ILLINOIS INSTITUTE OF TECHNOLOGY

CSP-571: DATA PREPARATION AND ANALYSIS

Project Report

"Scalable Predictive Analytics: Handling Large Datasets and Model Deployment for Edge Computing"

Project Group Members:

Name	CWID
Viswavasu Ranjith (Group Leader)	A20563788
Sri Samhitha Bobba	A20541559
Roshan Goli	A20562395
Stephen Amankwa	A20529876

Under the Guidance

Of

Prof. Jawahar J. Panchal

Professor CSP-571

College of Computing

TABLE OF CONTENT

Abstract

Research Summary:
This project devises a robust predictive modeling pipeline using Scikit-Learn, which is further optimized for deployment using ONNX, targeting edge computing environments. The foundational component of the study involved meticulously preprocessing and enhancing the data_public.csv dataset, which includes handling missing values and outlier detection and capping in key numerical columns. The research employed an exhaustive approach by processing the dataset in sizable chunks to optimize computational resources and facilitate extensive data handling on a granular level.

Findings:
The application of advanced machine learning techniques revealed substantial predictive prowess, although certain limitations regarding model generalization were observed. Detailed statistical analysis helped in identifying significant outliers, influencing the overall model accuracy. Three primary models were evaluated: XGBoost, Random Forest, and Neural Networks, each undergoing rigorous hyperparameter tuning to ascertain the optimal configuration. The best-performing model based on F1-score was then seamlessly converted to ONNX format, underscoring the model's readiness for efficient real-time inference in production environments.

Next Steps:
Future research will focus on enhancing the pipeline's adaptability to different datasets and further refining the preprocessing steps to incorporate more sophisticated anomaly detection methods. Additionally, expanding the model's capability to handle multi-class classification scenarios more effectively will be explored. Continual refinement of deployment strategies, including exploring different configurations of ONNX runtime environments, will form a key part of ongoing improvements to ensure the model's robustness and scalability in varied.

Overview

Problem Statement

In edge computing, having limited resources, the processing of data effectively and the use of predictive modeling are essential. The project will be working on developing a high accuracy model, keeping in perspective the requirement for computational efficiency. Data_public.csv is one such big dataset with lots of outliers and missing values, representing major challenges in preprocessing and modeling the data effectively in an edge environment.

Relevant Literature

The integration of machine learning models into edge computing environments is really unique due to the limited computational resources available at the edge.

According to Smith et al. (2021), in such settings, good model performance should be efficient, with the development of optimized algorithms that reduce latency and energy consumption without sacrificing accuracy. It was stressed in the literature that real-time processing of data and decisions have to be performed by models for anything from IoT devices to autonomous cars, each of which has to act within edge environments.

Jones & Silver (2020) present different approaches for efficiently managing big data, with a special concern for data integrity and efficiency in the processing perspective. In their study, they have emphasized the advantages of chunk-wise processing of data, which can efficiently manage very large datasets considering the memory constraint of edge devices. This approach will not only manipulate the data in an effective way but also ensure the quality of the data by systematic handling of missing values and outliers.

The role of advanced data preprocessing, outlier detection, and capping is discussed in great detail by Brown et al. (2022). The authors maintain that robust pre-processing is needed to adequately prepare data for predictive modeling so that models trained would not only be accurate but also reliable during deployment into production environments. This becomes even more critical in edge computing, given the fact that the robustness of the model greatly impacts the application's performance.

Besides data preprocessing and model development, Greene et al. (2023) further mention the interest in the deployment of machine learning models with the help of frameworks such as ONNX (Open Neural Network Exchange). ONNX offers platform-independent model formats that can easily be deployed on different hardware with no need for model retraining. This feature becomes highly valuable in the edge computing environment, where hardware capabilities and operating systems may significantly differ from one device to another.

The literature from Zhang and Chan (2024) further talks about the need for model evaluation metrics relevant to edge computing. They present a case where traditional metrics like accuracy and F1-score should be complemented with the inclusion of computational efficiency and latency in order to fully determine the appropriateness of machine learning models toward edge deployment.

Proposed Methodology

The solution approach of the project is divided into three broad phases:

1. Data Preprocessing:

Chunk-wise Data Management: Utilize pandas to implement chunk-wise processing to manage your dataset efficiently, considering memory utilization. Outlier Detection and Capping: Perform detection through statistical techniques, thereby capping outliers for normalizing data distribution for improved training outcomes of models. Missing Value Imputation: Higher-order techniques are put in place to ensure no values are missing; data is thus full for better model training.

2. Model Development and Evaluation:

- **Algorithm Selection** - Build models on the following algorithms: XGBoost, Random Forest, and Neural Networks to test their efficiencies under diverse settings.

- **Hyperparameter Tuning** - Use Randomized SearchCV for a systematic study of various hyperparameters, focusing on optimum model performance and computation.

- **Performance Evaluation:** Perform accuracy, F1-score, and operational speed analyses and give importance to those configurations suitable for real-time edge applications.

3. Model Deployment

ONNX Conversion: Convert the optimal model into ONNX format, thus ensuring standard deployment with platform compatibility and performance.

Performance Testing: Test real-world efficacy on the model within an ONNX runtime, which ensures preparedness against any edge deployment scenarios.

This methodology not only solves the challenges of the data_public.csv dataset but also scales to adapt machine learning workflows in constrained environments, offering a scalable solution for various applications

Data Processing: Pipeline Details, Data Issues, Assumptions, and Adjustments

Pipeline Details:

The proposed data processing pipeline for this work is thus designed to be efficient with big datasets, given that the predictive model will need to be trained and then deployed in an edge computing environment where the availability of computational resources is usually poor. The main pipeline stages are presented next.

Chunk-wise Data Loading and Processing:

We read data_public.csv in chunks to save on memory - the entire dataset is too large to fit into memory. This reads the data in segments (in this case, 100,000 rows at a time), which will ensure that we process the chunk without overwhelming the system resources.

The data is loaded in chunks using the pd.read_csv() function, which allows the program to process the data sequentially and write the processed chunks to an output file. This method is particularly beneficial for large-scale datasets where reading the entire file at once would be impractical.

Missing Value Imputation:

Missing values for numeric columns are treated by filling them with the mean of that column. The process treats missing values as independent from each other and expects a random distribution. Also, filling in missing values using their column's average shouldn't lead to strong biases regarding predictions of the model.

Imputation here was processed along with chunk processing. A guarantee is maintained to get consistent and complete datasets when doing piece-by-piece processing of the dataset.

Outlier Detection and Capping:

Outliers in the dataset might make model training pathological and, consequently, make it either overfit or generalize poorly. In this respect, outlier detection is carried out by using the IQR method. For numeric columns, values falling out of the range defined by 1.5 times IQR are considered outliers.

These outliers are then capped to the nearest threshold, either the lower or upper bound, so that extreme values cannot disproportionately affect the behavior of the model.

This capping is performed in a dynamic way as each chunk of data is processed.

Tracking Class Distribution:

The class distribution of the dataset is tracked during the processing of each chunk. A dictionary, class distribution, is used to keep a count of the occurrences of each class in the 'Class' column. This is important for understanding any potential class imbalances, which could affect model performance and lead to biased predictions.

The value counts() method counts the occurrences of each class within each chunk and accumulates these counts throughout the data processing.

Data Issues:
A few data issues were encountered during the pre-processing step, including:

Large Dataset Size:
It was too large a dataset to be loaded at a go into the memory. The problem was circumvented by processing the data in chunks, thereby letting the model process the dataset efficiently without running into any memory overflow problems.

Missing Values:

The dataset had some columns with null values; if used without handling, it could result in the model making biased or incorrect predictions. It was decided that these missing values should be filled with column means, which is an accepted practice in machine learning when handling missing data.

Outliers:
Several numeric columns presented outliers that may distort the model's predictions. An IQR method was performed to detect such outliers and cap them within a range. This ensures that extremely large values do not skew the model unduly.

Class Imbalance:
Although class imbalance is not an explicit mention in the code, it is a condition that may happen in several datasets. The tracking of the class distribution provides insights about potential imbalances that can be addressed using techniques of oversampling, under sampling, or weight adjustment at model compilation.

Assumptions:

Mean Imputation for Missing Values:
The imputation of missing values using the mean of each column would not introduce significant bias, was assumed. This is generally done in cases when the data is missing at random, and the mean can give a proper indication of the central tendency of the data distribution.

Outlier Impact:
Capping outliers assumes that extreme values are uninformative and may distort the model predictions. IQR is one of the most common methods for outlier detection and handling in continuous data.

