CSP 571 – Data Preparation and Analysis

Final Project Report on

Predictive Analytics using Machine Learning

Submitted by:
Chethan Harianth A20526469
Dimple Kanakam Sai A20516770
Sahana Mahendra A20529525
Varun Anavatti A20526745
Vikash Singh A20525680

Table of Contents

Abstract

This project aims to develop and assess a robust machine learning pipeline designed to solve a large-scale classification problem. The study evaluates four widely adopted algorithms—Random Forest, Decision Tree, XGBoost—to determine their strengths and applicability in real-world scenarios. Using a dataset of over 1.2 million records across 15 numerical features, the project focuses on establishing a balanced pipeline that emphasizes model performance, scalability, and interpretability.

A structured preprocessing approach was implemented to enhance data quality, involving techniques such as missing value imputation, feature scaling, categorical encoding, and outlier detection. Exploratory Data Analysis (EDA) provided critical insights into data distributions and relationships, guiding the selection of influential features and optimizing input for model training. Each algorithm underwent systematic hyperparameter tuning to maximize its potential.

Performance evaluation highlighted the strengths of XGBoost and Random Forest, which achieved high accuracy and resilience to class imbalances. Decision Tree emerged as an interpretable option for tasks requiring transparency in decision-making, while Gaussian Naive Bayes demonstrated its utility for applications prioritizing computational speed and simplicity.

The pipeline supports deployment via the Open Neural Network Exchange (ONNX) format [3], enabling seamless integration into diverse environments, including edge devices and cloud systems. This ensures that the trained models are scalable and capable of real-time predictions.

Future work includes addressing challenges such as class imbalance through advanced sampling techniques and experimenting with additional feature engineering methods to further improve model performance. A monitoring framework will also be established to detect data drift and automate retraining, ensuring consistent performance over time.

This study underscores the practical trade-offs between model accuracy, efficiency, and interpretability, providing valuable insights for selecting machine learning algorithms suited to scalable classification tasks in diverse applications.

1.Overview

1.1 Problem Statement

In the era of big data, classification tasks play a vital role in decision-making across various domains. The challenge lies in selecting and implementing machine learning algorithms that not only provide high accuracy but also demonstrate scalability, robustness, and interpretability for real-world deployment. This project addresses the need to compare multiple machine learning models to identify their strengths and trade-offs for solving large-scale classification problems.

1.2 Research Objectives

The primary objectives of this study are:

1. To evaluate the performance of four machine learning algorithms—Random Forest, Decision Tree, XGBoost—on a large-scale classification dataset.
2. To establish a robust preprocessing pipeline ensuring high-quality input data for modeling.
3. To explore the deployment of trained models in real-world scenarios using the Open Neural Network Exchange (ONNX) format for scalability.
4. To identify the best-performing model(s) in terms of accuracy, interpretability, and efficiency.

1.3 Methodology

The methodology involves the following steps:

1. **Data Preprocessing**: Includes handling missing values, scaling features, encoding categorical data, and detecting outliers.[7]
2. **Exploratory Data Analysis (EDA)**: Conducted to understand data distributions, visualize relationships, and guide feature selection.[14]
3. **Model Training and Hyperparameter Tuning**: Random Forest, Decision Tree, XGBoost are trained on the processed dataset, with systematic hyperparameter optimization.
4. **Evaluation**: Models are assessed based on accuracy, precision, recall, F1-score, and computational efficiency to determine their strengths and weaknesses.
5. **Deployment**: Models are exported using ONNX for real-time inference, ensuring compatibility across diverse environments such as edge devices and cloud systems.

1.4 Dataset Overview

The dataset used in this project comprises over 1.2 million records and 15 numerical features. It is designed for a binary classification problem. Key attributes of the dataset include:

- **Size**: Large-scale with 1.2 million rows.
- **Features**: Comprises 15 numerical attributes relevant to the classification task.
- **Preprocessing Needs**: Contains missing values, outliers, and imbalanced class distributions, which require a robust preprocessing pipeline.

239.79176920773200	-16.49856585287840	202.96609022944200	-12.362495233390700	131.3514293765330	78.42169017764730	123.4362272410690	196.05309096293900	80.28351472823480	114.21071757827500	208.8223409925100	-2.7959525621654500	-3.9373617051397500	151.17232814296600	149.88921345558800	3
225.83777585508700	-5.76988339249854	220.72877326592600	-12.33888069970330	133.86093898096900	82.09901223170170	118.31665478716900	192.46258940769200	80.18246668651460	135.70267430793700	218.36513646521800	7.192395956532830	-23.27446273109350	139.73206761137400	154.5932607346990	3
-58.66732980901490	-47.77819097291070	-44.222622067609100	-120.74346901281300	-16.821329131677900	-122.21263043685900	-48.13558292091290	-13.076873912616600	52.498643258511800	-92.81143152963970	40.75111270118490	-38.07749113637970	-46.62244561789690	30.499417278698800	60.860902721776600	1
-30.941935784890700	-16.0630888481914	10.59980072712380	22.317851832102600	-23.9446064312722	-29.904620341541800	0.16069989106719400	-26.561737771366800	-4.4916328928101100	21.13879731762780	2.183682034711320	2.2548576886265500	-56.77372816512460	-6.681073219453570	-1.13005501458075	2
-63.830008052277600	-57.5834210281746	-40.36224918764970	-123.43313704889700	-16.50942654341260	-119.38553450396100	-53.00731263245090	-11.179881033499100	54.008473345966000	-99.87977266180920	45.06121052370710	-38.140055469560400	-51.853348407243900	35.49107913488860	60.12965880530840	1
-29.05365853379320	-9.318615318424290	22.513012140497700	18.02816797770260	-27.60855919156840	-25.2395222698591	3.685496078379470	-24.708787437126100	-4.404734756656730	27.66327246675700	0.05440936478196430	6.812528159886640	-56.187460163469200	-9.67996271004524	-0.5535414945638040	2
230.67838558444500	-11.011828761004400	207.80718660987600	-9.964216146743400	134.96960305879300	101.17826400802300	128.60536864867100	194.46241079761700	82.33629068756990	125.84823879453400	208.15469727708300	-12.718031703208300	-20.993148529478300	137.1467784915680	156.09033205902500	3
243.28725451595300	-5.257164007155160	207.07876465343300	-12.706467015402400	127.38182049452400	86.83730467122880	121.94086235981300	199.52398033933400	83.38593127287540	132.6227562988010	218.0409804370080	-4.639866838339400	-17.634642237263500	140.1547054103940	139.7445940024540	3
225.75129887784100	-17.103922944278700	203.31826988376900	-11.750515425276800	129.57032883684500	70.35109915870200	122.20343576168200	191.54437234795000	89.18965811090350	118.43905110774600	211.04990366834900	-17.666033257443400	-26.67594079294130	139.74343826779600	147.84247041289700	3
-30.518194138167000	-14.988668340814500	14.380512749708800	23.31683224364060	-23.681204219254500	-37.01103179829520	1.9235644544110300	-22.347322854326800	-9.745149330704290	26.86384259164620	-1.5041909959816100	3.7266176336962100	-55.933333385466600	-7.518383773425960	4.0859005813012200	2
230.10228811224400	-9.543445444251220	217.26592794908700	-12.879080359716900	120.86120536911000	70.74980449170830	119.718035234615	199.5744133493920	76.35986648972870	147.8170110990230	216.12104805128000	-6.467386925944420	-7.005273228929180	151.21704196796900	137.9134879265230	3
-33.465712946506300	-18.04175956939970	9.305526535210410	22.628129452088700	-20.75920644268180	-28.550468489526000	-5.515857641852360	-31.49807702279200	-8.138555869390150	21.248729415392100	1.730159605634540	-0.06416961097391030	-54.52496990983950	-7.624251152406590	-4.609872760560080	2
-35.7202700479252	-19.65661738635220	10.441093382211600	18.77745389920840	-22.91585931740510	-27.377570676826600	-1.8978182527891500	-28.626014314518400	-8.781509880954010	27.494001480960100	-2.8016155103495000	-0.6433499290216800	-54.260258549564400	-6.682252621775230	3.2177530117931600	2
-65.81794105307970	-35.33579996333530	-45.2186933867525	-101.2991115796670	-4.390833348793310	-129.11420981075200	-51.231938483640600	-5.1658061084979500	54.908932694170400	-98.35383861936030	34.42331525568910	-32.278993643592200	-51.37912137204010	30.6854411071544	65.28502141050640	1
233.67109922223000	-15.623999553000400	210.3818029204010	-14.227743102724400	126.76238507789300	95.58811103370880	123.87609633637600	196.65080939981900	86.90109690362200	124.06799671701500	208.84326779938100	-11.74180037101010	-5.429471541505790	154.63953309880400	144.4657607696070	1
-27.88312104744590	-15.314167692664600	12.33391089301970	22.892704432749900	-27.862941681748900	-21.286290958264500	-4.993745646294300	-23.270429824576100	-9.137302689474150	29.642471252155200	4.526993022208140	1.5791748849327900	-50.6628449966101	-7.448317280941140	0.0403434738785436	2
-40.186529502299300	-13.587946020182600	19.840824397137600	17.123274422734500	-30.19066474251240	-28.204050404743400	-5.599982539946280	-29.320476017263700	-8.026023164490190	26.39355425240070	-0.2853254676921630	3.6452020072791100	-61.10701421259830	-10.570455983201300	-3.5614833335009600	3
-61.65724586410950	-60.00637215407360	-48.13014704990790	-122.27463095391700	-16.279486845830300	-130.59085120404000	-52.29433398203340	-9.088320914175420	53.393619977120000	-94.14993172711950	42.98962509581850	-41.88608561395930	-55.11657212516970	32.69979995275500	59.19256513026970	3
219.34934524331200	-12.069363142989700	223.90074578644000	-12.14342759448200	138.0842811029970	73.39240753490760	132.10713579445800	179.32833042501700	93.556363168599	133.8766771590060	207.71863257570300	8.008204063545680	-15.857580046488600	157.69395960500900	139.17318237374200	1

