Configuration Manual

MSc Research Project
MSc Cybersecurity

Vaibhav Tupe
Student ID: X23162929

School of Computing
National College of Ireland

Supervisor: Niall Heffernan

National College of Ireland

MSc Project Submission

Sheet School of Computing

Student Name:	Vaibhav Ramesh Tupe
Student ID:	X23162929
Programme:	MSc Cybersecurity **Year:** 2024-2025
Module:	MSc Research Project
Lecturer:	Niall Heffernan
Submission Due Date:	12/12/2024
Project Title:	Optimizing Deep Packet Inspection for Securing Remote Work Communication using Machine Learning: Addressing Performance and Privacy Concerns.
Word Count:	3079 Words. **Page Count:** 31 Pages.

Signature: Vaibhav Ramesh Tupe

Date: 12/12/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Optimizing Deep Packet Inspection for Securing Remote work communication using Machine learning: Addressing Performance and Privacy Concerns

Vaibhav Ramesh Tupe
Student ID: x23162929

1. Introduction

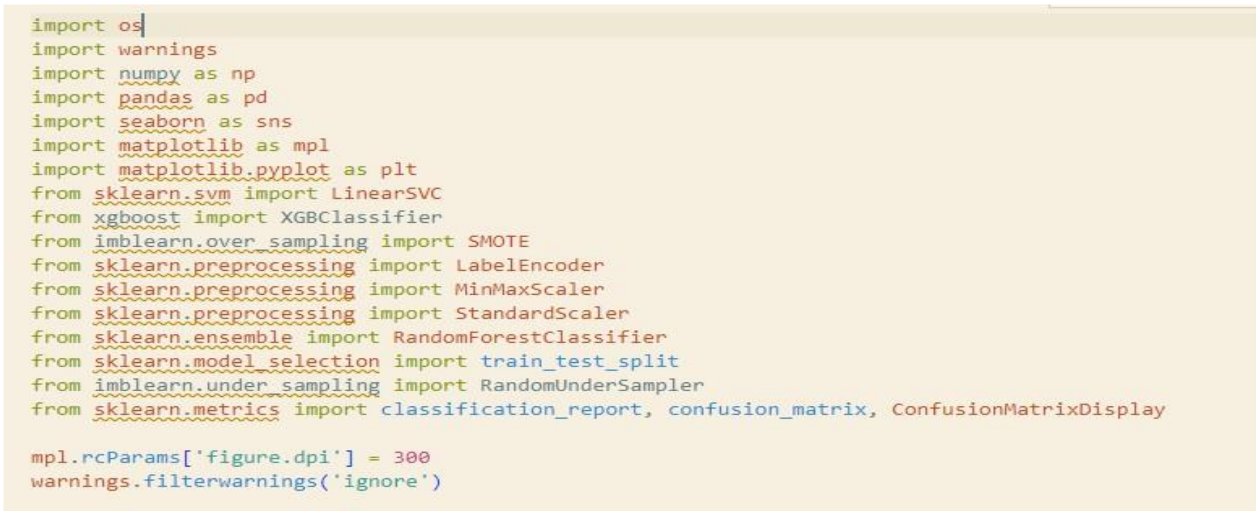
In this config guide, we explain how to configure and optimize a machine learning-based system for deep packet inspection to secure remote work communication for both individual and enterprise networks. This proposed system uses the advanced data analysis techniques and three chosen classification techniques i.e. SVM, Random Forest, and XGBoost to detect the malicious network traffic activities. These step-by- step instructions make sure that the researchers following this guide can replicate the setup for themselves in effective experimentation and implementation.

2. System Requirements and Libraries

To set up this proposed system we will need a machine with Python installed and contain sufficient computational resources in order to process the large datasets. In this study the key libraries such as NumPy, Pandas, Scikit-learn, and XGBoost will handle data manipulation, machine learning, and visualization tasks while the tools like Matplotlib and Seaborn will be able to generate insights and visualizations through plots. Moreover the specialized libraries such as Imbalanced-learn will help balance this uneven dataset which we chose. Make sure to have these dependencies installed for error free implementation.

3. Data Execution Explanation

3.1. Import the Libraries

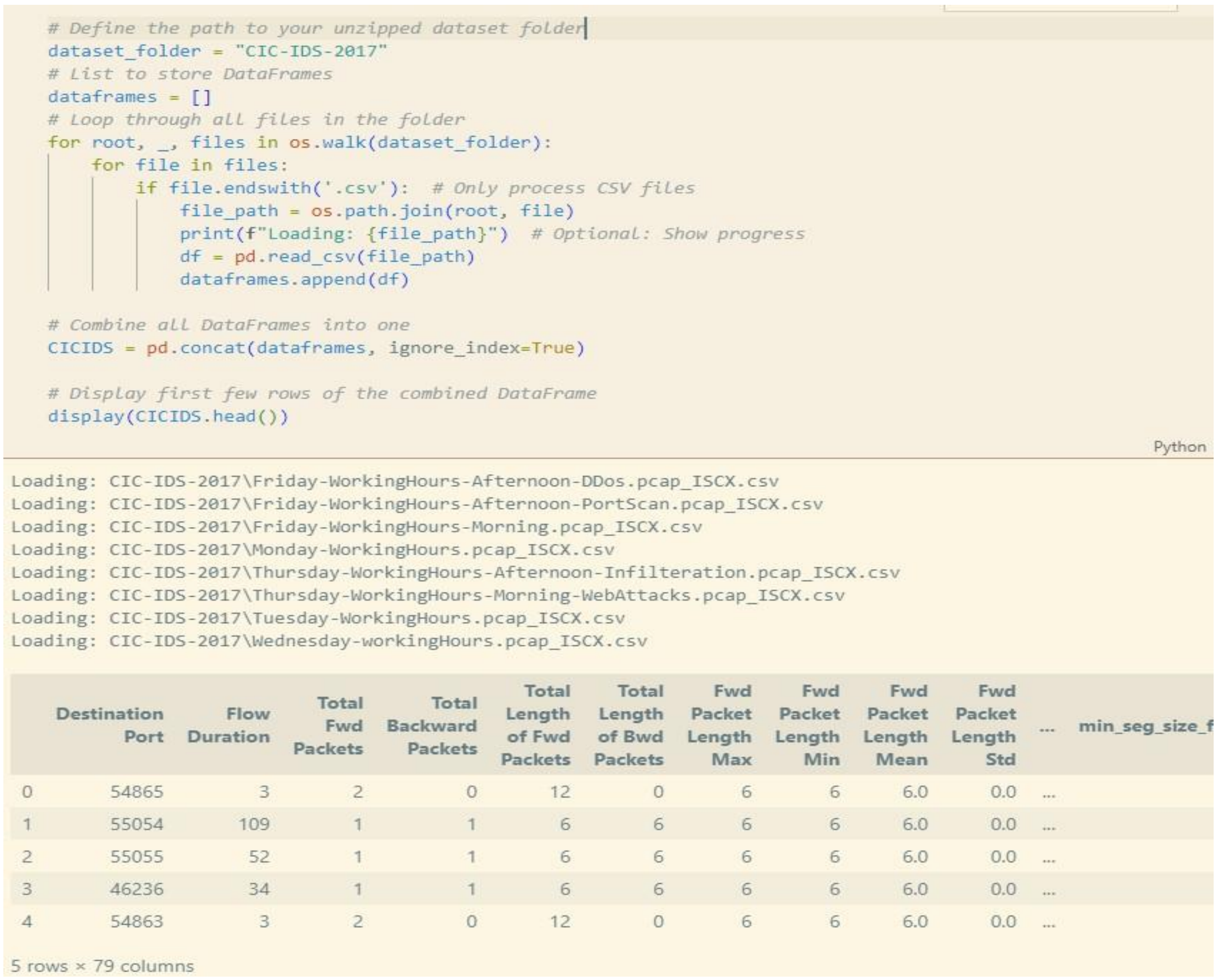


```
import os
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
from sklearn.svm import LinearSVC
from xgboost import XGBClassifier
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from imblearn.under_sampling import RandomUnderSampler
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

mpl.rcParams['figure.dpi'] = 300
warnings.filterwarnings('ignore')
```

Figure 1: This code sets up essential libraries for data processing, visualization, and machine learning to ensure smooth workflow

Step 1: In this step, we first import various libraries essential for data processing and model building. Libraries like os and warnings are used for managing paths and suppressing warnings, while numpy and pandas handle numerical computations and data manipulation. Visualization tools such as matplotlib and seaborn create insightful plots, and machine learning libraries like scikit-learn and xgboost support the implementation of classification algorithms. The imbalanced-learn library addresses class imbalance issues. This step sets up the foundational environment for the project.