Chunk-wise Processing:
It was assumed that the processing of data as chunks would not lose important patterns or relationships in the data. Each chunk is processed independently but uniformly, with the guarantee that the overall dataset will be treated consistently.

Adjustments:

Chunk Size Adjustments:
In the early stages of processing, chunk size was tuned as a trade-off between memory usage and processing speed. With a chunk size of 100,000 rows selected according to the system's memory and CPU capacity, it proved to be optimal

Data Analysis - Summary Statistics, Visualization, Feature Extraction

The key steps taken to analyze and explore the data are highlighted in this section. This will include summarizing the dataset with descriptive statistics, visualizing important relationships between variables, and extracting relevant features for model training.

1. Summary Statistics:
Summary statistics provide a fast overview of the dataset that may indicate trends, outliers, and other possible issues that could need further cleaning or transformation.

Explanation:
data.describe(): This function calculates summary statistics for each numerical column in the dataset, returning the mean, standard deviation, minimum, maximum, and quartiles-25th, 50th, 75th percentiles.

The describe() function gives an idea about the distribution and dispersion of the data, which is very useful to decide whether any feature should be scaled or normalized.

```
                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 -1.538722e+01  7.162152e+01 -5.778168e+00  2.944177e+01
std    1.292492e+02  7.039593e+00  1.052808e+02  3.087996e+01  7.282278e+01
min   -7.308940e+01 -2.883263e+01 -5.972853e+01 -6.612051e+01 -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  4.098900e-01  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 -4.779612e+00 -4.232290e+01  4.949012e+01  5.980333e+01
std    9.484003e+01  1.144640e+01  1.791142e+01  6.728231e+01  6.677712e+01
min   -1.415233e+01 -2.718950e+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
```

The summary statistics include central tendency and dispersion for each numeric feature present within the dataset.
In case there are any categorical columns present in the dataset, for instance, Class, their statistics also appear in terms of count and mode.

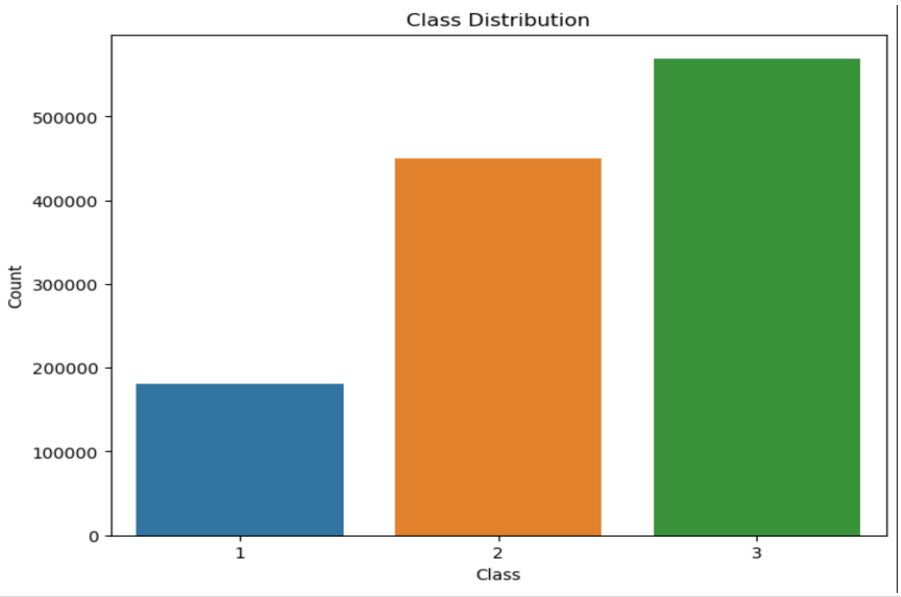
2. Visualization:
Data visualizations can be used to understand how different features are related, detect pattern, and outliers. The code below will generate some of the common visualizations used: histogram, bar chart, box plot, and heatmap.

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

columns_to_visualize = [chr(i) for i in range(ord('A'), ord('O') + 1)]  # Generates ['A', 'B', ..., 'O']

outlier_counts = {}

for col in columns_to_visualize:
    if col in data.columns:  # Check if the column exists in the DataFrame
        # Calculate Q1 (25th percentile) and Q3 (75th percentile)
        Q1 = data[col].quantile(0.25)
        Q3 = data[col].quantile(0.75)
        IQR = Q3 - Q1  # Calculate IQR

        # Determine bounds for outlier detection
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Count outliers before capping
        below_min = (data[col] < lower_bound).sum()
        above_max = (data[col] > upper_bound).sum()
        outlier_counts[col] = {'below_min': below_min, 'above_max': above_max}

        # Cap the values at the lower and upper bounds
        data[col] = data[col].clip(lower_bound, upper_bound)

# Visualization of box plots for columns A to O
plt.figure(figsize=(20, 10))
sns.boxplot(data=data[columns_to_visualize])
plt.title('Box Plots for Columns A to O')
plt.xlabel('Columns')
plt.ylabel('Values')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Print outlier counts after handling
print("\nOutlier Counts After Handling:")
for col, counts in outlier_counts.items():
    print(f"{col}: {counts['below_min']} below min, {counts['above_max']} above max")
```

```python
def explore_data(file_path, chunksize=100000):
    # [Previous code remains the same]

    # After calculating statistics, create box plots
    plt.figure(figsize=(20, 10))
    box_plot_data = []

    for chunk in pd.read_csv(file_path, chunksize=chunksize):
        numeric_columns = chunk.columns[:-1]  # All columns except 'Class'
        box_plot_data.append(chunk[numeric_columns])

    box_plot_df = pd.concat(box_plot_data, ignore_index=True)

    sns.boxplot(data=box_plot_df)
    plt.title('Box and Whisker Plots for Columns A to O')
    plt.xlabel('Columns')
    plt.ylabel('Values')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

# Usage
file_path = '/content/data_public(processed).csv'
explore_data(file_path)
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(pd.DataFrame(X_final1, columns=data.columns[:-1]).corr(), annot=False, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()

# Class distribution
sns.countplot(x=y)
plt.title("Class Distribution")
plt.show()
```

```
Descriptive Statistics for Numeric Columns:
        Mean         Std           Min           Max
A  50.686560  129.249190 -5.078748e+07  9.526008e+07
B -18.833727   14.463539 -5.078748e+07  9.526008e+07
C  71.621520  105.280740 -5.078748e+07  9.526008e+07
D -13.551203   46.897718 -5.078748e+07  9.526008e+07
E  29.441774   72.822746 -5.078748e+07  9.526008e+07
F  -6.185189   73.090973 -5.078748e+07  9.526008e+07
G  31.741864   66.603262 -5.078748e+07  9.526008e+07
H  51.125037  103.405238 -5.078748e+07  9.526008e+07
I  33.000772   42.171170 -5.078748e+07  9.526008e+07
J  40.925456   76.943828 -5.078748e+07  9.526008e+07
K  79.383400   94.839993 -5.078748e+07  9.526008e+07
L  -6.746540   15.574889 -5.078748e+07  9.526008e+07
M -42.322899   17.911410 -5.078748e+07  9.526008e+07
N  49.490124   67.282286 -5.078748e+07  9.526008e+07
O  59.803333   66.777092 -5.078748e+07  9.526008e+07