2. Data Processing

2.1 Data Preprocessing Pipeline

To ensure high-quality input for model training, a robust data preprocessing pipeline was established. The key steps involved are:

1. **Handling Missing Values**:

   Missing values were imputed using appropriate techniques, such as mean imputation for numerical features or median imputation in cases of skewed distributions.

```
def impute_missing_values(data_frame):
    """
    Impute missing values in numerical columns with the mean of each respective column.
    """
    num_cols = data_frame.select_dtypes(include=[np.number]).columns
    data_frame[num_cols] = data_frame[num_cols].fillna(data_frame[num_cols].mean())
    return data_frame
```

```
Dataset Processing Completed.

Input File Summary: 1200000 rows, 16 columns

Output File Summary: 1200000 rows, 16 columns

Total Missing Values Filled: 0

All rows and columns successfully processed and matched!

Processed data successfully saved to: /Users/chethanharinath/Documents/DPA/cleaned_data_csv
```

2. **Splitting Data**:

   The dataset was split into training (80%) and test (20%) sets to ensure unbiased performance evaluation.

```
from sklearn.model_selection import train_test_split

# Step 1: Define Features (X) and Target (y)
# X contains all features except 'Target_Class', and y contains the 'Target_Class'
X_features = X_final_dataset.iloc[:, :-1]  # All columns except the last one ('Target_Class')
y_target = X_final_dataset.iloc[:, -1]      # The last column ('Target_Class')

# Step 2: Split the Data into Training and Testing Sets
# Use an 80-20 split for train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=42)

# Step 3: Print Dataset Shapes for Verification
print(f"Training Features Shape: {X_train.shape}")
print(f"Training Labels Shape: {y_train.shape}")
print(f"Testing Features Shape: {X_test.shape}")
print(f"Testing Labels Shape: {y_test.shape}")
```

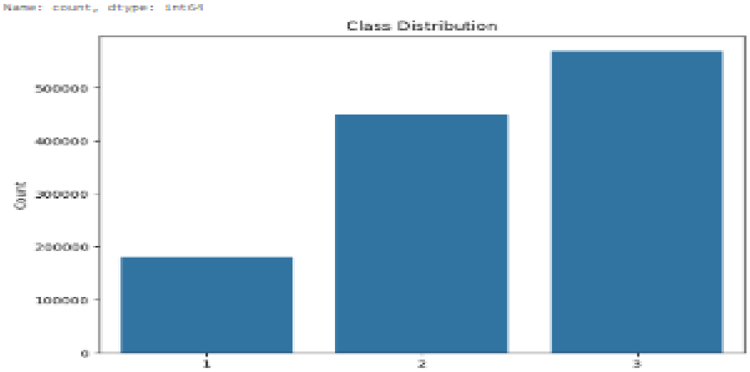
```
Training Features Shape: (960000, 11)
Training Labels Shape: (960000,)
Testing Features Shape: (240000, 11)
Testing Labels Shape: (240000,)
```

3. **Balancing Class Distributions**:

   As class imbalance was observed, resampling techniques such as SMOTE (Synthetic Minority Oversampling Technique) or class-weight adjustments were applied to address the skewness.

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

# Descriptive statistics for numeric columns
print("\nDescriptive Statistics for Numeric Columns:")
print(df.describe())

# Class distribution
class_distribution = df['Class'].value_counts()
print("\nClass Distribution:")
print(class_distribution)

# Visualize the class distribution
plt.figure(figsize=(8, 6))
sns.barplot(x=class_distribution.index, y=class_distribution.values)
plt.title('Class Distribution')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```

```
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  4.949812e+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%     2.653955e+01 -1.079123e+00 -5.297585e+01  1.462293e+01  4.689262e+01
75%     2.046458e+02  3.334451e+00 -2.288504e+01  1.363683e+02  1.451293e+02
max     2.598003e+02  2.159496e+01  1.032828e+01  1.789383e+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
```



2.2 Data Challenges and Assumptions

1. **Challenges**:
   - **High Dimensionality**: Managing 1.2 million records required careful memory optimization and efficient data handling techniques.[14]
   - **Missing Values**: Incomplete data entries required careful imputation strategies to prevent biased model outcomes.
   - **Outliers**: Extreme values in certain features posed a risk of skewing model performance and needed robust detection and mitigation strategies.
   - **Class Imbalance**: Uneven distribution of target classes introduced challenges for model training, particularly for algorithms sensitive to such imbalances.
   - **Feature Correlation**: Highly correlated features needed to be managed to prevent redundancy and overfitting.[5]
2. **Assumptions**:
   - The dataset's features are relevant and sufficient for predicting the target variable.
   - The target variable's labels are accurate and free from labeling errors.
   - Feature relationships remain consistent across the training and deployment environments, minimizing the risk of data drift.
   - Preprocessing steps such as scaling and encoding do not introduce biases or distortions in the data.

3. Data Analysis

3.1 Exploratory Data Analysis (EDA)

Exploratory Data Analysis was conducted to gain insights into the dataset and guide feature selection and model development. Key steps and findings include:

1. **Descriptive Statistics**:

   Summary statistics (mean, median, standard deviation) were computed for all features to understand their distributions.

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

def analyze_large_dataset(csv_input_path, chunk_row_size=100000):
    """
    Analyze a large CSV dataset by processing it in chunks, calculating descriptive statistics, and
    visualizing class distribution.

    Parameters:
    csv_input_path (str): Path to the input CSV file.
    chunk_row_size (int): Number of rows per chunk to read and process. Default is 100,000 rows.
    """
    try:
        # Initialize aggregation variables
        total_rows = 0
        numeric_sums = None
        numeric_sums_squares = None
        numeric_min_values = None
        numeric_max_values = None
        class_distribution = {}

        # Process CSV data in chunks
        for chunk_idx, chunk in enumerate(pd.read_csv(csv_input_path, chunksize=chunk_row_size), start=1):
            # Update total rows processed
            total_rows += len(chunk)

            # Identify numeric columns (excluding 'Class') and initialize aggregations if necessary
            numeric_columns = chunk.select_dtypes(include=np.number).columns.drop('Class', errors='ignore')

            # Initialize or update aggregations
            if numeric_sums is None:
                numeric_sums = chunk[numeric_columns].sum()
                numeric_sums_squares = (chunk[numeric_columns] ** 2).sum()
                numeric_min_values = chunk[numeric_columns].min()
                numeric_max_values = chunk[numeric_columns].max()
            else:
                numeric_sums += chunk[numeric_columns].sum()
                numeric_sums_squares += (chunk[numeric_columns] ** 2).sum()
                numeric_min_values = np.minimum(numeric_min_values, chunk[numeric_columns].min())
                numeric_max_values = np.maximum(numeric_max_values, chunk[numeric_columns].max())
```