3.2. About the Dataset



```python
# Define the path to your unzipped dataset folder
dataset_folder = "CIC-IDS-2017"
# List to store DataFrames
dataframes = []
# Loop through all files in the folder
for root, _, files in os.walk(dataset_folder):
    for file in files:
        if file.endswith('.csv'):  # Only process CSV files
            file_path = os.path.join(root, file)
            print(f"Loading: {file_path}")  # Optional: Show progress
            df = pd.read_csv(file_path)
            dataframes.append(df)

# Combine all DataFrames into one
CICIDS = pd.concat(dataframes, ignore_index=True)

# Display first few rows of the combined DataFrame
display(CICIDS.head())
```

Python

```
Loading: CIC-IDS-2017\Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv
Loading: CIC-IDS-2017\Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv
Loading: CIC-IDS-2017\Friday-WorkingHours-Morning.pcap_ISCX.csv
Loading: CIC-IDS-2017\Monday-WorkingHours.pcap_ISCX.csv
Loading: CIC-IDS-2017\Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv
Loading: CIC-IDS-2017\Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv
Loading: CIC-IDS-2017\Tuesday-WorkingHours.pcap_ISCX.csv
Loading: CIC-IDS-2017\Wednesday-workingHours.pcap_ISCX.csv
```

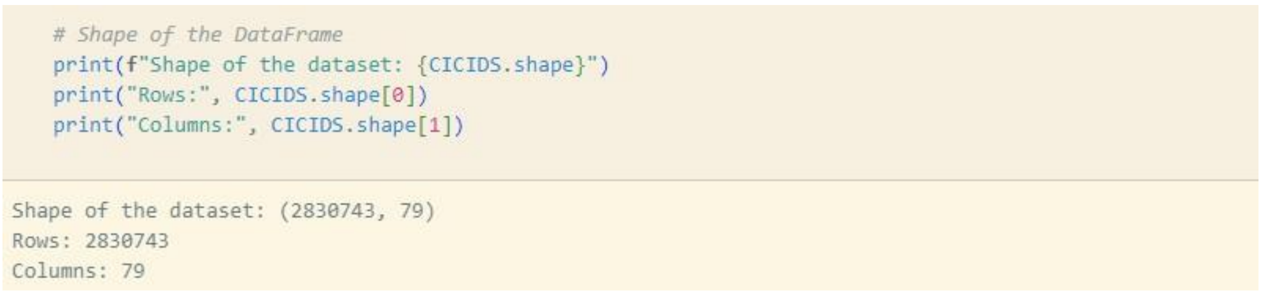
	Destination Port	Flow Duration	Total Fwd Packets	Total Backward Packets	Total Length of Fwd Packets	Total Length of Bwd Packets	Fwd Packet Length Max	Fwd Packet Length Min	Fwd Packet Length Mean	Fwd Packet Length Std	...	min_seg_size_f
0	54865	3	2	0	12	0	6	6	6.0	0.0	...	
1	55054	109	1	1	6	6	6	6	6.0	0.0	...	
2	55055	52	1	1	6	6	6	6	6.0	0.0	...	
3	46236	34	1	1	6	6	6	6	6.0	0.0	...	
4	54863	3	2	0	12	0	6	6	6.0	0.0	...	

5 rows × 79 columns

Figure 2: This code loads and combines multiple CSV files from the CIC-IDS-2017 dataset into a single DataFrame for analysis.

Step 2: This step focuses on loading the dataset. It involves specifying the dataset folder path and using os.walk to iterate through its contents, processing only .csv files with pd.read_csv(). These files are read into DataFrames, appended to a list, and concatenated into a single large DataFrame using pd.concat(). The dataset is previewed using display(CICIDS.head()) to ensure successful loading.

3.3. Basic Analysis

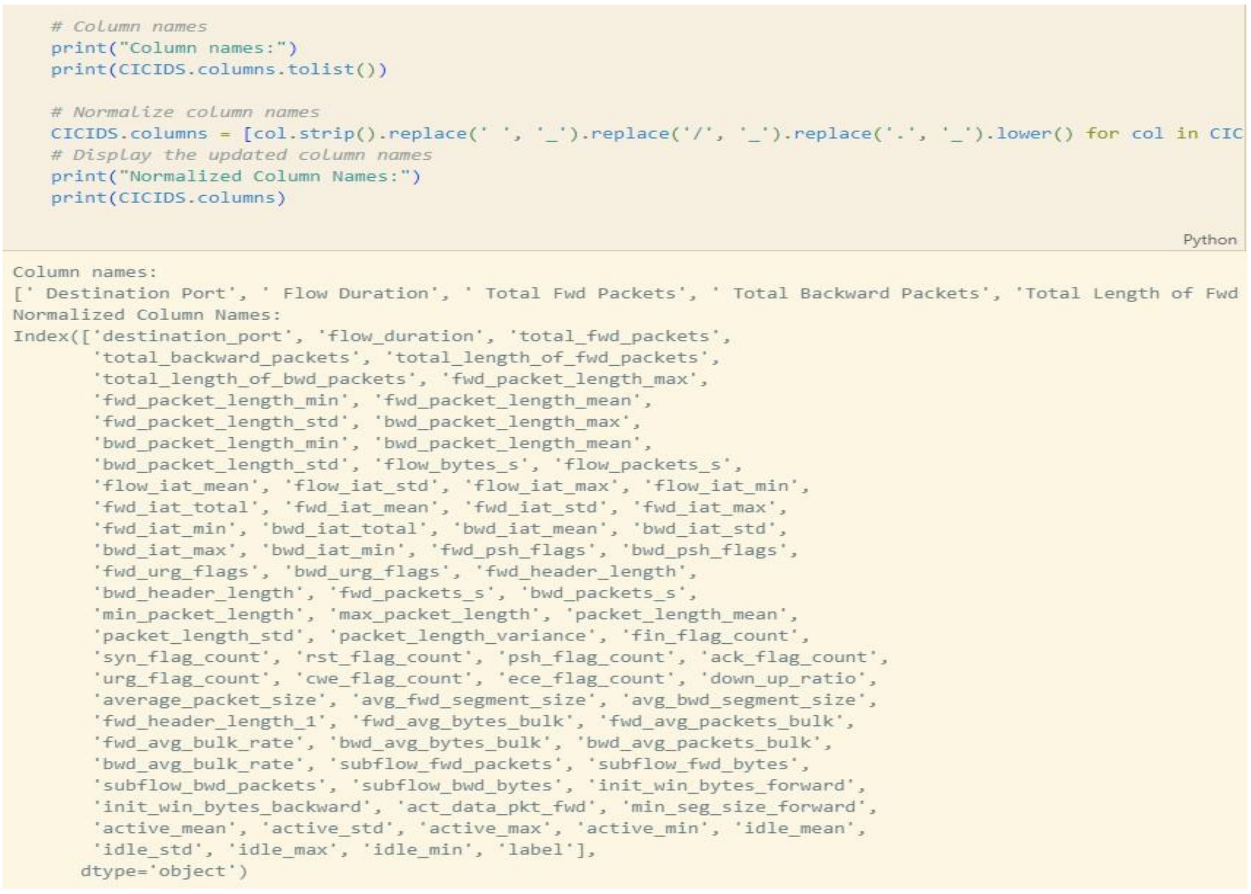


```
# Shape of the DataFrame
print(f"Shape of the dataset: {CICIDS.shape}")
print("Rows:", CICIDS.shape[0])
print("Columns:", CICIDS.shape[1])
```

```
Shape of the dataset: (2830743, 79)
Rows: 2830743
Columns: 79
```

Figure 3: This code retrieves and displays the number of rows and columns in the combined dataset.