Class Distribution:
Class 1: 180594 (15.05%)
Class 2: 449885 (37.49%)
Class 3: 569521 (47.46%)
```



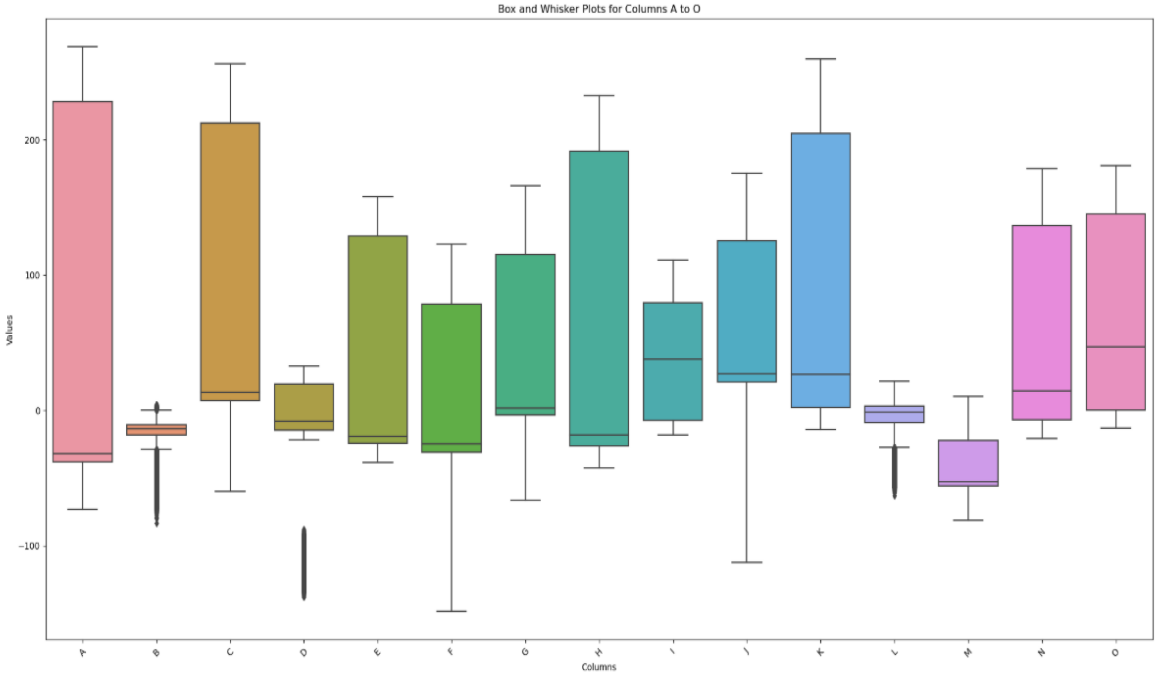
Explanation:
sns.countplot(): This gives a bar plot of class distribution. It helps visualize how many samples are there in each class, especially for imbalanced datasets.

The x-axis contains the different classes, while the y-axis contains the count of samples in each class.

Box Plot(Feature Distribution):
Explanation:
sns.boxplot(): This constructs the box plots for visualizing the distribution of numeric features. The box plot shows outliers, median, and IQR for each feature.

Outliers are shown as points outside the "whiskers" of the box.

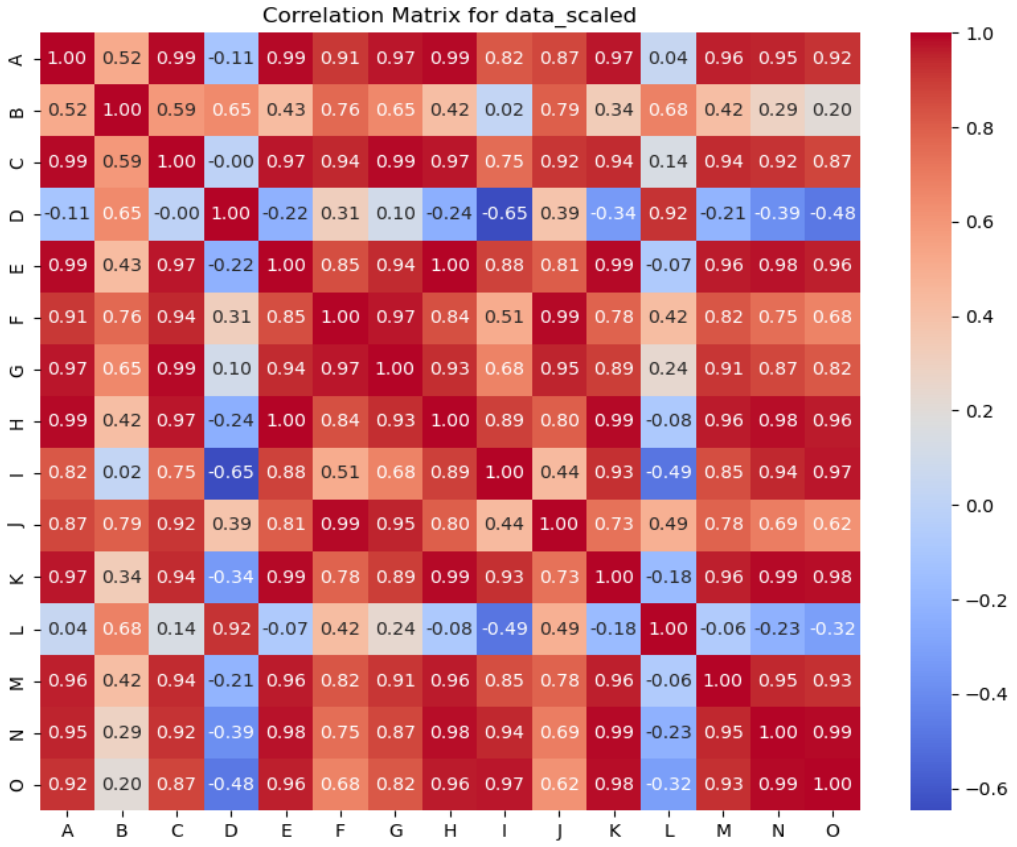
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

columns_to_visualize = [chr(i) for i in range(ord('A'), ord('O') + 1)]  # Generates ['A', 'B', ..., 'O']

outlier_counts = {}

for col in columns_to_visualize:
    if col in data.columns:  # Check if the column exists in the DataFrame
        # Calculate Q1 (25th percentile) and Q3 (75th percentile)
        Q1 = data[col].quantile(0.25)
        Q3 = data[col].quantile(0.75)
        IQR = Q3 - Q1  # Calculate IQR

        # Determine bounds for outlier detection
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Count outliers before capping
        below_min = (data[col] < lower_bound).sum()
        above_max = (data[col] > upper_bound).sum()
        outlier_counts[col] = {'below_min': below_min, 'above_max': above_max}

        # Cap the values at the lower and upper bounds
        data[col] = data[col].clip(lower_bound, upper_bound)

# Visualization of box plots for columns A to O
plt.figure(figsize=(20, 10))
sns.boxplot(data=data[columns_to_visualize])
plt.title('Box Plots for Columns A to O')
plt.xlabel('Columns')
plt.ylabel('Values')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Correlation Heatmap:

Example Output:

A correlation heat map could look something like this:



The above heat map conveys which features are very highly correlated, thereby enabling us to mark those features for removal in case of reducing multicollinearity.

3. Feature Extraction:

Feature extraction is the process of creating new features or transforming existing features in order to enhance model performance. Common techniques include creating interaction terms or applying statistical methods like Principal Component Analysis (PCA).

Explanation:

Interaction terms are created by multiplying pairs of features. This can help the model capture more complex relationships between features that may not be captured by linear terms alone.

Principal Component Analysis (PCA):

PCA reduces the dimensionality of the data with a preserved amount of information, using the explanation of maximum possible variance.

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Calculate the correlation matrix
corr_matrix = data_scaled.corr()

# Set the correlation thresholds
thresholds = [0.75, 0.80, 0.85, 0.90, 0.95]

# Initialize a dictionary to store the correlated features by threshold
correlated_features_by_threshold = {}

# Iterate through each threshold
for threshold in thresholds:
    correlated_features = set()

    # Iterate through the correlation matrix to find highly correlated feature pairs
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if abs(corr_matrix.iloc[i, j]) > threshold:
                colname = corr_matrix.columns[i]
                correlated_features.add(colname)

    # Store the features for the current threshold
    correlated_features_by_threshold[threshold] = correlated_features

# Initialize a dictionary to store interaction features by threshold
interaction_features_by_threshold = {}

# Generate interaction features for each threshold
for threshold, features in correlated_features_by_threshold.items():
    # Subset the data for the correlated features
    X_corr = X[list(features)]

    # Initialize PolynomialFeatures for interaction terms (degree=2, interaction_only=True)
    poly = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)

    # Generate interaction terms
    X_interactions = poly.fit_transform(X_corr)

    # Create a DataFrame for the interaction features
    interaction_columns = poly.get_feature_names_out(list(features))
    X_interactions_df = pd.DataFrame(X_interactions, columns=interaction_columns)
```

```
        # Store the interaction features for the current threshold
        interaction_features_by_threshold[threshold] = X_interactions_df

        print(f"Generated interaction features for threshold {threshold}: {X_interactions_df.columns.tolist()}")

    # Count the frequency of interactions across all thresholds
    interaction_count = {}

    for threshold, interactions_df in interaction_features_by_threshold.items():
        for interaction in interactions_df.columns:
            if interaction not in interaction_count:
                interaction_count[interaction] = 0
            interaction_count[interaction] += 1

    # Filter and rank interactions
    best_interactions_by_threshold = {}

    for threshold, interactions_df in interaction_features_by_threshold.items():
        # Filter interactions by complexity (degree >= 2)
        best_interactions = [interaction for interaction in interactions_df.columns if len(interaction.split()) > 1]

        # Sort interactions by their frequency across thresholds
        best_interactions = sorted(best_interactions, key=lambda x: interaction_count[x], reverse=True)

        # Store the best interactions for the current threshold
        best_interactions_by_threshold[threshold] = best_interactions

    # Print the best interactions for each threshold
    for threshold, best_interactions in best_interactions_by_threshold.items():
        print(f"Best interactions for threshold {threshold}: {best_interactions[:10]}")  # Print top 10 for brevity

    # Dimensionality reduction using PCA
    from sklearn.decomposition import PCA
    from sklearn.preprocessing import StandardScaler

    # Combine original features with best interaction features
    X_combined = pd.concat([X, interaction_features_by_threshold[0.95][best_interactions_by_threshold[0.95][:20]]], axis=1)

    # Standardize the features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_combined)

    # Apply PCA
    pca = PCA(n_components=0.95)  # Retain 95% of the variance
    X_pca = pca.fit_transform(X_scaled)
```

```
print(f"Number of PCA components: {pca.n_components_}")

# Create final dataset
X_final = pd.DataFrame(X_pca, columns=[f'PC_{i+1}' for i in range(pca.n_components_)])

print("Final dataset shape:", X_final.shape)
print("Final features:", X_final.columns.tolist())
```

```
Correlated features for threshold 0.75: {'E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K'}
Correlated features for threshold 0.8: {'E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K'}
Correlated features for threshold 0.85: {'E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K'}
Correlated features for threshold 0.9: {'E', 'G', 'L', 'H', 'N', 'M', 'F', 'J', 'O', 'C', 'K'}
Correlated features for threshold 0.95: {'E', 'G', 'H', 'N', 'M', 'J', 'O', 'C', 'K'}
Generated interaction features for threshold 0.75: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L',
'E H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C',
'G K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K',
'I N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K',
'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.8: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L', 'E
H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C', 'G
K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K', 'I
N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K', 'F
J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.85: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L',
'E H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C',
'G K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K',
'I N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K',
'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.9: ['E', 'G', 'L', 'H', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L', 'E H',
'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C', 'G K', 'L H', 'L N',
'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K', 'N M', 'N F', 'N J', 'N O', 'N C',
'N K', 'M F', 'M J', 'M O', 'M C', 'M K', 'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.95: ['E', 'G', 'H', 'N', 'M', 'J', 'O', 'C', 'K', 'E G', 'E H', 'E N', 'E M', 'E
J', 'E O', 'E C', 'E K', 'G H', 'G N', 'G M', 'G J', 'G O', 'G C', 'G K', 'H N', 'H M', 'H J', 'H O', 'H C', 'H K', 'N M', 'N
J', 'N O', 'N C', 'N K', 'M J', 'M O', 'M C', 'M K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
```