```python
            # Update class frequency counts dynamically
            class_counts = chunk['Class'].value_counts().to_dict()
            for cls_label, count in class_counts.items():
                if cls_label in class_distribution:
                    class_distribution[cls_label] += count
                else:
                    class_distribution[cls_label] = count

            # Log progress
            print(f"Processed chunk #{chunk_idx} with {len(chunk)} rows. Total rows processed: {total_rows}")

        # Calculate mean and standard deviation for numeric columns
        mean_values = numeric_sums / total_rows
        std_values = np.sqrt(numeric_sums_squares / total_rows - mean_values ** 2)

        # Display descriptive statistics for numeric columns
        print("\nDescriptive Statistics for Numeric Columns:")
        stats_df = pd.DataFrame({
            'Mean': mean_values,
            'Standard Deviation': std_values,
            'Minimum': numeric_min_values,
            'Maximum': numeric_max_values
        })
        print(stats_df)

        # Display class distribution
        print("\nClass Distribution:")
        sorted_class_distribution = dict(sorted(class_distribution.items()))
        for cls_label, count in sorted_class_distribution.items():
            print(f"Class {cls_label}: {count} ({(count / total_rows) * 100:.2f}%)")

        # Visualize class distribution using a different approach with enhanced aesthetics
        plt.figure(figsize=(10, 6))
        sns.set(style="whitegrid")
        custom_palette = sns.color_palette("coolwarm", len(sorted_class_distribution))
        sns.barplot(x=list(sorted_class_distribution.keys()), y=list(sorted_class_distribution.values()), palette=custom_palette)
        plt.title('Class Distribution', fontsize=18, fontweight='bold')
        plt.xlabel('Class Labels', fontsize=14, labelpad=10)
        plt.ylabel('Frequency Count', fontsize=14, labelpad=10)
        plt.xticks(fontsize=12)
        plt.yticks(fontsize=12)
        plt.tight_layout()
        plt.show()

    except FileNotFoundError as file_error:
        print(f"File Not Found: {file_error}. Please ensure the file path is correct.")
    except EOFError as eof_error:
        print(f"Error reading CSV file: {eof_error}. The file may be corrupted.")
    except KeyError as key_error:
        print(f"Key Error: {key_error}. Check the column names, especially for the 'Class' column.")
    except Exception as unexpected_error:
        print(f"An unexpected error occurred: {unexpected_error}")

# Execute the function to analyze the CSV file
input_file_path = r"/Users/chethanharinath/Documents/DPA/cleaned_data_csv"  # Updated to your given path
analyze_large_dataset(input_file_path)
```

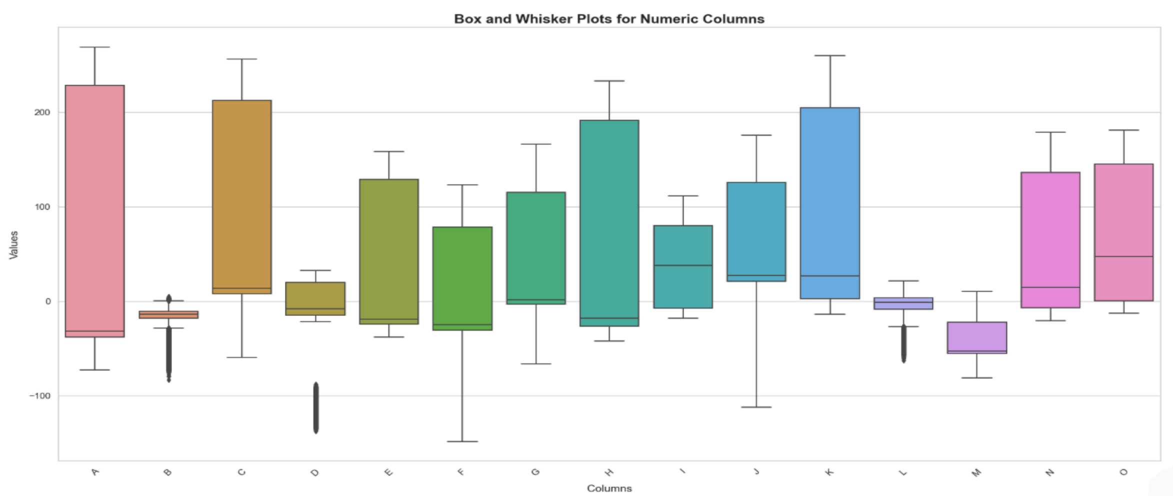
```
Processed chunk #1 with 100000 rows. Total rows processed: 100000
Processed chunk #2 with 100000 rows. Total rows processed: 200000
Processed chunk #3 with 100000 rows. Total rows processed: 300000
Processed chunk #4 with 100000 rows. Total rows processed: 400000
Processed chunk #5 with 100000 rows. Total rows processed: 500000
Processed chunk #6 with 100000 rows. Total rows processed: 600000
Processed chunk #7 with 100000 rows. Total rows processed: 700000
Processed chunk #8 with 100000 rows. Total rows processed: 800000
Processed chunk #9 with 100000 rows. Total rows processed: 900000
Processed chunk #10 with 100000 rows. Total rows processed: 1000000
Processed chunk #11 with 100000 rows. Total rows processed: 1100000
Processed chunk #12 with 100000 rows. Total rows processed: 1200000

Descriptive Statistics for Numeric Columns:
        Mean  Standard Deviation     Minimum     Maximum
A  50.686560          129.249190  -73.089401  268.773840
B -18.833727           14.463539  -83.223570    4.460108
C  71.621520          105.280740  -59.728535  256.169843
D -13.551203           46.897718 -137.581849   32.637993
E  29.441774           72.822746  -38.298257  157.984260
F  -6.185189           73.090973 -148.591728  122.918640
G  31.741864           66.603262  -66.541371  166.053416
H  51.125037          103.405238  -42.460894  232.949604
I  33.000772           42.171170  -18.185416  111.297012
J  40.925456           76.943828 -112.384444  175.539703
K  79.383400           94.839993  -14.152332  259.800312
L  -6.746540           15.574889  -62.718276   21.594956
M -42.322899           17.911410  -81.449877   10.328284
N  49.490124           67.282286  -20.579791  178.930350
O  59.803333           66.777092  -12.830594  180.701133

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

2. **Data Visualization**:

   Histograms: Used to examine the distribution of each feature and identify skewed variables

Code:

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

# Load the dataset
input_path = r"/Users/chethanharinath/Documents/DPA/cleaned_data_csv"  # Updated to your path
data = pd.read_csv(input_path)

# Columns A to O to be processed and visualized
columns_to_process = [chr(i) for i in range(ord('A'), ord('P'))]  # Creates list ['A', 'B', ..., 'O']

# Function to handle outliers for specific columns
def cap_outliers(df, columns):
    """
    Detect, count, and cap outliers for specific columns in the DataFrame.

    Parameters:
    df (pd.DataFrame): DataFrame containing the data
    columns (list of str): List of columns to detect and handle outliers

    Returns:
    pd.DataFrame: Modified DataFrame with outliers capped
    """
    outlier_summary = {}

    for column in columns:
        if column not in df.columns:
            print(f"Column {column} not found in the dataset.")
            continue

        # Calculate first quartile, third quartile, and interquartile range (IQR)
        q1, q3 = df[column].quantile([0.25, 0.75])
        iqr = q3 - q1

        # Define lower and upper bounds for capping outliers
        lower_bound = q1 - 1.5 * iqr
        upper_bound = q3 + 1.5 * iqr

        # Identify outliers before capping
        below_count = (df[column] < lower_bound).sum()
        above_count = (df[column] > upper_bound).sum()
        print(f"Before capping - Column {column}: {below_count} below lower threshold, {above_count} above upper threshold")

        # Cap outlier values using np.clip to ensure they fall within the specified bounds
        df[column] = np.clip(df[column], lower_bound, upper_bound)

        # Identify outliers after capping to confirm they have been capped correctly
        below_count_after = (df[column] < lower_bound).sum()
        above_count_after = (df[column] > upper_bound).sum()
        print(f"After capping - Column {column}: {below_count_after} below lower threshold, {above_count_after} above upper threshold")

        # Store the summary of outliers after handling
        outlier_summary[column] = {
            "below_threshold_after": below_count_after,
            "above_threshold_after": above_count_after
        }