Step 3: In this step, the number of rows and columns in the dataset is determined to understand its structure. This basic analysis provides a quick overview of the dataset's dimensions and scope.



```
# Column names
print("Column names:")
print(CICIDS.columns.tolist())

# Normalize column names
CICIDS.columns = [col.strip().replace(' ', '_').replace('/', '_').replace('.', '_').lower() for col in CIC
# Display the updated column names
print("Normalized Column Names:")
print(CICIDS.columns)
```

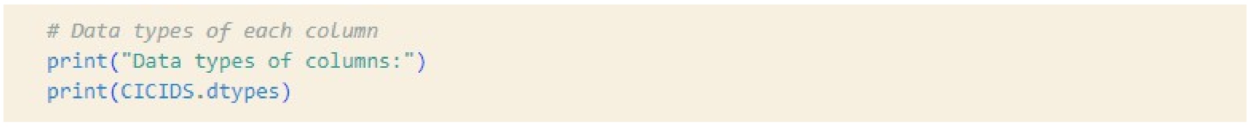
Python

```
Column names:
[' Destination Port', ' Flow Duration', ' Total Fwd Packets', ' Total Backward Packets', 'Total Length of Fwd
Normalized Column Names:
Index(['destination_port', 'flow_duration', 'total_fwd_packets',
       'total_backward_packets', 'total_length_of_fwd_packets',
       'total_length_of_bwd_packets', 'fwd_packet_length_max',
       'fwd_packet_length_min', 'fwd_packet_length_mean',
       'fwd_packet_length_std', 'bwd_packet_length_max',
       'bwd_packet_length_min', 'bwd_packet_length_mean',
       'bwd_packet_length_std', 'flow_bytes_s', 'flow_packets_s',
       'flow_iat_mean', 'flow_iat_std', 'flow_iat_max', 'flow_iat_min',
       'fwd_iat_total', 'fwd_iat_mean', 'fwd_iat_std', 'fwd_iat_max',
       'fwd_iat_min', 'bwd_iat_total', 'bwd_iat_mean', 'bwd_iat_std',
       'bwd_iat_max', 'bwd_iat_min', 'fwd_psh_flags', 'bwd_psh_flags',
       'fwd_urg_flags', 'bwd_urg_flags', 'fwd_header_length',
       'bwd_header_length', 'fwd_packets_s', 'bwd_packets_s',
       'min_packet_length', 'max_packet_length', 'packet_length_mean',
       'packet_length_std', 'packet_length_variance', 'fin_flag_count',
       'syn_flag_count', 'rst_flag_count', 'psh_flag_count', 'ack_flag_count',
       'urg_flag_count', 'cwe_flag_count', 'ece_flag_count', 'down_up_ratio',
       'average_packet_size', 'avg_fwd_segment_size', 'avg_bwd_segment_size',
       'fwd_header_length_1', 'fwd_avg_bytes_bulk', 'fwd_avg_packets_bulk',
       'fwd_avg_bulk_rate', 'bwd_avg_bytes_bulk', 'bwd_avg_packets_bulk',
       'bwd_avg_bulk_rate', 'subflow_fwd_packets', 'subflow_fwd_bytes',
       'subflow_bwd_packets', 'subflow_bwd_bytes', 'init_win_bytes_forward',
       'init_win_bytes_backward', 'act_data_pkt_fwd', 'min_seg_size_forward',
       'active_mean', 'active_std', 'active_max', 'active_min', 'idle_mean',
       'idle_std', 'idle_max', 'idle_min', 'label'],
      dtype='object')
```

Figure 4: This code lists and normalizes column names by removing spaces and special characters, making them easier to work with.

Step 4: Column names in the dataset are normalized for consistency. The code removes spaces and special characters while converting names to lowercase using methods like strip(), replace(), and lower(). This ensures column names are easier to work with during analysis and model building.

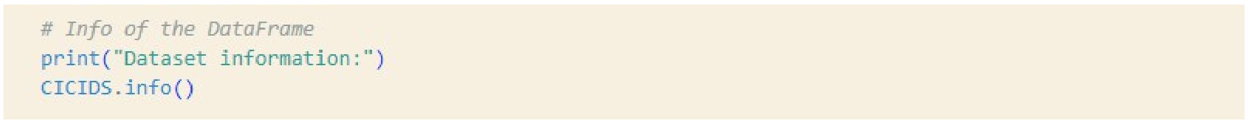
Component	Description
CICIDS.columns.tolist()	Lists all column names in the dataset.
strip().replace().lower()	Normalizes column names by removing spaces, special characters, and converting to lowercase.
CICIDS.columns	Updates the dataset with normalized column names.
print()	Displays original and updated column names.



```python
# Data types of each column
print("Data types of columns:")
print(CICIDS.dtypes)
```

Figure 5: This code displays the data types of all columns in the dataset to identify numeric, categorical, or other data types.

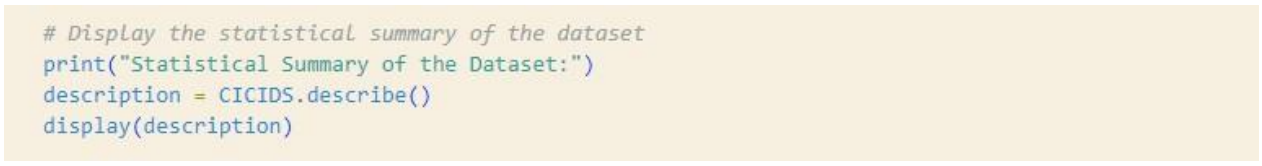
Step 5: Here in this step the data types of all columns are identified such that the CICIDS.dtypes function will list the column types and categorize these as numeric, categorical, or otherwise. This step will help the next preprocessing steps.



```python
# Info of the DataFrame
print("Dataset information:")
CICIDS.info()
```

Figure 6: This code summarizes the dataset's structure, including column details, non-null entries, and data types.

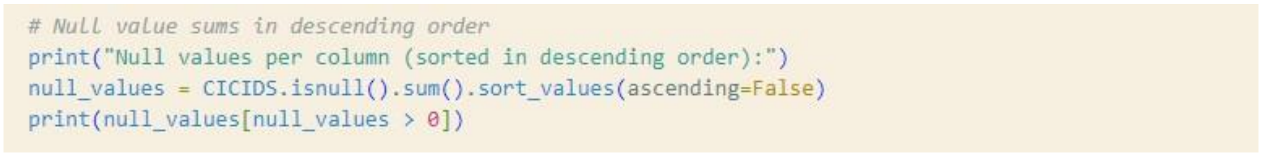
Step 6: This step shows the dataset's structure, including column details, non-null entries, and data types, using CICIDS.info() and it helps in understanding the dataset's data quality.



```python
# Display the statistical summary of the dataset
print("Statistical Summary of the Dataset:")
description = CICIDS.describe()
display(description)
```

Figure 7: This code provides a statistical summary of the numeric columns in the dataset, showing key metrics like mean, standard deviation, and range.