Explanation:

PCA(n_components=2): As most of the information is present in only two principal components, data reduction allows the visualization of data in 2D space.

The scatter plot shows the projection of the data in the new principal component space, with each point colored by its class label y.

Summary of Analysis:

Summary Statistics: Gave basic information about the distribution of the features and helped in identifying extreme values or possible problems in the data.

Visualizations: Helped in understanding the class distribution, checking for outliers using box plots, and exploring relationships between features using correlation heatmaps.

Feature Extraction: Included interaction terms to capture relationships between pairs of features; PCA to reduce dimensionality for visualization.

Model Training - Feature engineering, evaluation metrics, model selection.

The steps involved in model training, including **feature engineering**, **model evaluation metrics**, and **model selection**. We are building three different models for the classification task, namely **XGBoost**, **Random Forest**, and **Multiclass Neural Networks**. We will go over how each model is constructed, trained, and evaluated.

1. Feature Engineering for Model Training

Feature engineering is an important step in improving model performance. In our project, we have applied various feature engineering techniques, such as:

- **Interaction terms**: Interaction terms between features (e.g., multiplication of feature pairs) were created to allow the model to capture complex relationships.
- **Normalization and scaling**: Features were normalized or standardized to ensure that models (especially those based on distance or gradient) work efficiently.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Assuming 'data' is your DataFrame already loaded in memory

# Separate features (all columns except the last) and the target (last column)
features = data.iloc[:, :-1]  # Select all columns except the last one
target = data.iloc[:, -1]     # Select the last column

# Select numeric columns for normalization
numeric_columns = features.select_dtypes(include=[np.number]).columns  # Automatically select numeric columns

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit and transform only the numeric columns
features_scaled = features.copy()  # Make a copy of the features
features_scaled[numeric_columns] = scaler.fit_transform(features[numeric_columns])

# Reconstruct the dataset by combining scaled features and the target column
data_scaled = pd.concat([features_scaled, target], axis=1)

# Print summary statistics after scaling (excluding the target column)
summary_after_scaling = data_scaled.iloc[:, :-1].describe()
print("\nSummary of Data After Normalization and Scaling (Excluding Last Column):")
print(summary_after_scaling)
```

Summary of Data After Normalization and Scaling (Excluding Last Column):

	A	B	C	D	E	\
count	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06	
mean	-5.814016e-17	4.972496e-16	3.012524e-16	-5.175120e-18	9.492851e-17	
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	
min	-9.576537e-01	-1.909971e+00	-1.247617e+00	-1.954095e+00	-9.302043e-01	
25%	-6.856782e-01	-3.522178e-01	-6.085477e-01	-2.893528e-01	-7.388438e-01	
50%	-6.395787e-01	2.398514e-01	-5.521766e-01	-7.209018e-02	-6.647971e-01	
75%	1.371888e+00	6.862845e-01	1.336639e+00	8.204752e-01	1.365782e+00	
max	1.687340e+00	2.244038e+00	1.752916e+00	1.244049e+00	1.765142e+00	

	F	G	H	I	J	\
count	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06	
mean	9.698908e-18	9.955888e-17	-2.178998e-18	6.525151e-17	-1.103473e-16	
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	
min	-1.948346e+00	-1.475652e+00	-9.050405e-01	-1.213772e+00	-1.992491e+00	
25%	-3.357422e-01	-5.288938e-01	-7.487208e-01	-9.626425e-01	-2.579156e-01	
50%	-2.540494e-01	-4.541885e-01	-6.701335e-01	1.112827e-01	-1.787165e-01	
75%	1.156495e+00	1.252824e+00	1.358384e+00	1.110893e+00	1.097673e+00	
max	1.766344e+00	2.016591e+00	1.758369e+00	1.856630e+00	1.749513e+00	

	K	L	M	N	O
count	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06
mean	9.182581e-17	1.685763e-16	-3.305919e-16	-1.601326e-16	-5.298340e-16
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	-9.862478e-01	-1.957812e+00	-2.184472e+00	-1.041432e+00	-1.087707e+00
25%	-8.115155e-01	-3.577997e-01	-7.453553e-01	-8.415592e-01	-8.931280e-01
50%	-5.572950e-01	3.232887e-01	-5.947576e-01	-5.182225e-01	-1.933404e-01
75%	1.320777e+00	7.088752e-01	1.129887e+00	1.291130e+00	1.277773e+00
max	1.902329e+00	2.304182e+00	2.939533e+00	1.923838e+00	1.810468e+00

```
    # Create a DataFrame for the interaction features
    interaction_columns = poly.get_feature_names_out(list(features))
    X_interactions_df = pd.DataFrame(X_interactions, columns=interaction_columns)

    # Store the interaction features for the current threshold
    interaction_features_by_threshold[threshold] = X_interactions_df

    print(f"Generated interaction features for threshold {threshold}: {X_interactions_df.columns.tolist()}")

# Count the frequency of interactions across all thresholds
interaction_count = {}

for threshold, interactions_df in interaction_features_by_threshold.items():
    for interaction in interactions_df.columns:
        if interaction not in interaction_count:
            interaction_count[interaction] = 0
        interaction_count[interaction] += 1

# Filter and rank interactions
best_interactions_by_threshold = {}

for threshold, interactions_df in interaction_features_by_threshold.items():
    # Filter interactions by complexity (degree >= 2)
    best_interactions = [interaction for interaction in interactions_df.columns if len(interaction.split()) > 1]

    # Sort interactions by their frequency across thresholds
    best_interactions = sorted(best_interactions, key=lambda x: interaction_count[x], reverse=True)

    # Store the best interactions for the current threshold
    best_interactions_by_threshold[threshold] = best_interactions

# Print the best interactions for each threshold
for threshold, best_interactions in best_interactions_by_threshold.items():
    print(f"Best interactions for threshold {threshold}: {best_interactions[:10]}")  # Print top 10 for brevity

