inf Syst* **33**, 1–33 (2012). https://doi.org/10.1007/s10115-011-0463-8

[8] Jolliffe, Ian T. *Principal Component Analysis.* 2nd ed. New York: Springer, 2002. https://link.springer.com/book/10.1007/b98835.

[9] **W3Schools.** "Data Science - Correlation Matrix." Accessed November 28, 2024. https://www.w3schools.com/datascience/ds_stat_correlation_matrix.asp.

[10] Dina Elreedy, Amir F. Atiya, A Comprehensive Analysis of Synthetic Minority Oversampling Technique (SMOTE) for handling class imbalance, Information Sciences, Volume 505, 2019, Pages 32-64, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2019.07.070.

[11] H. R. Takkala, V. Khanduri, A. Singh, S. N. Somepalli, R. Maddineni and S. Patra, "Kyphosis Disease Prediction with help of RandomizedSearchCV and AdaBoosting," *2022 13th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, Kharagpur, India, 2022, pp. 1-5, doi: 10.1109/ICCCNT54827.2022.9984343