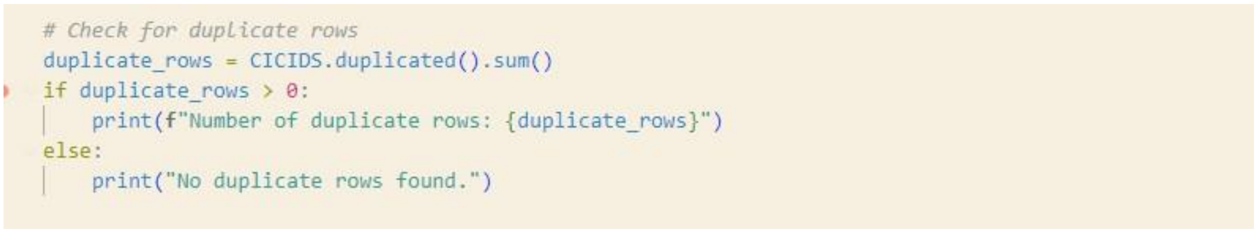
Step 7: This step is a statistical summary of these numeric columns and it is generated using CICIDS.describe(). Key metrics like count, mean, standard deviation, minimum and maximum values are also shown.



```python
# Null value sums in descending order
print("Null values per column (sorted in descending order):")
null_values = CICIDS.isnull().sum().sort_values(ascending=False)
print(null_values[null_values > 0])
```

Figure 8: This code identifies columns with null values and displays their counts in descending order for easier analysis.

Step 8: This step shows these columns with null values and counts them and this code sorts these columns in descending order of null counts and filters out columns with no missing values.

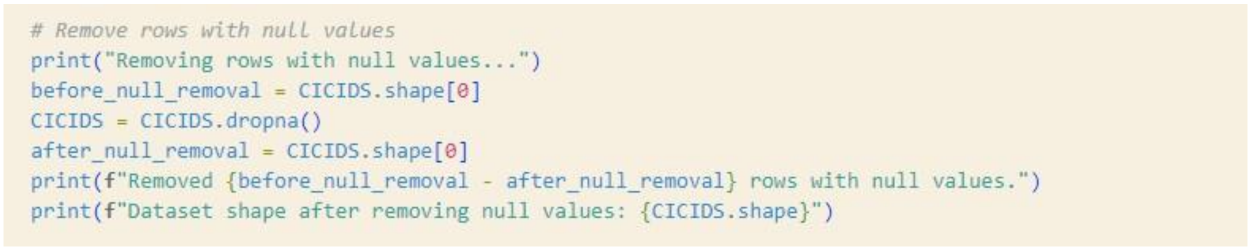


```python
# Check for duplicate rows
duplicate_rows = CICIDS.duplicated().sum()
if duplicate_rows > 0:
    print(f"Number of duplicate rows: {duplicate_rows}")
else:
    print("No duplicate rows found.")
```

Figure 9: This code checks for duplicate rows in the dataset and reports their count, if any exist.

Step 9: The dataset is checked for duplicate rows using CICIDS.duplicated(). Duplicate entries, if any, are counted and reported to assess the dataset's uniqueness and quality.

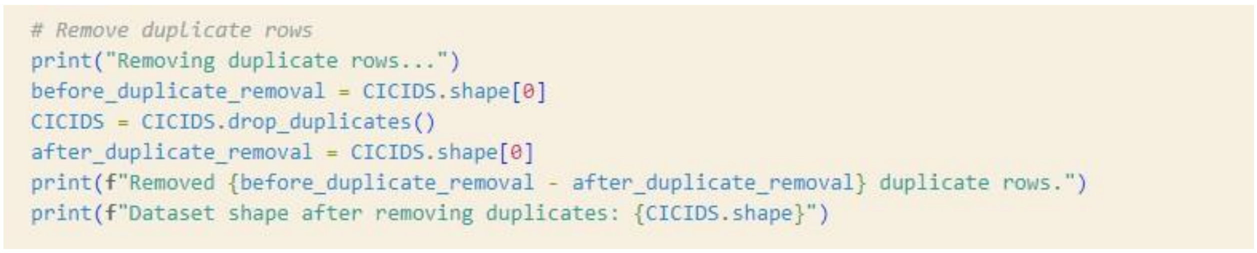
3.4. Data Cleaning



```python
# Remove rows with null values
print("Removing rows with null values...")
before_null_removal = CICIDS.shape[0]
CICIDS = CICIDS.dropna()
after_null_removal = CICIDS.shape[0]
print(f"Removed {before_null_removal - after_null_removal} rows with null values.")
print(f"Dataset shape after removing null values: {CICIDS.shape}")
```

Figure 10: This code removes rows with null values and updates the dataset, reporting the number of rows removed and the new shape.

Step 10: Rows containing null values are removed to ensure data consistency. The code calculates the number of rows removed and updates the dataset's shape, reporting the changes for transparency.

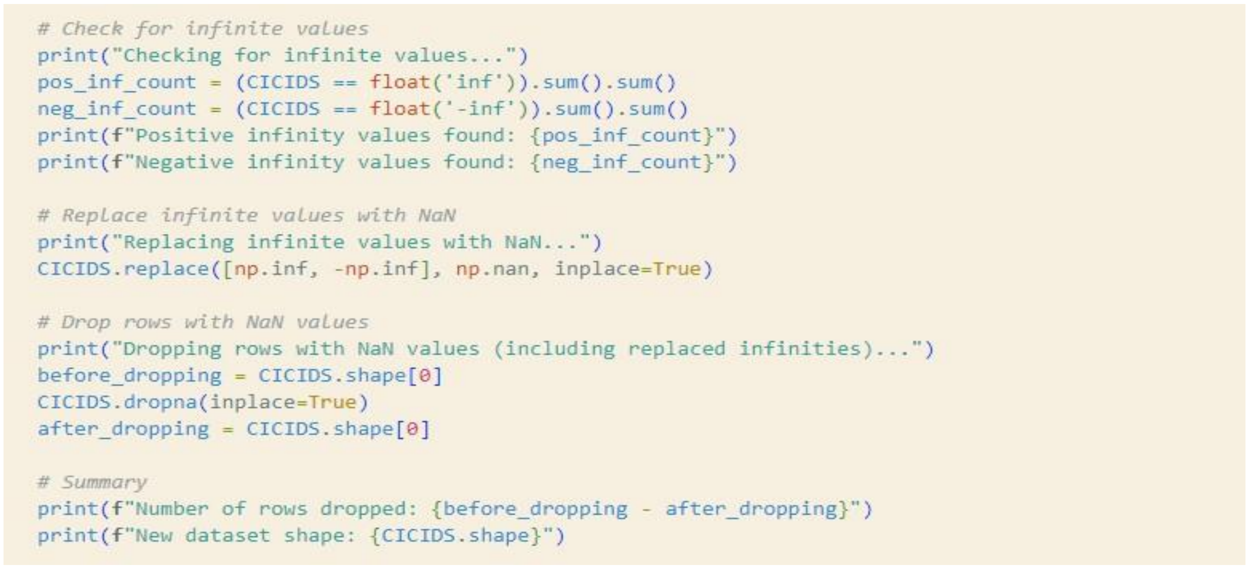


```python
# Remove duplicate rows
print("Removing duplicate rows...")
before_duplicate_removal = CICIDS.shape[0]
CICIDS = CICIDS.drop_duplicates()
after_duplicate_removal = CICIDS.shape[0]
print(f"Removed {before_duplicate_removal - after_duplicate_removal} duplicate rows.")
print(f"Dataset shape after removing duplicates: {CICIDS.shape}")
```

Figure 11: This code removes duplicate rows from the dataset, updating its shape and reporting the number of duplicates removed.

Step 11: Duplicate rows are removed to enhance data quality. The code calculates the number of duplicates removed and updates the dataset's shape, ensuring an accurate dataset.