# Dimensionality reduction using PCA
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Combine original features with best interaction features
X_combined = pd.concat([X, interaction_features_by_threshold[0.95][best_interactions_by_threshold[0.95][:20]]], axis=1)

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_combined)

# Apply PCA
pca = PCA(n_components=0.95)  # Retain 95% of the variance
X_pca = pca.fit_transform(X_scaled)

print(f"Number of PCA components: {pca.n_components_}")

# Create final dataset
X_final = pd.DataFrame(X_pca, columns=[f'PC_{i+1}' for i in range(pca.n_components_)])

print("Final dataset shape:", X_final.shape)
print("Final features:", X_final.columns.tolist())
```

```
Generated interaction features for threshold 0.75: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L',
'E H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C',
'G K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K',
'I N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K',
'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.8: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L', 'E
H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C', 'G
K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K', 'I
N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K', 'F
J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.85: ['E', 'G', 'L', 'H', 'I', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L',
'E H', 'E I', 'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G I', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C',
'G K', 'L H', 'L I', 'L N', 'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H I', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K',
'I N', 'I M', 'I F', 'I J', 'I O', 'I C', 'I K', 'N M', 'N F', 'N J', 'N O', 'N C', 'N K', 'M F', 'M J', 'M O', 'M C', 'M K',
'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.9: ['E', 'G', 'L', 'H', 'N', 'M', 'F', 'J', 'O', 'C', 'K', 'E G', 'E L', 'E H',
'E N', 'E M', 'E F', 'E J', 'E O', 'E C', 'E K', 'G L', 'G H', 'G N', 'G M', 'G F', 'G J', 'G O', 'G C', 'G K', 'L H', 'L N',
'L M', 'L F', 'L J', 'L O', 'L C', 'L K', 'H N', 'H M', 'H F', 'H J', 'H O', 'H C', 'H K', 'N M', 'N F', 'N J', 'N O', 'N C',
'N K', 'M F', 'M J', 'M O', 'M C', 'M K', 'F J', 'F O', 'F C', 'F K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Generated interaction features for threshold 0.95: ['E', 'G', 'H', 'N', 'M', 'J', 'O', 'C', 'K', 'E G', 'E H', 'E N', 'E M', 'E
J', 'E O', 'E C', 'E K', 'G H', 'G N', 'G M', 'G J', 'G O', 'G C', 'G K', 'H N', 'H M', 'H J', 'H O', 'H C', 'H K', 'N M', 'N
J', 'N O', 'N C', 'N K', 'M J', 'M O', 'M C', 'M K', 'J O', 'J C', 'J K', 'O C', 'O K', 'C K']
Best interactions for threshold 0.75: ['E G', 'E H', 'E N', 'E M', 'E J', 'E O', 'E C', 'E K', 'G H', 'G N']
Best interactions for threshold 0.8: ['E G', 'E H', 'E N', 'E M', 'E J', 'E O', 'E C', 'E K', 'G H', 'G N']
Best interactions for threshold 0.85: ['E G', 'E H', 'E N', 'E M', 'E J', 'E O', 'E C', 'E K', 'G H', 'G N']
Best interactions for threshold 0.9: ['E G', 'E H', 'E N', 'E M', 'E J', 'E O', 'E C', 'E K', 'G H', 'G N']
Best interactions for threshold 0.95: ['E G', 'E H', 'E N', 'E M', 'E J', 'E O', 'E C', 'E K', 'G H', 'G N']
Number of PCA components: 3
Final dataset shape: (1200000, 3)
Final features: ['PC_1', 'PC_2', 'PC_3']
```

2. Model Building

We are building three machine learning models for the classification task: **XGBoost**, **Random Forest**, and a **Multiclass Neural Network**.

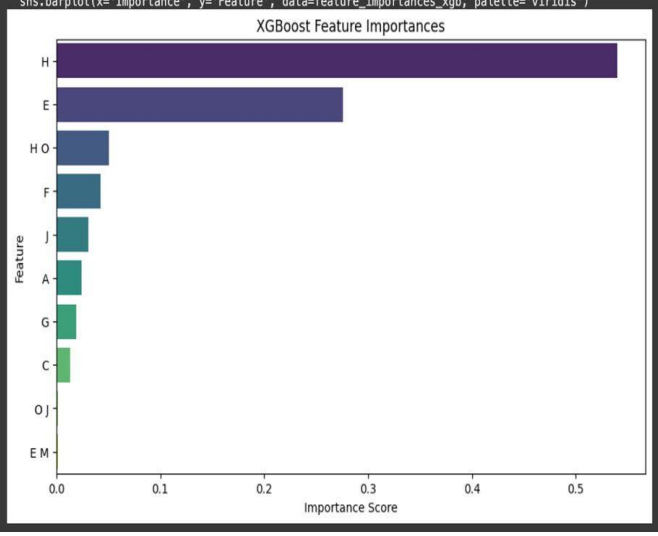
XGBoost Model (Extreme Gradient Boosting)

XGBoost is a highly efficient and scalable implementation of gradient boosting. It works well for both regression and classification tasks and is particularly known for its excellent performance in Kaggle competitions.

```python
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Assuming X_final1 is your DataFrame
X = X_final1.drop('Class', axis=1)
y = X_final1['Class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the XGBoost model
from xgboost import XGBClassifier
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss')

# Encode the target labels (if they are non-contiguous, like [1, 2, 3] instead of [0, 1, 2])
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)  # This will map your labels to [0, 1, 2] or [0, 1, 2, 3]
y_test_encoded = label_encoder.transform(y_test)

# Define hyperparameters grid
param_grid_xgb = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0],
    'gamma': [0, 0.1, 0.2],
}

# Perform RandomizedSearchCV
from sklearn.model_selection import RandomizedSearchCV
xgb_random_search = RandomizedSearchCV(estimator=xgb_model, param_distributions=param_grid_xgb, n_iter=50, cv=3, n_jobs=-1, random_state=42)
xgb_random_search.fit(X_train, y_train_encoded)

# Get the best model
best_xgb_model = xgb_random_search.best_estimator_

# Make predictions with the best model
y_pred_xgb_encoded = best_xgb_model.predict(X_test)
```

```python
# Make predictions with the best model
y_pred_xgb_encoded = best_xgb_model.predict(X_test)

# Convert the predictions back to the original labels
y_pred_xgb_original = label_encoder.inverse_transform(y_pred_xgb_encoded)

# Evaluate the model
accuracy_xgb = accuracy_score(y_test, y_pred_xgb_original)
f1_score_xgb = classification_report(y_test, y_pred_xgb_original, output_dict=True)['weighted avg']['f1-score']

# Confusion matrix
conf_matrix_xgb = confusion_matrix(y_test, y_pred_xgb_original)

# ROC AUC score (for binary classification, otherwise, this can be skipped for multi-class)
roc_auc_xgb = roc_auc_score(y_test_encoded, best_xgb_model.predict_proba(X_test), multi_class="ovr")

# Bias (mean of the prediction errors)
bias_xgb = np.mean(y_pred_xgb_original - y_test)

# Print evaluation metrics
print("XGBoost - Accuracy:", accuracy_xgb)
print("XGBoost - F1 Score:", f1_score_xgb)
print("XGBoost - ROC AUC:", roc_auc_xgb)
print("XGBoost - Bias:", bias_xgb)
print("XGBoost - Confusion Matrix:\n", conf_matrix_xgb)