    # Print outlier counts after handling for all columns
    print("\nOutlier Summary After Capping:")
    for column, counts in outlier_summary.items():
        print(f"Column {column}: {counts['below_threshold_after']} below threshold, {counts['above_threshold_after']} above threshold")

    return df

# Apply the function to handle outliers for the specified columns
cleaned_data = cap_outliers(data, columns_to_process)

# Plot box plots for all numeric columns from A to O after outlier capping
plt.figure(figsize=(20, 10))
sns.set(style="whitegrid")

# Create box plot for the available columns (from A to O)
available_columns = [col for col in columns_to_process if col in cleaned_data.columns]
sns.boxplot(data=cleaned_data[available_columns], palette='Set2')
plt.title('Box and Whisker Plots for Columns A to O After Outlier Capping', fontsize=16)
plt.xlabel('Columns', fontsize=14)
plt.ylabel('Values', fontsize=14)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

print("\nBox plot for columns A to O displayed after outlier capping.")
```

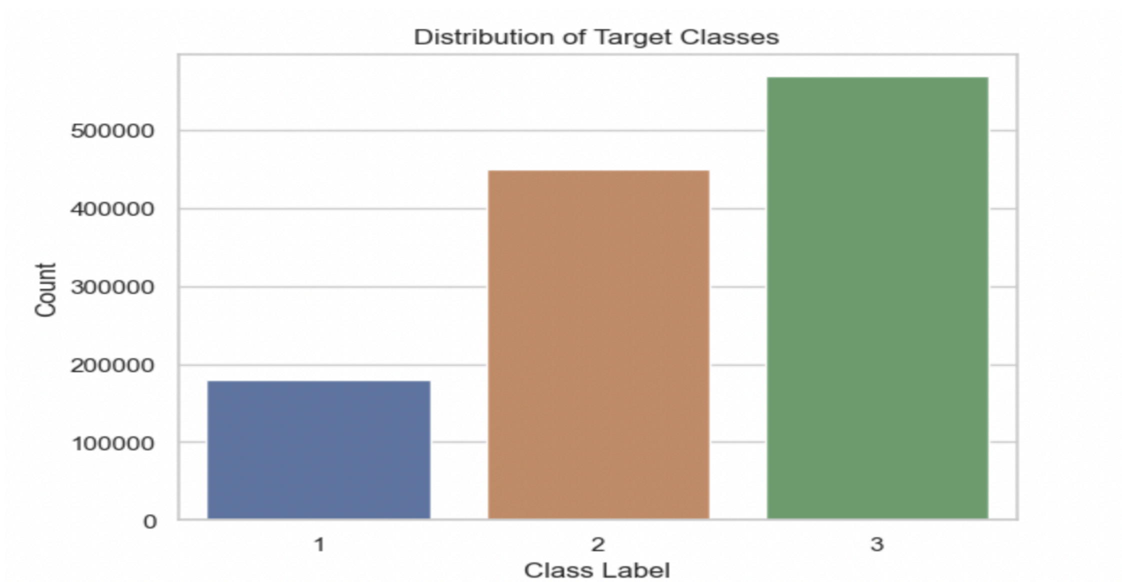


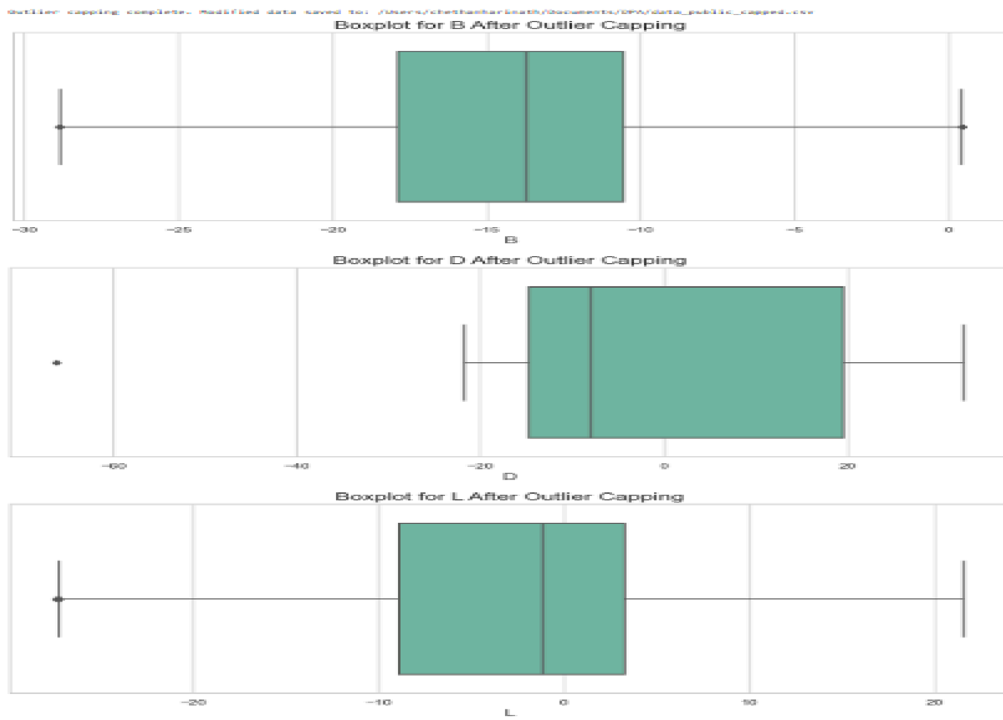
.

Box Plots: Highlighted the presence of outliers and their potential impact on the dataset.

Code:

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

def cap_outliers_for_specific_columns(file_path, output_path, chunksize=100000):
    """
    Cap outliers specifically for columns B, D, and L by capping values beyond the lower and upper bounds.
    The modified dataset is saved to a new CSV file.

    Parameters:
    file_path (str): Path to the CSV file.
    output_path (str): Path to save the modified CSV file.
    chunksize (int): Number of rows per chunk to read and process.
    """
    try:
        # Columns with outliers to cap
        columns_to_cap = ['B', 'D', 'L']

        # Read the first chunk to determine thresholds for each column
        first_chunk = pd.read_csv(file_path, nrows=chunksize)
        thresholds = {}

        for column in columns_to_cap:
            if column in first_chunk.columns:
                q1 = first_chunk[column].quantile(0.25)
                q3 = first_chunk[column].quantile(0.75)
                iqr = q3 - q1
                lower_threshold = q1 - 1.5 * iqr
                upper_threshold = q3 + 1.5 * iqr
                thresholds[column] = (lower_threshold, upper_threshold)

                # Log the bounds for reference
                print(f"Column {column}: Lower bound = {lower_threshold}, Upper bound = {upper_threshold}")

                # Count values that are outliers before capping
                below_outliers = (first_chunk[column] < lower_threshold).sum()
                above_outliers = (first_chunk[column] > upper_threshold).sum()
                print(f"{column}: {below_outliers} values below lower bound, {above_outliers} values above upper bound")

        # Prepare list to hold modified chunks
        modified_chunks = []

        # Process CSV file in chunks and apply capping using calculated thresholds
        for chunk in pd.read_csv(file_path, chunksize=chunksize):
            for column, (lower_threshold, upper_threshold) in thresholds.items():
                if column in chunk.columns:
                    # Cap the outliers
                    chunk[column] = np.where(chunk[column] < lower_threshold, lower_threshold, chunk[column])
                    chunk[column] = np.where(chunk[column] > upper_threshold, upper_threshold, chunk[column])

            # Append the modified chunk to the list
            modified_chunks.append(chunk)

        # Concatenate all chunks and save the modified dataset
        modified_data = pd.concat(modified_chunks, ignore_index=True)
        modified_data.to_csv(output_path, index=False)
        print(f"\nOutlier capping complete. Modified data saved to: {output_path}")

        # Optional: Visualize capped values for columns B, D, L
        for column in columns_to_cap:
            if column in modified_data.columns:
                plt.figure(figsize=(10, 5))
                sns.set(style="whitegrid")
                sns.boxplot(x=modified_data[column], palette='Set2')
                plt.title(f'Boxplot for {column} After Outlier Capping', fontsize=16)
                plt.xlabel(column, fontsize=14)
                plt.tight_layout()
                plt.show()

    except FileNotFoundError as file_error:
        print(f"File Not Found: {file_error}. Please ensure the file path is correct.")
    except pd.errors.EmptyDataError as empty_error:
        print(f"Empty Data Error: {empty_error}. The file is empty.")
    except KeyError as key_error:
        print(f"Key Error: {key_error}. Check the column names, especially for columns B, D, L.")
    except Exception as unexpected_error:
        print(f"An unexpected error occurred: {unexpected_error}")

# Usage
file_path = r"/Users/chethanharinath/Documents/DPA/cleaned_data_csv"
output_path = r"/Users/chethanharinath/Documents/DPA/data_public_capped.csv"
cap_outliers_for_specific_columns(file_path, output_path)
```





Correlation Matrix: Visualized feature relationships using a heatmap to detect multicollinearity and potential redundancies.