Component	Description
CICIDS.drop_duplicates( )	Removes duplicate rows from the dataset.
CICIDS.shape[0]	Retrieves the number of rows before and after duplicate removal.
before_duplicate_remova l - after_duplicate_removal	Calculates the number of rows removed.
print()	Displays the number of removed duplicates and the updated dataset shape.

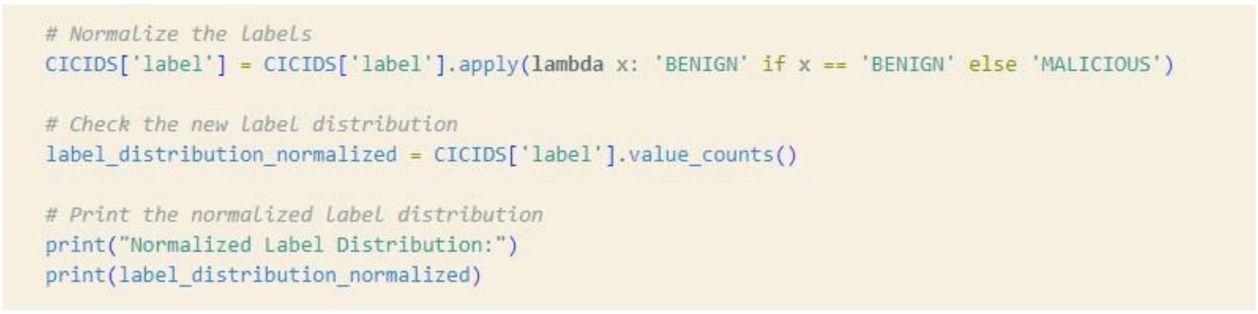


```python
# Check for infinite values
print("Checking for infinite values...")
pos_inf_count = (CICIDS == float('inf')).sum().sum()
neg_inf_count = (CICIDS == float('-inf')).sum().sum()
print(f"Positive infinity values found: {pos_inf_count}")
print(f"Negative infinity values found: {neg_inf_count}")

# Replace infinite values with NaN
print("Replacing infinite values with NaN...")
CICIDS.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values
print("Dropping rows with NaN values (including replaced infinities)...")
before_dropping = CICIDS.shape[0]
CICIDS.dropna(inplace=True)
after_dropping = CICIDS.shape[0]

# Summary
print(f"Number of rows dropped: {before_dropping - after_dropping}")
print(f"New dataset shape: {CICIDS.shape}")
```

Figure 12: This code identifies and replaces infinite values in the dataset with NaN, then removes rows containing these values to ensure data consistency.

Step 12: This step identifies and replaces infinite values with NaN, then removes rows containing these values. This ensures numerical stability during analysis and modeling.

3.5. Label Normalization

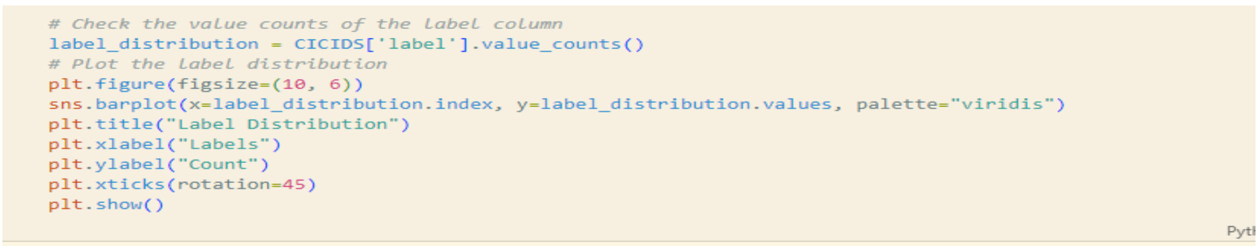


```python
# Normalize the labels
CICIDS['label'] = CICIDS['label'].apply(lambda x: 'BENIGN' if x == 'BENIGN' else 'MALICIOUS')

# Check the new label distribution
label_distribution_normalized = CICIDS['label'].value_counts()

# Print the normalized label distribution
print("Normalized Label Distribution:")
print(label_distribution_normalized)
```

Figure 13: This code standardizes the dataset labels into 'BENIGN' and 'MALICIOUS' categories and displays their distribution.

Step 13: Dataset labels are standardized into 'BENIGN' and 'MALICIOUS' categories using CICIDS['label'].apply(). The distribution of these labels is displayed to check class balance.

3.6. EDA - Label Distribution



```python
# Check the value counts of the label column
label_distribution = CICIDS['label'].value_counts()
# Plot the label distribution
plt.figure(figsize=(10, 6))
sns.barplot(x=label_distribution.index, y=label_distribution.values, palette="viridis")
plt.title("Label Distribution")
plt.xlabel("Labels")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()
```

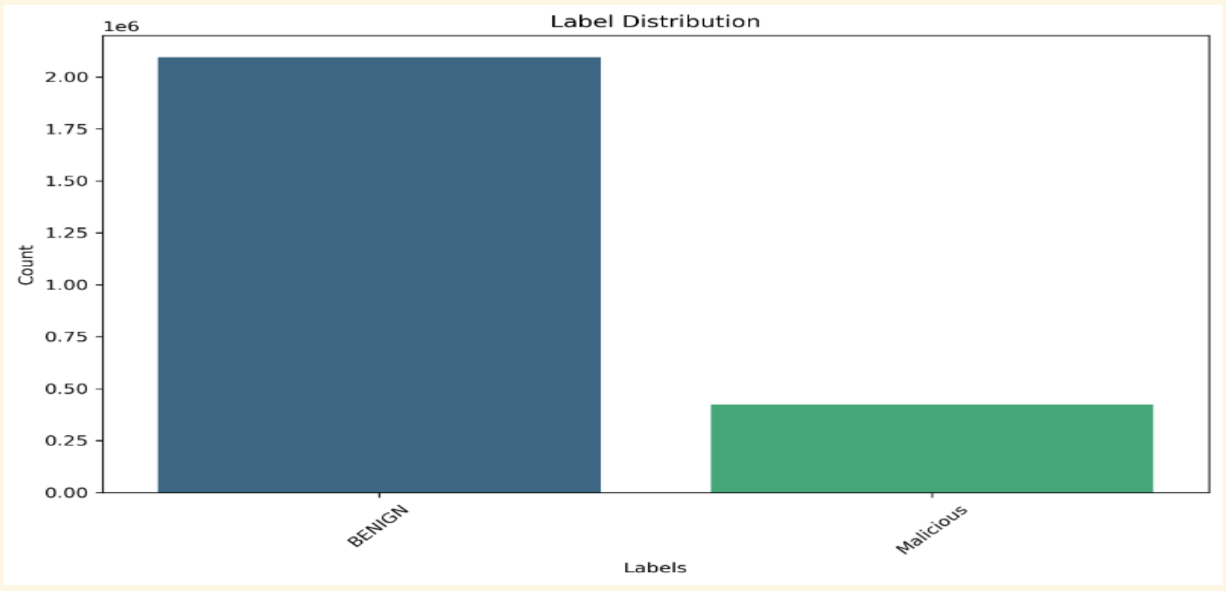


Figure 14: This code visualizes the distribution of 'BENIGN' and 'MALICIOUS' labels in the dataset using a bar plot.

Step 14: A bar plot visualizing the distribution of 'BENIGN' and 'MALICIOUS' labels is created. This aids in understanding the prevalence of each label in the dataset.