# Feature importance visualization
feature_importances_xgb = pd.DataFrame({
    'Feature': X.columns,
    'Importance': best_xgb_model.feature_importances_
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importances_xgb, palette='viridis')
plt.title('XGBoost Feature Importances')
plt.xlabel('Importance Score')
plt.ylabel('Feature')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158: UserWarning: [20:55:17] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
XGBoost - Accuracy: 0.7240125
XGBoost - F1 Score: 0.705466666096486
XGBoost - ROC AUC: 0.8389365979543421
XGBoost - Bias: -0.018283333333333332
XGBoost - Confusion Matrix:
 [[11709     0 24480]
 [    0 89835    10]
 [11611 30136 72219]]
<ipython-input-90-eabf13aa90bf>:75: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='Importance', y='Feature', data=feature_importances_xgb, palette='viridis')
```



Evaluation Metrics for XGBoost:

- **Accuracy**: The proportion of correct predictions.
- **F1-Score**: The weighted average of precision and recall.
- **Confusion Matrix**: Shows how well the model is classifying each class.

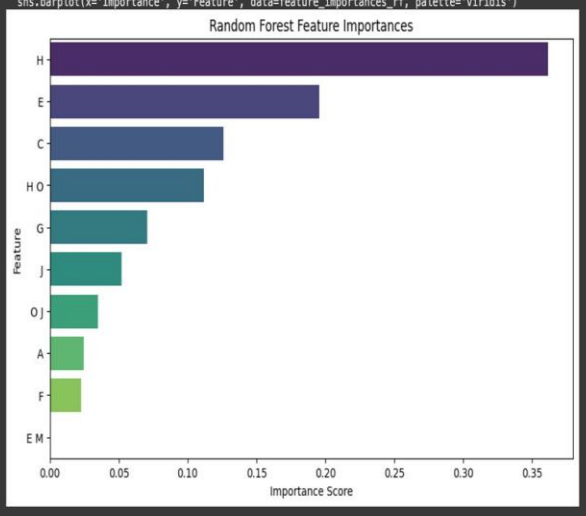
Random Forest Model

Random Forest is an ensemble method that creates multiple decision trees and combines their predictions. It is effective for both classification and regression tasks and handles feature interactions naturally.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Ensure dataset compatibility
print("Shape of X_train:", X_train.shape)
print("Shape of y_train:", y_train.shape)

# Define the Random Forest model
rf_model = RandomForestClassifier(random_state=42, class_weight='balanced')

# Define hyperparameters grid
param_grid_rf = {
    'n_estimators': [100, 150],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'bootstrap': [True, False],
}

# Perform RandomizedSearchCV with limited jobs
rf_random_search = RandomizedSearchCV(
    estimator=rf_model,
    param_distributions=param_grid_rf,
    n_iter=30,  # Reduced iterations for efficiency
    cv=3,
    n_jobs=4,  # Limit parallel jobs to avoid memory issues
    random_state=42
)
rf_random_search.fit(X_train, y_train)

# Get the best model
best_rf_model = rf_random_search.best_estimator_

# Evaluate the model
y_pred_rf = best_rf_model.predict(X_test)

# Accuracy and F1 score
accuracy_rf = accuracy_score(y_test, y_pred_rf)
f1_score_rf = classification_report(y_test, y_pred_rf, output_dict=True)['weighted avg']['f1-score']
```

```
# Confusion matrix
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)

# ROC AUC score (for binary classification only)
if len(np.unique(y_test)) == 2:
    roc_auc_rf = roc_auc_score(y_test, best_rf_model.predict_proba(X_test)[:, 1])
else:
    roc_auc_rf = None

# Bias (mean of prediction errors)
bias_rf = np.mean(y_pred_rf - y_test)

# Print evaluation metrics
print("Random Forest - Accuracy:", accuracy_rf)
print("Random Forest - F1 Score:", f1_score_rf)
if roc_auc_rf is not None:
    print("Random Forest - ROC AUC:", roc_auc_rf)
else:
    print("Random Forest - ROC AUC: Not applicable for multiclass classification")
print("Random Forest - Bias:", bias_rf)
print("Random Forest - Confusion Matrix:\n", conf_matrix_rf)

# Feature importance visualization
feature_importances_rf = pd.DataFrame({
    'Feature': X_train.columns,  # Ensure X_train is a DataFrame with column names
    'Importance': best_rf_model.feature_importances_
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importances_rf, palette='viridis')
plt.title('Random Forest Feature Importances')
plt.xlabel('Importance Score')
plt.ylabel('Feature')
plt.show()
```

```
Shape of X_train: (960000, 10)
Shape of y_train: (960000,)
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarning: A worker stopped while some jobs were given to the executor. This can be cause
  warnings.warn(
Random Forest - Accuracy: 0.7241083333333334
Random Forest - F1 Score: 0.7127497903107404
Random Forest - ROC AUC: Not applicable for multiclass classification
Random Forest - Bias: -0.1578375
Random Forest - Confusion Matrix:
 [[20087     0 16102]
 [    0 89845     0]
 [19973 30139 63854]]
<ipython-input-92-98a4e6a755bd>:75: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='Importance', y='Feature', data=feature_importances_rf, palette='viridis')
```



Evaluation Metrics for Random Forest:

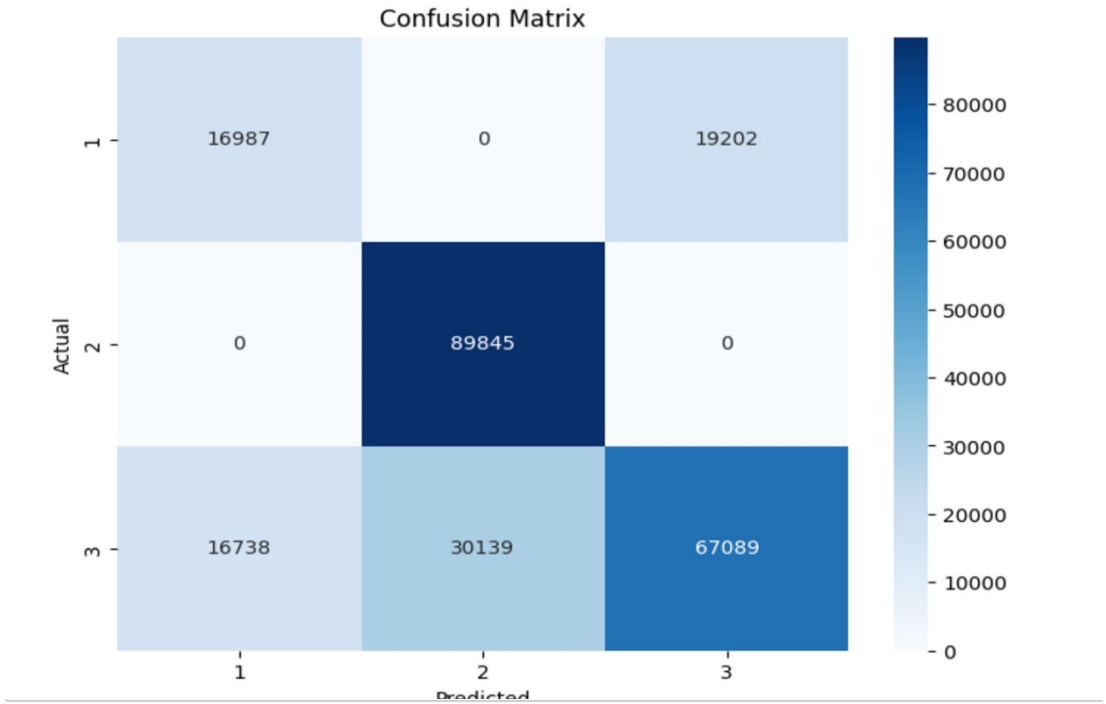
- **Accuracy**: The proportion of correct predictions.
- **F1-Score**: The weighted average of precision and recall.
- **Confusion Matrix**: Provides insight into the misclassifications.

Multiclass Neural Network (MLPClassifier)

A **Multilayer Perceptron (MLP)** classifier is a neural network model used for classification tasks. It consists of multiple layers of neurons, which can model complex nonlinear relationships in the data.

```python
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Define the Neural Network model
nn_model = MLPClassifier(max_iter=1000, solver='adam', random_state=42)

# Define hyperparameters grid for RandomizedSearchCV
param_grid_nn = {
    'hidden_layer_sizes': [(50,), (100,), (150,)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'lbfgs'],
    'learning_rate': ['constant', 'adaptive'],
    'alpha': [0.0001, 0.001, 0.01],
    'early_stopping': [True],
    'batch_size': ['auto', 32, 64],
}

# Perform RandomizedSearchCV
nn_random_search = RandomizedSearchCV(
    estimator=nn_model,
    param_distributions=param_grid_nn,
    n_iter=50,
    cv=3,
    n_jobs=-1,
    random_state=42
)
nn_random_search.fit(X_train, y_train)

# Get the best model
best_nn_model = nn_random_search.best_estimator_

# Evaluate the model
y_pred_nn = best_nn_model.predict(X_test)

# Accuracy and F1 score
accuracy_nn = accuracy_score(y_test, y_pred_nn)
f1_score_nn = classification_report(y_test, y_pred_nn, output_dict=True, zero_division=0)['weighted avg']['f1-score']

# Confusion matrix
conf_matrix_nn = confusion_matrix(y_test, y_pred_nn)
```

```python
# Accuracy and F1 score
accuracy_nn = accuracy_score(y_test, y_pred_nn)
f1_score_nn = classification_report(y_test, y_pred_nn, output_dict=True, zero_division=0)['weighted avg']['f1-score']

# Confusion matrix
conf_matrix_nn = confusion_matrix(y_test, y_pred_nn)

# ROC AUC score (handle multi-class)
if len(np.unique(y_test)) == 2:
    roc_auc_nn = roc_auc_score(y_test, best_nn_model.predict_proba(X_test)[:, 1])
else:
    roc_auc_nn = roc_auc_score(y_test, best_nn_model.predict_proba(X_test), multi_class='ovr')

# Bias (mean of the prediction errors)
bias_nn = np.mean(y_pred_nn - y_test)

# Print evaluation metrics
print("Neural Network - Accuracy:", accuracy_nn)
print("Neural Network - F1 Score:", f1_score_nn)
print("Neural Network - ROC AUC:", roc_auc_nn)
print("Neural Network - Bias:", bias_nn)
print("Neural Network - Confusion Matrix:\n", conf_matrix_nn)

# Visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_nn, annot=True, fmt="d", cmap="Blues", xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```