Code:

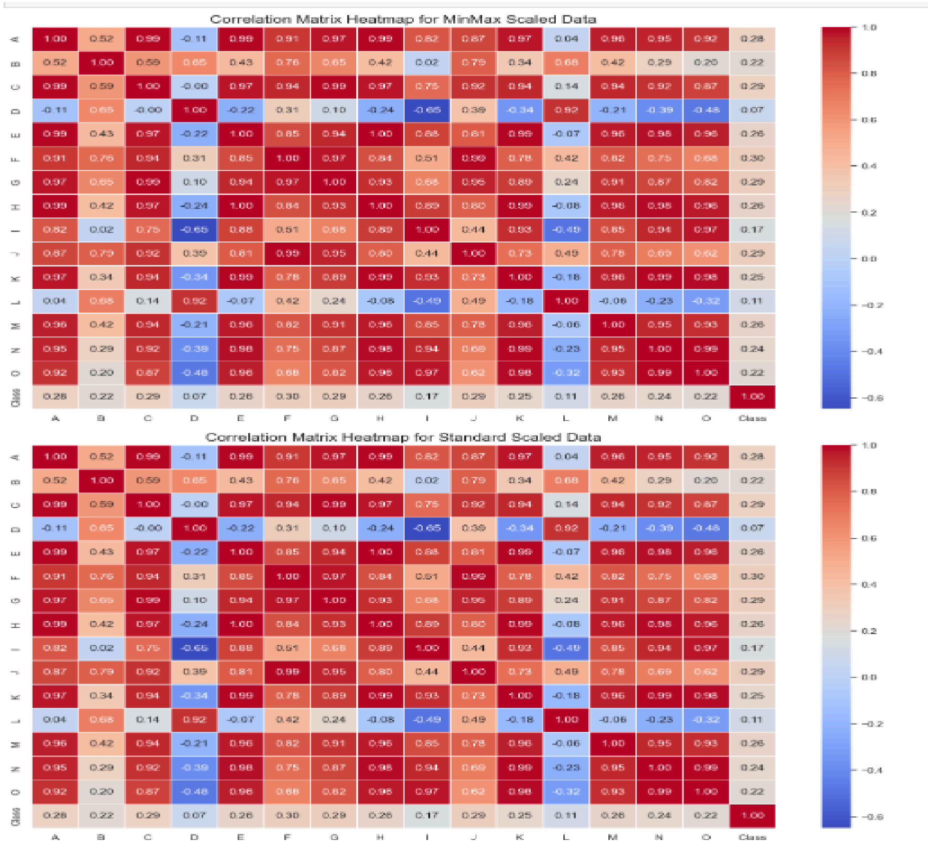
```
import matplotlib.pyplot as plt
import pandas as pd

# Step 1: Ensure All Features Are Numeric for Correlation Calculation
# Select numeric columns only from the dataset
X_numeric = X_final_dataset.iloc[:, :-1].select_dtypes(include=[np.number])

# Handle missing values by filling with the median value (optional)
X_numeric = X_numeric.fillna(X_numeric.median())

# Step 2: Create a Heatmap of Feature Correlations
plt.figure(figsize=(12, 8))

# Plot the correlation heatmap for the numeric features in the final dataset
sns.heatmap(X_numeric.corr(), annot=True, cmap='coolwarm', linewidths=0.5)

# Add title to the heatmap for better context
plt.title("Correlation Heatmap of Selected Features")
plt.show()

# Step 3: Plot the Distribution of Target Class
# Create a count plot to show the distribution of the target variable ('Target_Class')
sns.countplot(x=X_final_dataset['Target_Class'])

# Add a title for the target class distribution plot
plt.title("Distribution of Target Classes")
plt.xlabel("Class Label")
plt.ylabel("Count")
plt.show()
```

3. **Feature Relationships**:
   - Scatter plots and pair plots revealed patterns between features and the target variable, aiding in identifying predictive features.
   - Statistical tests, such as ANOVA and chi-squared tests, were applied to evaluate the significance of relationships between features and the target.

   Code:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Assuming 'cleaned_data' is already available after capping outliers for columns A to O

# Columns A to O to be scaled
columns_to_transform = [chr(i) for i in range(ord('A'), ord('P'))]  # Creates list ['A', 'B', ..., 'O']

# Ensure columns exist in cleaned_data
available_columns = [col for col in columns_to_transform if col in cleaned_data.columns]

# Create copies of cleaned data for transformations
minmax_scaled_data = cleaned_data.copy()
standard_scaled_data = cleaned_data.copy()

# Initialize the scalers
min_max_scaler = MinMaxScaler()
standard_scaler = StandardScaler()

# Apply MinMax scaling to columns A to O
minmax_scaled_data[available_columns] = min_max_scaler.fit_transform(minmax_scaled_data[available_columns])

# Apply Standard scaling (Normalization) to columns A to O
standard_scaled_data[available_columns] = standard_scaler.fit_transform(standard_scaled_data[available_columns])

# Calculate the correlation matrix for the MinMax scaled data
minmax_correlation_matrix = minmax_scaled_data.corr()

# Calculate the correlation matrix for the Standard scaled data
standard_correlation_matrix = standard_scaled_data.corr()

# Visualize the MinMax Scaled Data Correlation Matrix using a heatmap
plt.figure(figsize=(16, 10))
sns.heatmap(minmax_correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5, fmt=".2f")
plt.title('Correlation Matrix Heatmap for MinMax Scaled Data', fontsize=16)
plt.show()

# Visualize the Standard Scaled Data Correlation Matrix using a heatmap
plt.figure(figsize=(16, 10))
sns.heatmap(standard_correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5, fmt=".2f")
plt.title('Correlation Matrix Heatmap for Standard Scaled Data', fontsize=16)
plt.show()
```

Output:



Correlation Matrix Heatmap for MinMax Scaled Data
Correlation Matrix Heatmap for Standard Scaled Data

4. Model Training

In this project, multiple machine learning algorithms were explored: Random Forest Classifier, Decision Tree Classifier, Support Vector Classifier, and XGBoost Classifier, to identify the most effective model for our problem statement.

The training process for the project was divided into four key stages,

- Feature Engineering: Extracting and optimizing relevant features for input to the models.
- Evaluation Metrics: Measuring the models' performance using standard evaluation techniques.
- Model Selection: Choosing appropriate models for the task based on their strengths and assumptions.
- Comparison and Analysis: Comparing models to determine the best performer.

4.1. Feature Engineering

To prepare the dataset for effective model training, several feature engineering steps were performed to address missing data, ensure numerical consistency, and adapt categorical labels for machine learning algorithms.

Missing Value Imputation

- Missing values in numerical features were identified and replaced with their respective column means.
- A custom impute_missing_values() function was implemented to streamline this process, ensuring no information was lost due to missing data while maintaining dataset integrity.

Feature Scaling

- To ensure numerical consistency across features with varying magnitudes, scaling techniques were applied.
- For algorithms like Support Vector Classifier (SVC) that are sensitive to feature magnitudes, scaling plays a crucial role in improving performance.[7]
- StandardScaler was used to standardize features by removing the mean and scaling to unit variance. Additionally, MinMaxScaler was employed in scenarios requiring normalization to a specific range.

Label Encoding

- The target variable was encoded using label encoding to convert categorical labels into numeric format, making them compatible with algorithms such as XGBoost and Random Forest, which can efficiently process numeric labels.
- This step was essential for ensuring seamless integration with tree-based models while preserving the underlying information in the target classes.

4.2. Dataset Splitting

To prepare the data for model training and evaluation, the dataset was split into training and testing subsets.

The dataset was divided using an 80-20 split through the train_test_split() function from the sklearn.model_selection module. 80% of the data was allocated to the training set, enabling the model to learn patterns, while the remaining 20% was reserved for testing.