3.7. Flow-Based Features

```python
flow_features = ['flow_duration', 'flow_bytes_s', 'flow_packets_s',
                 'flow_iat_mean', 'flow_iat_std', 'flow_iat_max', 'flow_iat_min']

# Plot histograms for flow-based features
CICIDS[flow_features].hist(figsize=(20, 15), bins=30)
plt.suptitle("Histograms of Flow-Based Features")
plt.tight_layout()
plt.show()
```

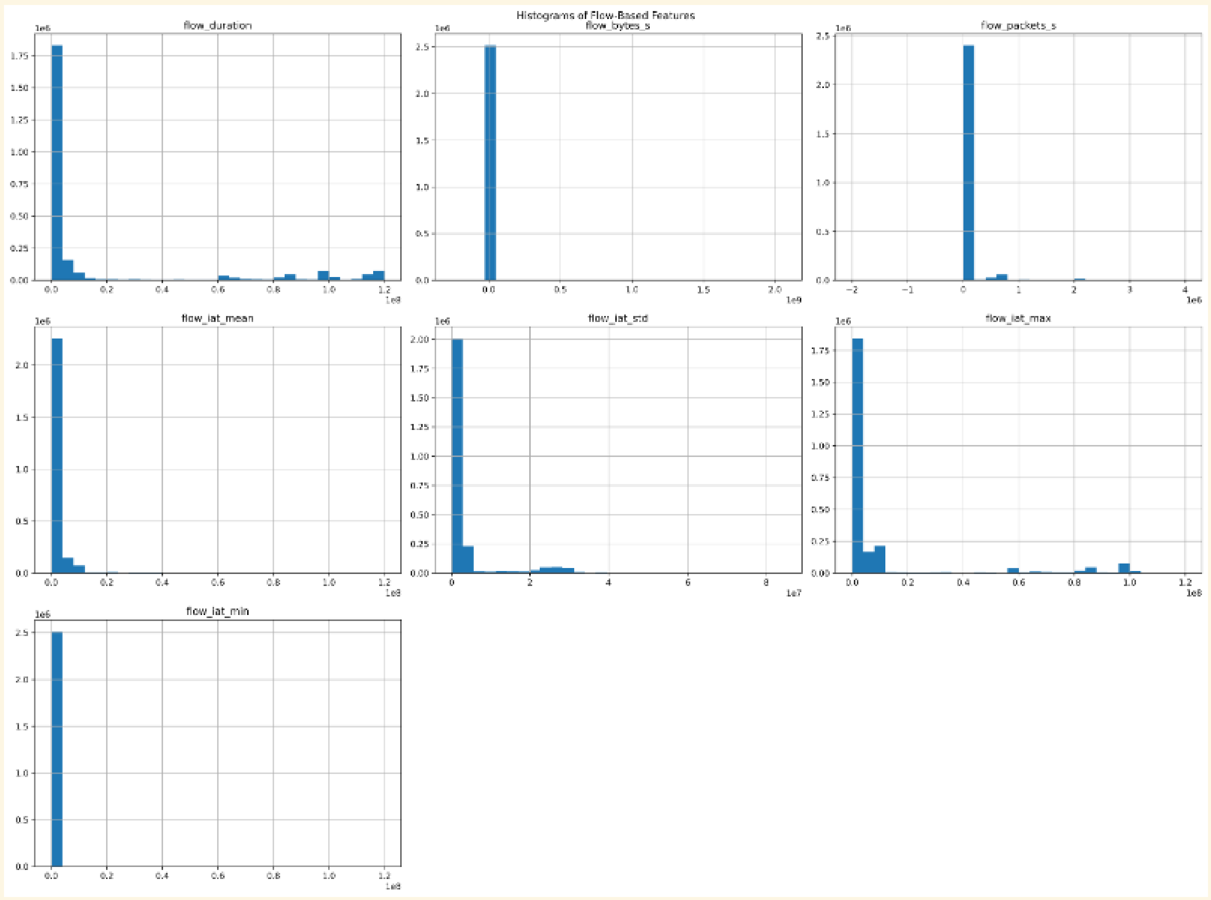
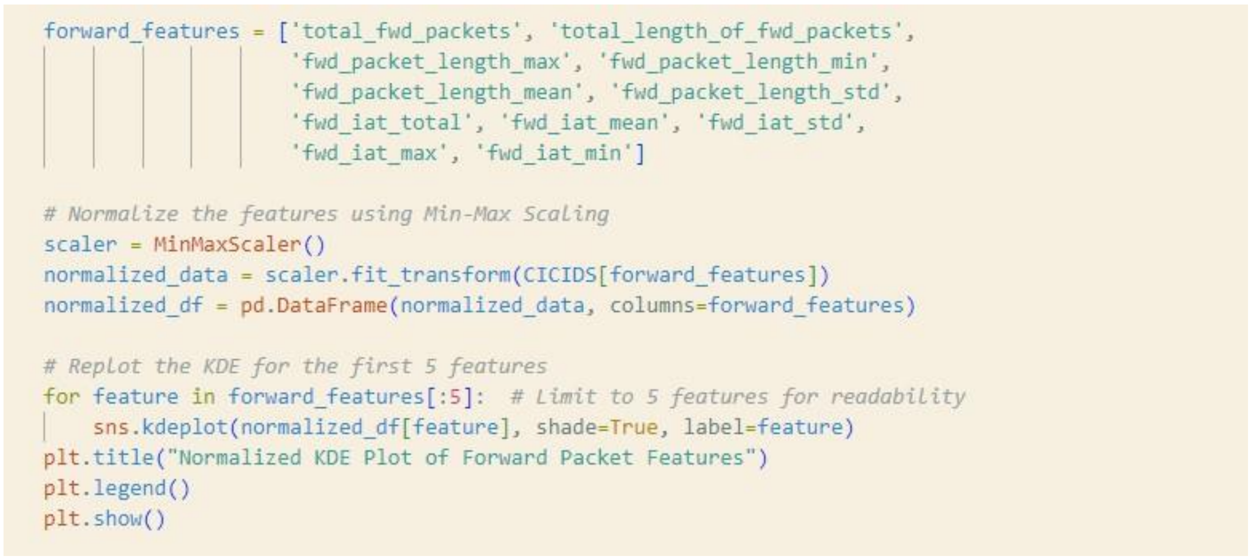


Figure 15: This code generates histograms to visualize the distribution of flow-based features in the dataset.

Step 15: Histograms for flow-based features are generated to visualize their distributions. The plots provide insights into the range and frequency of feature values.

3.8. Forward Packet Features



```python
forward_features = ['total_fwd_packets', 'total_length_of_fwd_packets',
                    'fwd_packet_length_max', 'fwd_packet_length_min',
                    'fwd_packet_length_mean', 'fwd_packet_length_std',
                    'fwd_iat_total', 'fwd_iat_mean', 'fwd_iat_std',
                    'fwd_iat_max', 'fwd_iat_min']

# Normalize the features using Min-Max Scaling
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(CICIDS[forward_features])
normalized_df = pd.DataFrame(normalized_data, columns=forward_features)

# Replot the KDE for the first 5 features
for feature in forward_features[:5]:  # Limit to 5 features for readability
    sns.kdeplot(normalized_df[feature], shade=True, label=feature)
plt.title("Normalized KDE Plot of Forward Packet Features")
plt.legend()
plt.show()
```

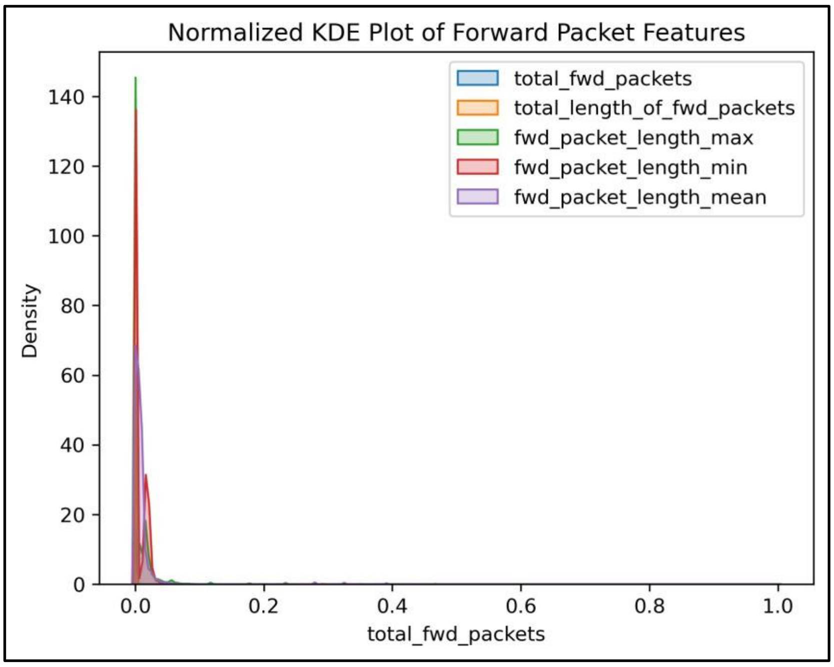


Figure 16: This code normalizes forward packet features using Min-Max Scaling and visualizes their distributions using KDE plots for the first five features.