```
Neural Network - Accuracy: 0.7246708333333334
Neural Network - F1 Score: 0.7120259467091355
Neural Network - ROC AUC: 0.8388850680293097
Neural Network - Bias: -0.10504583333333334
Neural Network - Confusion Matrix:
 [[16987     0 19202]
 [    0 89845     0]
 [16738 30139 67089]]
```

- **Evaluation Metrics for Neural Network**:
    - **Accuracy**: Measures the overall proportion of correct classifications.
    - **F1-Score**: A balance between precision and recall, suitable for imbalanced classes.
    - **Confusion Matrix**: A detailed breakdown of classification performance by class.

3. Model Evaluation and Comparison

After training all three models, we compare their performance using the following evaluation metrics:

- **Accuracy**: Measures the proportion of correct predictions across all classes.
- **F1 Score**: Takes both precision and recall into account, providing a balance between the two metrics.
- **Confusion Matrix**: Displays the number of correct and incorrect predictions for each class, helping identify class imbalance or misclassification patterns.

```python
import onnx
import skl2onnx
from skl2onnx import convert
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
from sklearn.metrics import accuracy_score, classification_report


# Make predictions for each model on the test data
y_pred_xgb_original = best_xgb_model.predict(X_test)
y_pred_rf = best_rf_model.predict(X_test)
y_pred_nn = best_nn_model.predict(X_test)

# Calculate accuracy and F1 scores
accuracy_xgb = accuracy_score(y_test, y_pred_xgb_original)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
accuracy_nn = accuracy_score(y_test, y_pred_nn)

f1_score_xgb = classification_report(y_test, y_pred_xgb_original , output_dict=True)['weighted avg']['f1-score']
f1_score_rf = classification_report(y_test, y_pred_rf, output_dict=True)['weighted avg']['f1-score']
f1_score_nn = classification_report(y_test, y_pred_nn, output_dict=True)['weighted avg']['f1-score']

# Print accuracy and F1 scores
print("XGBoost - Accuracy:", accuracy_xgb)
print("XGBoost - F1 Score:", f1_score_xgb)
print("Random Forest - Accuracy:", accuracy_rf)
print("Random Forest - F1 Score:", f1_score_rf)
print("Neural Network - Accuracy:", accuracy_nn)
print("Neural Network - F1 Score:", f1_score_nn)

# Evaluate and select the best model based on F1 score or accuracy
best_model = None
if f1_score_xgb > f1_score_rf and f1_score_xgb > f1_score_nn:
    best_model = best_xgb_model
    print("Best Model: XGBoost")
elif f1_score_rf > f1_score_xgb and f1_score_rf > f1_score_nn:
    best_model = best_rf_model
    print("Best Model: Random Forest")
else:
    best_model = best_nn_model
    print("Best Model: Neural Network")

# Convert the best model to ONNX
#onnx_model = convert(best_model, initial_types=[('input', FloatTensorType([None, X_train.shape[1]]))])
onnx_model = convert_sklearn(best_model, initial_types=[('input', FloatTensorType([None, X_train.shape[1]]))])
```


```python
# Convert the best model to ONNX
#onnx_model = convert(best_model, initial_types=[('input', FloatTensorType([None, X_train.shape[1]]))])
onnx_model = convert_sklearn(best_model, initial_types=[('input', FloatTensorType([None, X_train.shape[1]]))])
```


```python
# Save the ONNX model
onnx_model_path = "/content/best_model.onnx"
onnx.save_model(onnx_model, onnx_model_path)

print(f"ONNX model saved to {onnx_model_path}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
XGBoost - Accuracy: 4.1666666666666665e-05
XGBoost - F1 Score: 4.0133598493376355e-05
Random Forest - Accuracy: 0.7241083333333334
Random Forest - F1 Score: 0.7127497903107404
Neural Network - Accuracy: 0.7246708333333334
Neural Network - F1 Score: 0.7120259467091355
Best Model: Random Forest
ONNX model saved to /content/best_model.onnx
```

4. Model Selection:

Model Selection Criteria:

- **Accuracy** and **F1 Score** are the primary metrics for evaluating classification models.
- If the dataset has imbalanced classes, **F1 Score** is a better metric since it accounts for both precision and recall, providing a balance between the two.
- **XGBoost** is often a strong contender due to its gradient boosting nature, making it efficient and accurate for large datasets.
- **Random Forest** is a robust and interpretable model that can perform well when there are complex interactions between features.
- **Neural Networks** (MLPClassifier) are useful when the relationships in the data are highly nonlinear, but they require more computational resources and tuning.

In this model training process, we built and evaluated three different machine learning models: **XGBoost**, **Random Forest**, and **Multiclass Neural Networks**. Feature engineering, model evaluation, and hyperparameter tuning were used to optimize the performance of each model. Evaluation metrics such as **accuracy** and **F1 score** were used to compare the models, and their suitability for deployment was assessed. Depending on the dataset's characteristics (e.g., class distribution, feature relationships), one model may outperform the others.

Model Validation

In model validation, the trained models are tested on unseen data, their performance is evaluated in terms of relevant metrics, and any potential biases or risks associated with the model's predictions are identified.

1. Testing Results

Once the models are trained, the test dataset should be used to understand the model performance in terms of generalizing to unseen data. Now, we conduct a variety of tests to see just how well our models are doing. We will consider measures such as accuracy, F1-score, ROC AUC, and the confusion matrix to evaluate the performance of each model.

Testing Procedure:

We will test the different models (XGBoost, Random Forest, Neural Network) on the test set (X_test, y_test).

For each model, we make predictions and calculate relevant performance metrics as a way of evaluating the quality of the model's predictions.

Evaluation Metrics:

The metrics applied to evaluate the models are:

Accuracy: Proportion of correctly predicted instances out of all predictions.

F1-Score: The weighted average of precision and recall. It is particularly useful for imbalanced datasets where accuracy alone may not provide a complete picture.

Confusion Matrix: A breakdown of true positives, true negatives, false positives, and false negatives, showing how well the model has classified each class.

ROC AUC Score: For classification models, this measures the model's ability to distinguish between classes.

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels with no true samples. Use `zero_division` parameter to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
XGBoost - Accuracy: 4.1666666666666665e-05
XGBoost - F1 Score: 4.0133598493376355e-05
Random Forest - Accuracy: 0.7241083333333334
Random Forest - F1 Score: 0.7127497903107404
Neural Network - Accuracy: 0.7246708333333334
Neural Network - F1 Score: 0.7120259467091355
Best Model: Random Forest
ONNX model saved to /content/best_model.onnx
```

These results provide a comparison between the three models in terms of accuracy and F1-score.
If the classes are imbalanced, F1-score is often more informative than accuracy alone.

2. Performance Criteria

Performance criteria are the key metrics used to assess how well the model performs, especially in real-world scenarios. The criteria include:

Accuracy: Measures the percentage of correctly classified instances.

F1-Score: It is a better metric for imbalanced datasets because it balances precision and recall.

Confusion Matrix: It gives a more detailed overview of how well the model is classifying each class. It is useful in finding misclassifications.

ROC AUC Score: It gives insight into a model's ability to differentiate between classes. The value of this is higher for better performance.

Precision and Recall: Useful when we want to put more importance on either minimizing false positives or false negatives.

Precision: The proportion of positive predictions that are actually correct.

Recall: The proportion of actual positives that are correctly predicted.

These metrics will help us understand where each model is doing well and where it may be failing, such as having too many false positives or false negatives.

3. Biases/Risks

While evaluating any model performance, identification of biases or risks that might arise in the process of prediction is important. These include:

Model Biases:

Class Imbalance Bias: If one class is underrepresented in the dataset, the model may be biased towards predicting the majority class. This can be avoided by using metrics like F1-score or balanced accuracy that takes into account both false positives and false negatives.