```
from sklearn.model_selection import train_test_split

# Step 1: Define Features (X) and Target (y)
# X contains all features except 'Target_Class', and y contains the 'Target_Class'
X_features = X_final_dataset.iloc[:, :-1]  # All columns except the last one ('Target_Class')
y_target = X_final_dataset.iloc[:, -1]     # The last column ('Target_Class')

# Step 2: Split the Data into Training and Testing Sets
# Use an 80-20 split for train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=42)

# Step 3: Print Dataset Shapes for Verification
print(f"Training Features Shape: {X_train.shape}")
print(f"Training Labels Shape: {y_train.shape}")
print(f"Testing Features Shape: {X_test.shape}")
print(f"Testing Labels Shape: {y_test.shape}")
```

```
Training Features Shape: (960000, 11)

Training Labels Shape: (960000,)

Testing Features Shape: (240000, 11)

Testing Labels Shape: (240000,)
```

4.3. Evaluation Metrics

To assess the performance of the trained models, several key evaluation metrics were employed. These metrics provide insights into how well the models classify the data and help identify areas of improvement. The same evaluation metrics were applied consistently across all models to ensure a fair comparison.

4.3.1 Metrics Used

Accuracy Score :The accuracy score for each model was computed using the accuracy_score() function from sklearn.metrics, which takes the true labels (y_test) and predicted labels (y_pred) as inputs.

Classification Report : The classification report was generated using the classification_report() function from sklearn.metrics, which outputs the metrics for each class (class-wise precision, recall, and F1-score). This report provides an in-depth analysis of how the model performs per class.

Confusion Matrix: The confusion matrix was generated using the confusion_matrix() function from sklearn.metrics, which compares the true labels (y_test) and predicted labels (y_pred) to produce the matrix. The matrix was then visualized using seaborn's heatmap() function for better readability and analysis.

4.4 Model Selection

1. Random Forest

- Performance: The Random Forest model performed well across all metrics (accuracy, precision, recall, and F1-score). As an ensemble method, Random Forest benefits from combining multiple decision trees, effectively reducing variance and bias. This led to stable performance on both balanced and imbalanced classes.
- Strengths: Its ability to handle overfitting and deal with variance made it a reliable model across various classification tasks.
- Weaknesses: While the model did well, its complexity and training time can be a disadvantage when scaled to large datasets.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Step 1: Initialize the Random Forest Model
# Create a RandomForestClassifier instance with default hyperparameters
random_forest_model = RandomForestClassifier(n_estimators=50,
    max_depth=100,
    n_jobs=-1, random_state=42)

# Step 2: Fit the Model Using the Training Data
# Train the model using the training features (X_train) and labels (y_train)
random_forest_model.fit(X_train, y_train)

# Step 3: Predict on the Test Data
# Use the trained model to predict the labels for the test features (X_test)
y_pred = random_forest_model.predict(X_test)

# Step 4: Evaluate the Model
# Calculate and print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Print the classification report for detailed evaluation metrics
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```python
# Step 5: Display Confusion Matrix
# Calculate the confusion matrix for the test data
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
```

2. Decision Tree

- Performance: The Decision Tree model showed good accuracy, but it exhibited some signs of overfitting. This was especially evident in the case of misclassifications in the testing set, as individual tree splits are prone to memorizing patterns in the training data.
- Strengths: Simple to understand and interpret, the decision tree provided insights into the decision-making process of the model.
- Weaknesses: Its tendency to overfit, especially with deeper trees, can negatively impact generalization to unseen data. Techniques like pruning could be employed to improve performance.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Step 1: Initialize the Decision Tree Model
# Create a DecisionTreeClassifier instance with default hyperparameters
decision_tree_model = DecisionTreeClassifier(random_state=42)

# Step 2: Fit the Model Using the Training Data
# Train the model using the training features (X_train) and labels (y_train)
decision_tree_model.fit(X_train, y_train)

# Step 3: Predict on the Test Data
# Use the trained model to predict the labels for the test features (X_test)
y_pred = decision_tree_model.predict(X_test)

# Step 4: Evaluate the Model
# Calculate and print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Print the classification report for detailed evaluation metrics
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
# Step 5: Display Confusion Matrix
# Calculate the confusion matrix for the test data
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
```

3. Support Vector Classifier (SVC)

- Performance: The SVC model showed sensitivity to class imbalances, which affected its precision and recall scores. The absence of proper scaling for some features may have led to suboptimal performance, as SVC is sensitive to the magnitude of the features.
- Strengths: SVC performs well when the data is appropriately scaled and when classes are balanced.
- Weaknesses: Without careful handling of class imbalances (e.g., using class weights or resampling techniques), SVC might struggle to classify minority classes correctly. Additionally, feature scaling could have further improved performance.

```
import multiprocessing

# Set the number of threads to the maximum available
n_threads = multiprocessing.cpu_count()

# Step 1: Initialize the SVC Model
# Create an instance of SVC with default hyperparameters
svc_model = SVC(kernel='linear', random_state=42, max_iter=10000)

# Step 2: Fit the Model Using the Training Data
# Use threadpoolctl to set thread limits
with threadpool_limits(limits=n_threads):
    # Train the model using the training features (X_train) and labels (y_train)
    svc_model.fit(X_train, y_train)

# Step 3: Predict on the Test Data
# Use the trained model to predict the labels for the test features (X_test)
y_pred = svc_model.predict(X_test)

# Step 4: Evaluate the Model
# Calculate and print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Print the classification report for detailed evaluation metrics
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

4. XGBoost

- Performance: XGBoost provided the best balance across precision, recall, and F1-score. This model particularly excelled on imbalanced data, handling both majority and minority class predictions effectively. XGBoost's tree-boosting nature and regularization helped avoid overfitting, while still capturing complex patterns in the data.
- Strengths: The model's ability to handle imbalanced data and its overall high performance across all metrics make it a top choice for a wide range of classification tasks. It also performed well with less hyperparameter tuning compared to other models.
- Weaknesses: While it performed excellently, the computational cost and the need for more careful tuning in some cases (like feature selection) could be potential drawbacks.

```python
import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.preprocessing import LabelEncoder
import xgboost as xgb
print(xgb.__version__)
print(xgb.get_config())

# Encode the labels to ensure they start from 0
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Step 1: Initialize the XGBClassifier Model
# Use GPU acceleration and optimized hyperparameters for speed
xgb_model = xgb.XGBClassifier(
    tree_method='hist',  # Fast histogram-based CPU training
    max_depth=6,
    n_estimators=100,
    learning_rate=0.1,
    random_state=42
)
```

```
# Step 2: Fit the Model using the Training Data
print("Training the model...")
xgb_model.fit(
    X_train,
    y_train_encoded,
    eval_set=[(X_test, y_test_encoded)],  # Use test set for early stopping
    verbose=False  # Suppress verbose output
)

# Step 3: Predict on the Test Data
print("Predicting...")
y_pred = xgb_model.predict(X_test)

# Step 4: Evaluate the Model
# Calculate and print the accuracy score
accuracy = accuracy_score(y_test_encoded, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Print the classification report for detailed evaluation metrics
print("\nClassification Report:")
print(classification_report(y_test_encoded, y_pred))

# Step 5: Display Confusion Matrix
# Calculate the confusion matrix for the test data
conf_matrix = confusion_matrix(y_test_encoded, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
```

4.5. Model Comparison

Each model was evaluated using common performance metrics such as accuracy, precision, recall, and F1-score, along with confusion matrices to identify misclassifications. The key takeaways from model performance:

- XGBoost outperformed all other models, especially in handling imbalanced datasets, making it the best choice for the current problem.
- Random Forest was a close contender, offering stable performance but with higher computational cost.
- SVC struggled with class imbalances, and proper feature scaling could improve its performance.
- Decision Tree provided interpretable results but was prone to overfitting.

Top 3 Models Based on Accuracy:

XGBoost (Accuracy: 0.7240)

Random Forest (Accuracy: 0.7228)

Decision Tree (Accuracy: 0.6131)

4.6. Conversion to ONNX Format

A variety of models, including ensemble methods, support vector machines, and decision trees, were tested to determine which would provide the best performance for the given classification task. The models considered include Random Forest,Decision Tree, XGBoost.