3.9. Backward Packet Features

```python
backward_features = ['total_backward_packets', 'total_length_of_bwd_packets',
                     'bwd_packet_length_max', 'bwd_packet_length_min',
                     'bwd_packet_length_mean', 'bwd_packet_length_std',
                     'bwd_iat_total', 'bwd_iat_mean', 'bwd_iat_std']

# Box Plots
plt.figure(figsize=(12, 6))
sns.boxplot(data=CICIDS[backward_features], orient='h')
plt.title("Box Plot of Backward Packet Features")
plt.xlabel("Values")
plt.ylabel("Features")
plt.show()
```

Python

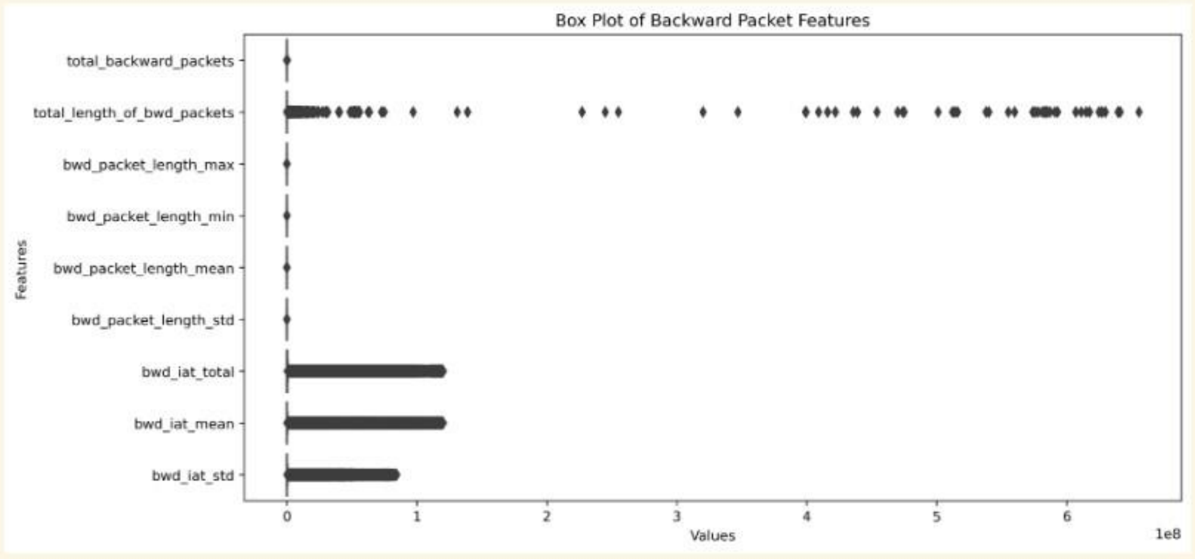


Figure 17: This code creates box plots to visualize the distribution and outliers of backward packet features in the dataset.

3.10. Packet Feature Length

```python
packet_length_features = ['min_packet_length', 'max_packet_length',
                          'packet_length_mean', 'packet_length_std',
                          'packet_length_variance', 'average_packet_size']

# Pairplot
sns.pairplot(CICIDS[packet_length_features])
plt.suptitle("Pairplot of Packet Length Features", y=1.02)
plt.show()
```

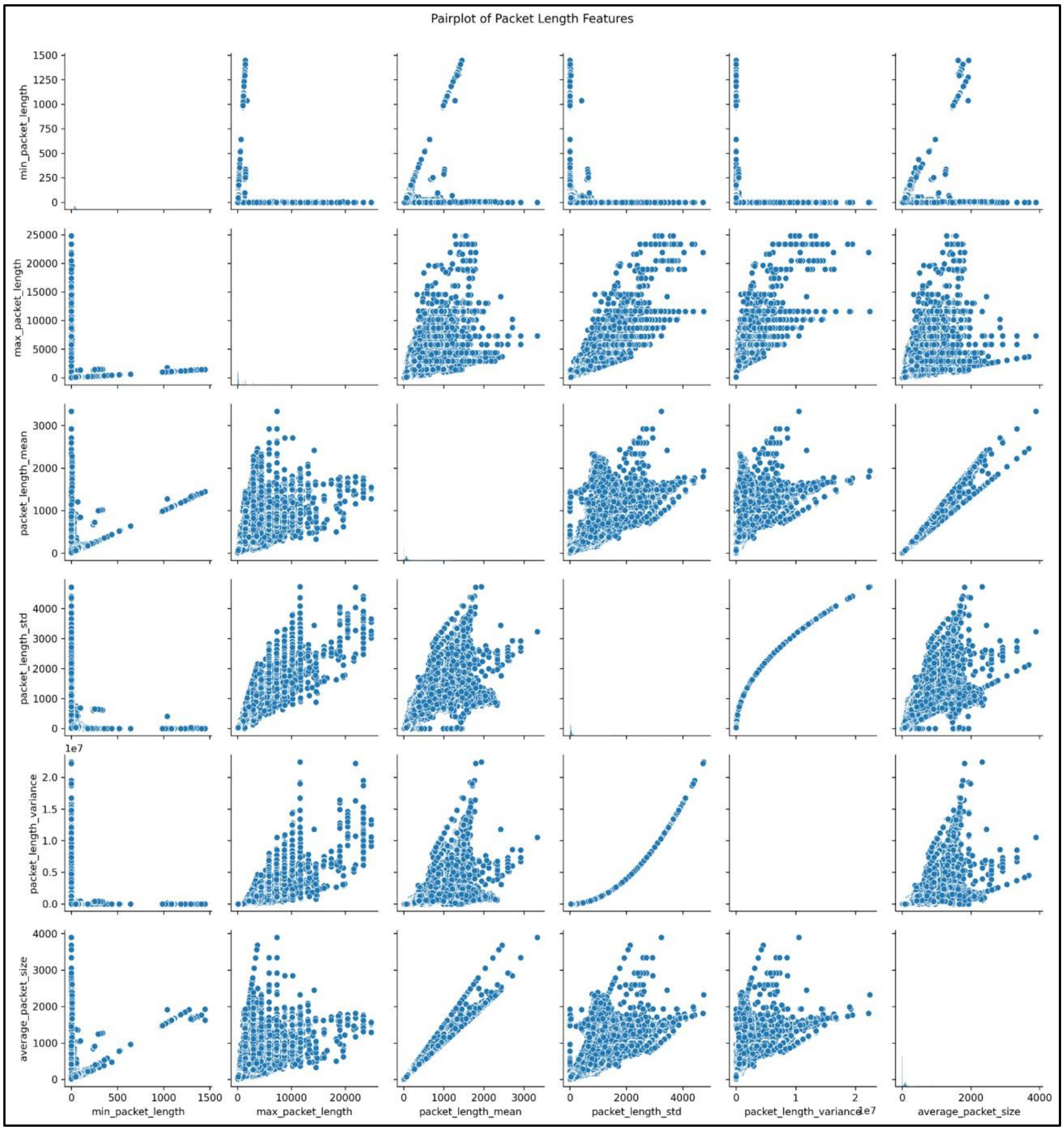


Figure 18: This code generates a pairplot to visualize relationships and distributions among packet length features in the dataset.