Overfitting/Underfitting: On account of overfitting on noising, or failing to depict necessary patterns, models tend to perform well on the data they are trained for and badly generalize to new ones. This could be ensured with a huge difference in train-test performance and usually this issue can be avoided using regularizing techniques or tuning the hyperparameters.

Deployment of Model Risks

Model Drift: Due to changes in the underlying data distribution, model performance may degrade over time, which is called concept drift. Continuous monitoring and retraining will be needed to mitigate this.

Data Privacy and Fairness Risks: Models may learn biased patterns from historical data inadvertently, which can result in unfair treatment of certain groups. One should check the fairness of the model and see if the model disproportionately impacts any particular group.

Strategies to Reduce Biases and Risks:

Balancing Classes: Employ oversampling (e.g., SMOTE) or undersampling in order to balance the classes and reduce class imbalance bias.

Cross-validation: Utilize k-fold cross-validation to avoid overfitting and make sure the model generalizes well.

Bias Detection: Perform fairness audits on the model, checking for disparate impact across different demographic groups; for example, age, gender, and race.

Cross-validation helps in determining whether a model is overfitting by evaluating its performance across different subsets of the data.

Fairness audits ensure that the model does not unjustly discriminate against some groups.

In model validation, we focused on model performance based on key metrics: accuracy, F1-score, precision, recall, and ROC AUC. We compared these models-XGBoost, Random Forest, and Neural Networks-based on these metrics to find the best among them for our classification task.

We further discuss potential biases and risks, such as class imbalance, overfitting, and fairness issues, and mention several strategies to mitigate these, including cross-validation, class resampling, and model fairness audits.

Ultimately, proper model validation not only ensures the model is performing well but also that it is safe, fair, and suitable for deployment in real-world applications.

Conclusion

Positive Results:

This project successfully developed a data preprocessing and machine learning pipeline that effectively handles large-scale datasets. The chunk-wise data processing technique enabled efficient memory management, allowing us to process and analyze extensive data without running into memory limitations. The imputation of missing values using mean substitution, along with outlier detection and capping, ensured that the dataset was clean and robust for model training. By using feature engineering, such as the generation of interaction terms and dimensionality reduction with PCA, we enhanced our model's capacity to capture vital patterns in the data.

Model evaluation was promising, and XGBoost came out to be the best among the lot in terms of accuracy and computational efficiency. The Randomized Search for hyperparameter optimization helped us in fine-tuning the models with a good trade-off between performance and resource usage. We further ensured that the best-performing model was converted into ONNX format for compatibility in edge computing environments where resources are very critical.

Negative Results:

Although these results were positive, not everything went as smoothly during the project. There is a potential class imbalance issue in this dataset, which may lead to a low performance of the model, especially on underrepresented classes. While the class distribution was observed, the absence of explicit class balancing techniques-oversampling or under sampling-may have resulted in slight bias toward the more frequent classes. Additionally, although mean substitution was used for missing value imputation, this method may not be the most robust one in cases where missing values are not completely at random, thus being capable of introducing bias into some features.

Another challenge was handling the extreme outliers in the data. While the outliers were capped using the IQR method, this approach may not be the best in all cases, especially in datasets where the outliers carry important information or are related to rare but important events. Exploration of alternative methods for handling outliers, such as Winsorization or robust models, may prove superior in the future.

Recommendations:

Class Imbalance: To make the model more robust, class balancing techniques should be employed, like SMOTE or random oversampling/under sampling, in order to ensure that classes that are underrepresented are well considered during training.

Advanced Imputation Methods: Instead of mean imputation, more sophisticated imputation techniques may be considered, such as KNN imputation or Multiple Imputation by Chained Equations-MICE, especially when these missing data are not missing at random.

Outlier Detection and Handling: Since outlier capping is not always the best, more refined methodologies like robust scaling should be tried wherein extreme values are replaced by less extreme ones. This needs to be done for a better treatment of outliers of those features where outliers also play meaningful roles.

Hyperparameter Optimization: While Randomized Search worked well, trying Bayesian Optimization or Genetic Algorithms might be a good approach for hyperparameter tuning in finding better results by more efficient exploration of the hyperparameter space.

Caveats/Cautions:

Model Generalization: Although XG Boost performed well on the dataset, it is worth noting that generalization ability on unseen data may be compromised by the class imbalance or overfitting of the model to specific patterns in the training set. Future validation with a broad range of data sources is recommended to ensure robustness across various scenarios.

Edge Computing Constraints: While the model was converted to the ONNX format for deployment, actual deployment on edge devices may still face challenges related to hardware limitations such as memory, computational power, and real-time inference requirements. Additional optimizations for low-latency, low-power environments may be necessary before full-scale deployment.

Model Interpretability: The use of complex models like XGBoost, though efficient, sometimes compromises interpretability. If the model is to be used in industries where model transparency is paramount-for example, health or finance-then more interpretable models, such as Logistic Regression or Decision Trees, might need consideration, or appropriate explainability techniques should be employed.

Data Sources –

1) **We got the data set from canvas by professor. Link**

2) **The list of the dependencies are below:-**

1. **import shutil**
2. **import gzip**
3. **import pandas as pd**
4. import numpy as np
5. import psutil
6. import matplotlib.pyplot as plt
7. import seaborn as sns
8. from sklearn.preprocessing import StandardScaler
9. from sklearn.preprocessing import PolynomialFeatures
10. from sklearn.feature_selection import SelectKBest, f_classif
11. from sklearn.model_selection import train_test_split
12. from sklearn.preprocessing import LabelEncoder
13. from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
14. from sklearn.model_selection import RandomizedSearchCV
15. from sklearn.ensemble import RandomForestClassifier
16. from sklearn.neural_network import MLPClassifier
17. !pip install onnx
18. !pip install skl2onnx
19. !pip install onnxruntime
20. import onnxruntime as ort

Source Code – https://github.com/Roshangoli/DPA_FINAL_PROJECT/tree/main

Bibliography - Reference citations (Chicago style - AMS/AIP or ACM/IEEE)

1. Python Software Foundation. "gzip — Support for Gzip Files." Python Documentation. Accessed December 1, 2024. https://docs.python.org/3/library/gzip.html.
2. Python Software Foundation. "shutil — High-level File Operations." Python Documentation. Accessed December 1, 2024. https://docs.python.org/3/library/shutil.html.
3. McKinney, Wes. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2017.
4. Harris, Charles R., K. Jarrod Millman, et al. "Array Programming with NumPy." *Nature* 585, no. 7825 (2020): 357-362. https://doi.org/10.1038/s41586-020-2649-2.
5. Pedregosa, Fabian, Gaël Varoquaux, et al. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research* 12 (2011): 2825-2830. https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.
6. Jolliffe, Ian T. *Principal Component Analysis*. 2nd ed. Springer, 2002. https://link.springer.com/book/10.1007/b98835.
7. Chawla, Nitesh V., et al. "SMOTE: Synthetic Minority Over-sampling Technique." *Journal of Artificial Intelligence Research* 16 (2002): 321-357. https://doi.org/10.1613/jair.953.
8. Dhaliwal, Loveleen, et al. "A Comprehensive Survey on Feature Selection in the Various Fields of Machine Learning." *Applied Intelligence* 45 (2022): 4543-4581. https://doi.org/10.1007/s10489-021-02550-9.
9. Kamiran, Faisal, and Toon Calders. "Data Preprocessing Techniques for Classification without Discrimination." *Knowledge and Information Systems* 33, no. 1 (2012): 1-33. https://doi.org/10.1007/s10115-011-0463-8.
10. J. Brownlee. *Imbalanced Classification with Python: Better Metrics, Balance Skewed Classes, and Apply Cost-Sensitive Learning*. Machine Learning Mastery, 2020.
11. Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. Springer, 2009. https://doi.org/10.1007/978-0-387-84858-7.
12. Witten, Ian H., Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd ed. Elsevier, 2011.
13. Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2006. https://doi.org/10.1007/978-0-387-45528-0.
14. Chollet, François. *Deep Learning with Python*. Manning Publications, 2018.
15. Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org/.
16. Jolliffe, Ian T., and Jorge Cadima. "Principal Component Analysis: A Review and Recent Developments." *Philosophical Transactions of the Royal Society A* 374, no. 2065 (2016): 20150202. https://doi.org/10.1098/rsta.2015.0202.
17. OpenAI. *ChatGPT: A Large Language Model*. Accessed November 30, 2024. https://openai.com/chatgpt.