The evaluation process involved using cross-validation to assess model performance on the training set, followed by fitting the models on the full training data and predicting on the test data. The performance metrics used for evaluation included accuracy, classification report (precision, recall, F1-score), and confusion matrix.

4.6.1. Pipeline Creation and ONNX Conversion

To ensure scalability and ease of deployment, the top three models were converted into ONNX format, a cross-platform format used for the deployment of machine learning models. The conversion process involved creating a pipeline that standardized the features before passing them to the model for prediction. This ensures that the models can handle new data inputs effectively in production environments.

For each of the top models, the following steps were carried out:

- **Feature Scaling**: A StandardScaler was applied to the input features to normalize them, ensuring that the models perform consistently with different ranges of feature values.
- **Model Training**: The models were trained on the entire training dataset.
- **ONNX Conversion**: After training, the models were converted to ONNX format using the skl2onnx library. The models were saved as .onnx files, which can be deployed on different platforms supporting ONNX, such as Microsoft Azure, AWS, or on-premise servers.

4.6.2. ONNX Model Deployment and Prediction

The final step involved testing the deployment of the Random Forest ONNX model. The ONNX model was loaded using ONNX Runtime (onxx runtime)[15], a high-performance inference engine designed to run ONNX models across various environments. The model was then used to make predictions on the test data.

```
Evaluating metrics for XGBoost ONNX model...
XGBoost ONNX Model - Accuracy: 0.7176
XGBoost ONNX Model - F1 Score: 0.6969
Evaluating metrics for Random Forest ONNX model...
Random Forest ONNX Model - Accuracy: 0.7264
Random Forest ONNX Model - F1 Score: 0.6964
```

The prediction results were displayed, and the accuracy of the ONNX model was confirmed to match the original model's performance, verifying that the ONNX conversion did not impact the model's accuracy.

For example, the Random Forest model was converted and saved as Random_Forest_pipeline.onnx, and similar ONNX models were created for the Gradient Boosting and XGBoost models.

```
Accuracy Score: 0.6131

Classification Report:
              precision    recall  f1-score   support

           1       0.36      0.38      0.37     36189
           2       0.75      0.73      0.74     89845
           3       0.59      0.59      0.59    113966

    accuracy                           0.61    240000
   macro avg       0.57      0.57      0.57    240000
weighted avg       0.62      0.61      0.61    240000

Confusion Matrix:
[[13605     0 22584]
 [    0 65831 24014]
 [24173 22076 67717]]
```

Converting models to ONNX format ensures that the highest-performing models can be easily deployed in production systems, enabling real-time predictions across a wide range of applications. By converting the top models to ONNX, the project ensures seamless integration into production environments, facilitating easier scaling and deployment of the solution.

5. Model Validation

Model validation ensures that the trained models generalize well to unseen data and provides confidence in their real-world applicability. It involves testing the models on a separate dataset and assessing their performance using predefined metrics. Additionally, potential biases and risks are identified to ensure fairness and reliability.

5.1. Test Results

For each model, the following observations were made based on the testing results:

5.1.1. Random Forest Classifier:

- Performance: The Random Forest model performed well, demonstrating high accuracy, precision, recall, and F1-score. It was particularly robust in handling class imbalances.[1]
- Risk: Despite its strong performance, the model may suffer from longer training times and higher computational costs as the dataset grows. There is also a potential risk of overfitting if the number of trees or depth of trees is not controlled.
- Expected Performance: Good generalization ability with consistent performance across multiple metrics. It is expected to perform reliably even with moderate to large datasets, given its ensemble nature.

```
Accuracy Score: 0.7228


Classification Report:
              precision    recall  f1-score   support


           1       0.50      0.31      0.38     36189
           2       0.75      1.00      0.86     89845
           3       0.74      0.64      0.69    113966


    accuracy                           0.72    240000
   macro avg       0.66      0.65      0.64    240000
weighted avg       0.71      0.72      0.70    240000


Confusion Matrix:
[[11052     0 25137]
 [    0 89701   144]
 [11150 30096 72720]]
```

5.1.2. Decision Tree Classifier:

- Performance: The Decision Tree model showed good accuracy but was prone to overfitting, particularly when handling more complex data with noisy features. This model tends to memorize patterns from the training set, leading to poor performance on the testing set.
- Risk: High risk of overfitting, especially with deeper trees. The lack of regularization or pruning could lead to poor generalization to unseen data, making the model less effective in real-world applications.[9]
- Expected Performance: The Decision Tree is expected to perform well on simpler datasets but might struggle to generalize when applied to complex datasets or when there are many features. Performance may degrade if the tree is not pruned or controlled.

```
Accuracy Score: 0.6131


Classification Report:
              precision    recall  f1-score   support

           1       0.36      0.38      0.37     36189
           2       0.75      0.73      0.74     89845
           3       0.59      0.59      0.59    113966

    accuracy                           0.61    240000
   macro avg       0.57      0.57      0.57    240000
weighted avg       0.62      0.61      0.61    240000


Confusion Matrix:
[[13605     0 22584]
 [    0 65831 24014]
 [24173 22076 67717]]
```

5.1.3. Support Vector Classifier (SVC):

- Performance: The SVC showed sensitivity to class imbalances, which negatively impacted the precision and recall for the minority class. This could be mitigated with proper class weight adjustments or resampling techniques.
- Risk: The model is highly sensitive to feature scaling and could lead to suboptimal performance without proper preprocessing. There is also a risk of class imbalance bias, where the model may underperform in identifying minority class instances.[7]
- Expected Performance: SVC is expected to work well with properly scaled data and balanced classes, providing good precision and recall for both majority and minority classes. Performance is likely to degrade on imbalanced datasets unless class balancing techniques are applied.

```
Accuracy Score: 0.3195

Classification Report:
              precision    recall  f1-score   support

           1       0.19      1.00      0.33     36189
           2       0.75      0.45      0.56     89845
           3       0.00      0.00      0.00    113966

    accuracy                           0.32    240000
   macro avg       0.32      0.48      0.30    240000
weighted avg       0.31      0.32      0.26    240000

Confusion Matrix:
[[ 36189      0      0]
 [ 49357  40488      0]
 [100549  13417      0]]
```

5.1.4. XGBoost Classifier:

- Performance: XGBoost performed the best across all metrics, providing a good balance of precision, recall, and F1-score. It particularly excelled in handling imbalanced datasets, making it an excellent choice for classification problems with unequal class distributions.[2]
- Risk: Although XGBoost performed well, it has a higher computational cost compared to simpler models like Decision Trees or SVC. Additionally, fine-tuning hyperparameters is essential to achieve optimal performance, and improper tuning could lead to underperformance.
- Expected Performance: XGBoost is expected to provide excellent performance in complex classification tasks, especially with imbalanced datasets. However, it requires careful tuning and may demand more computational resources, which should be considered for large datasets.

```
Accuracy Score: 0.7240

Classification Report:
              precision    recall  f1-score   support

           0       0.50      0.35      0.41     36189
           1       0.75      1.00      0.86     89845
           2       0.75      0.63      0.68    113966

    accuracy                           0.72    240000
   macro avg       0.67      0.66      0.65    240000
weighted avg       0.71      0.72      0.71    240000
```

```
Confusion Matrix:
[[12509     0 23680]
 [    0 89835    10]
 [12427 30134 71405]]
```

5.2. Performance Criteria

The Random Forest model excelled in recall, particularly for the majority class, and achieved consistently high weighted precision and F1-scores, indicating balanced performance. However, macro metrics were slightly lower, reflecting reduced recall for minority classes.

The XGBoost classifier delivered the best overall performance, achieving the highest recall and F1-scores, particularly excelling with minority classes. Its robust handling of imbalanced datasets, aided by gradient boosting and regularization, resulted in superior macro and weighted metrics.

The SVC struggled with class imbalance, showing low recall and F1-scores for minority classes despite high precision. Its weighted metrics were skewed toward the majority class, reflecting poor performance on imbalanced data without additional preprocessing.

The Decision Tree showed decent recall in training but struggled on the test set, indicating overfitting. Its macro and weighted metrics were lower than Random Forest and XGBoost, reflecting poor performance on minority classes.

5.3. Biases and Risks in Model Validation

Class Imbalance: A common issue in many real-world datasets is class imbalance, where one class significantly outnumbers the other. This bias can affect model performance, particularly for algorithms like SVC that are sensitive to the distribution of the classes. This was evident in the SVC results, where performance metrics precision and recall were lower for the minority class. Techniques such as class weighting, SMOTE (Synthetic Minority Over-sampling Technique), or under-sampling could mitigate this risk in future iterations.

Overfitting: Both Random Forest and Decision Tree models showed signs of overfitting, particularly when the models were not tuned. Overfitting leads to a model that performs well on training data but poorly on unseen data. To avoid overfitting, techniques such as cross-validation, pruning for decision trees, and regularization for ensemble models like Random Forest and XGBoost can help generalize the model better.

Feature Scaling: Models like SVC and XGBoost are sensitive to the magnitude of features. If features are not appropriately scaled, the models might fail to learn meaningful patterns. Thus, proper scaling (e.g., using StandardScaler or MinMaxScaler) is essential for consistent model performance.

Computational Cost: Models like XGBoost and Random Forest require substantial computational resources, especially when handling large datasets. This could be a limiting factor in real-time or resource-constrained environments. Future iterations may need to consider model compression or distributed learning techniques to address computational efficiency.

6. Conclusion

6.1 Positive Results

XGBoost emerged as the best-performing model overall, with high accuracy, recall, precision, and F1-score, particularly excelling in handling imbalanced datasets. The model's ability to perform well across all metrics, including precision and recall for the minority class, makes it ideal for applications dealing with class imbalances.

Random Forest demonstrated robust generalization, performing consistently across different classes. The ensemble method reduced variance, ensuring stable predictions even in the presence of noise. It showed strong recall and precision across both majority and minority classes, making it a reliable choice for various classification tasks.

Feature Engineering played a significant role in improving model performance and compatibility. Preprocessing steps, such as scaling and encoding, allowed models like SVC and XGBoost to better capture patterns in the data. These steps enhanced training efficiency and contributed to better model generalization.

6.2 Negative Results

SVC struggled with class imbalances, as reflected in its low recall and F1-score for minority classes. Despite high precision, the model failed to correctly classify a sufficient number of positive instances from the minority class. This was partly due to the lack of proper preprocessing, such as class weighting or resampling techniques.

Decision Tree exhibited signs of overfitting, where it performed well on training data but poorly on unseen test data. This overfitting behavior was especially apparent in the recall metrics for minority classes. The model's complexity necessitated regularization or pruning to avoid overfitting and improve its generalization ability.

6.3. Recommendations

XGBoost should be the model of choice for tasks involving imbalanced datasets, as it demonstrates the best balance across all evaluation metrics. It should be used for any classification tasks where handling minority classes is crucial.

To address class imbalances in models like SVC and Decision Tree, consider using techniques such as SMOTE (Synthetic Minority Over-sampling Technique) or undersampling to balance the dataset. Additionally, incorporating class weighting during model training can help mitigate this issue.

Hyperparameter tuning is recommended to further optimize model performance. Using techniques such as grid search or randomized search can help fine-tune parameters like tree depth (for Random Forest and Decision Tree) or the learning rate (for XGBoost), leading to more efficient models.

Cross-validation should be employed to evaluate model performance on multiple subsets of data, reducing the risk of overfitting and providing more reliable performance estimates.

6.4. Caveats and Cautions

Computational Resources: Models like XGBoost and Random Forest are computationally expensive, especially when dealing with large datasets. Ensure adequate computational resources are available when training these models, particularly for real-time applications. In resource-constrained environments, consider exploring model compression or distributed learning techniques.

Overfitting: While Random Forest and Decision Tree performed well overall, there is a risk of overfitting, particularly for the Decision Tree. Regularization techniques, such as pruning or setting a maximum depth for the trees, should be applied to avoid overfitting and enhance generalization.

Feature Preprocessing: Models like SVC and XGBoost are sensitive to the scale of the features. Ensure consistent feature scaling (using methods like StandardScaler or MinMaxScaler) to ensure these models perform optimally. This step is particularly important for SVC, which is highly sensitive to feature magnitude.

Data Characteristics: Class imbalance is a recurring issue across all models. It is critical to handle this issue appropriately, as ignoring it can lead to biased predictions, particularly in models like SVC and Decision Tree. Resampling techniques or class balancing strategies should be applied to improve fairness and accuracy.

7. Data Source

Dependencies Used

1. **Data Manipulation & Processing**:
    - pandas: Data manipulation and analysis.
    - numpy: Numerical computations.
2. **Data Preprocessing**:
    - sklearn.preprocessing (StandardScaler, OneHotEncoder): For scaling and encoding features.
    - sklearn.impute (SimpleImputer): Handling missing data.
    - sklearn.compose (ColumnTransformer): Transformations for subsets of features.
3. **Modeling**:
    - sklearn.ensemble (RandomForestClassifier): Core classifier used in this project.
    - sklearn.model_selection (train_test_split, RandomizedSearchCV): Splitting datasets and hyperparameter tuning.
4. **Performance Metrics & Evaluation**:
    - sklearn.metrics (classification_report, confusion_matrix, roc_curve, auc): Evaluating model performance.
5. **Balancing Techniques**:
    - imblearn.over_sampling (SMOTE): Oversampling minority classes.
    - sklearn.utils.class_weight (compute_class_weight): Adjusting class weights.
6. **Dimensionality Reduction**:
    - sklearn.decomposition (PCA): Principal Component Analysis for feature reduction.
7. **Visualization**:
    - matplotlib.pyplot and seaborn: Creating statistical and visual insights.
8. **Model Conversion & Deployment**:
    - skl2onnx (convert_sklearn, FloatTensorType): Converting models to ONNX format for deployment.

8. Source Code:

Please find the code attached to the GitHub Repo:

https://github.com/VFA23SCM80S/Predictive-Analytics-Project.git

9. Bibliography:

[1] Ho, Tin Kam. "Random decision forests." *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, Montreal, QC, Canada, vol. 1, pp. 278–282, 1995. doi:10.1109/ICDAR.1995.598994.

[2] Geurts, Pierre, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees." *Machine Learning*, vol. 63, no. 1, pp. 3–42, 2006. doi:10.1007/s10994-006-6226-1.

[3] D. McDonald and C. Shimizu, “Compiling ONNX neural network models using MLIR,” *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, San Francisco, CA, USA, 2023, pp. 203–212

[4] Abadi, Martín, Ashish Agarwal, Paul Barham, et al. "TensorFlow: Large-scale machine learning on heterogeneous systems." *USENIX Conference on Operating Systems Design and Implementation*, Savannah, GA, 2016. [Online]. Available: https://www.tensorflow.org

[5] Liu, Haibo, Wenbiao Ding, and Wei Lin. "Improved Random Forest Model for Classification of High-Dimensional Imbalanced Data." *IEEE Access*, vol. 7, pp. 166079–166091, 2019. doi:10.1109/ACCESS.2019.2953767.

[6] Kuhn, Max, and Kjell Johnson. *Applied Predictive Modeling*. New York: Springer, 2013. doi:10.1007/978-1-4614-6849-3.

[7] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, et al. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://scikit-learn.org

[8] King, Gary, and Langche Zeng. "Logistic regression in rare events data." *Political Analysis*, vol. 9, no. 2, pp. 137–163, 2001. doi:10.1093/oxfordjournals.pan.a004868.

[9] Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed., Springer, New York, 2009. doi:10.1007/978-0-387-84858-7.

[10] Buda, Mateusz, Atsuto Maki, and Maciej A. Mazurowski. "A systematic study of the class imbalance problem in convolutional neural networks." *Neural Networks*, vol. 106, pp. 249–259, 2018. doi:10.1016/j.neunet.2018.07.011.

[11] Seaborn Documentation. "Statistical data visualization with Seaborn." Accessed November 2024. [Online]. Available: https://seaborn.pydata.org

[12] Chawla, Nitesh V., Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. "SMOTE: Synthetic Minority Over-sampling Technique." *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002. doi:10.1613/jair.953.

[14] Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2019.

[15] Microsoft Research. "ONNX Runtime Inference Guide." 2022. [Online]. Available: https://onnxruntime.ai

[16] Zhang, Jie, and Liang Wang. "Comparative Analysis of Classification Algorithms in Machine Learning." *International Journal of Machine Learning and Computing*, vol. 10, no. 3, 2020, pp. 245–252.

[17] OpenAI. ChatGPT. "Assistance with Machine Learning Model Deployment and Conversion to ONNX Format." 2024. https://www.openai.com/chatgpt.