3.11. Flag Features

```python
flag_features = ['fin_flag_count', 'syn_flag_count', 'rst_flag_count',
                 'psh_flag_count', 'ack_flag_count', 'urg_flag_count',
                 'cwe_flag_count', 'ece_flag_count']

# Bar Plot of Flag Counts
flag_sums = CICIDS[flag_features].sum()
flag_sums.plot(kind='bar', figsize=(10, 5))
plt.title("Sum of Flag Features")
plt.xlabel("Flags")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()
```

Python

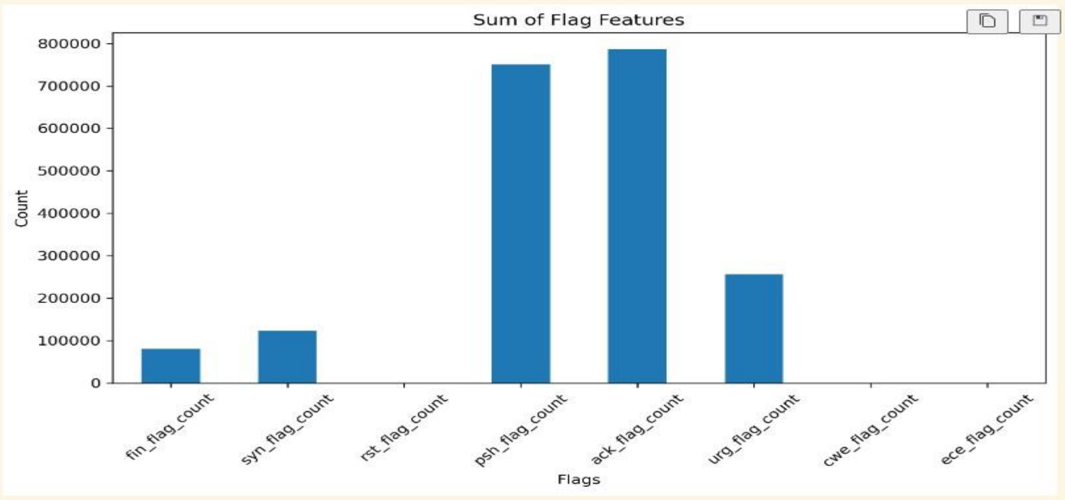
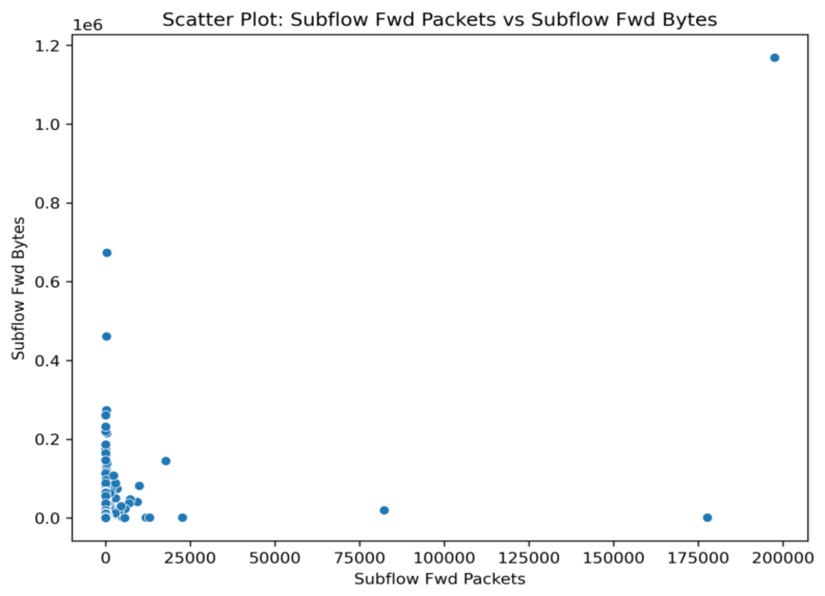


Figure 19: This code visualizes the total counts of flag features in the dataset using a bar plot.

3.12. Subway flows

```python
subflow_features = ['subflow_fwd_packets', 'subflow_fwd_bytes',
                    'subflow_bwd_packets', 'subflow_bwd_bytes']

# Select 10% of the dataset for faster computation
sampled_data = CICIDS[subflow_features].sample(frac=0.1, random_state=42)

# Scatter plot for two fwd features
plt.figure(figsize=(8, 6))
sns.scatterplot(x='subflow_fwd_packets', y='subflow_fwd_bytes', data=sampled_data)
plt.title("Scatter Plot: Subflow Fwd Packets vs Subflow Fwd Bytes")
plt.xlabel("Subflow Fwd Packets")
plt.ylabel("Subflow Fwd Bytes")
plt.show()

# Scatter plot for two bwd features
plt.figure(figsize=(8, 6))
sns.scatterplot(x='subflow_bwd_packets', y='subflow_bwd_bytes', data=sampled_data)
plt.title("Scatter Plot: Subflow Fwd Packets vs Subflow Fwd Bytes")
plt.xlabel("Subflow Fwd Packets")
plt.ylabel("Subflow Fwd Bytes")
plt.show()
```



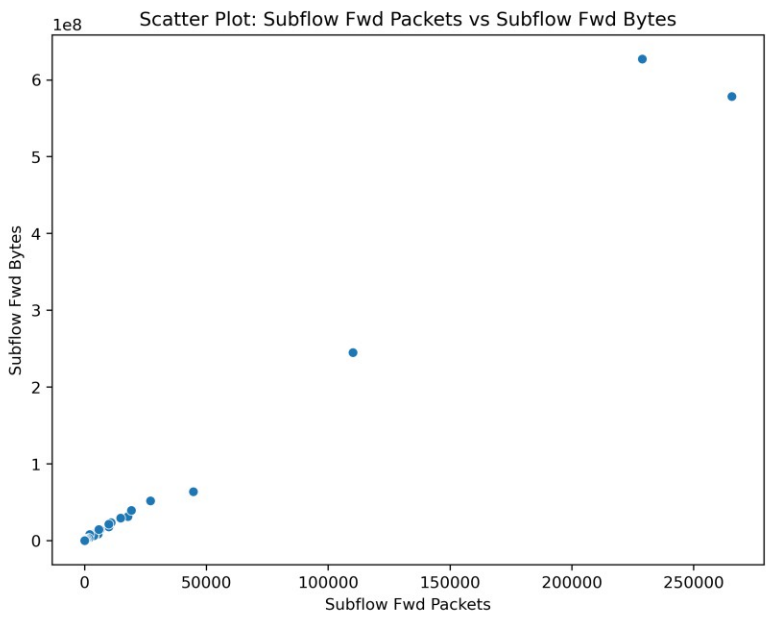


Figure 20: This code creates scatter plots to visualize the relationships between subflow forward and backward packet counts and byte counts.

Component	Description
subflow_features	List of subflow-related features for analysis.
CICIDS[subflow_features].sample(frac=0.1)	Samples 10% of the dataset for faster computation.
sns.scatterplot()	Creates scatter plots for visualizing relationships between subflow features.
x='subflow_fwd_packets', y='subflow_fwd_bytes'	Plots forward packet and byte relationships.

x='subflow_bwd_packets', y='subflow_bwd_bytes'	Plots backward packet and byte relationships.
plt.title(), plt.xlabel(), plt.ylabel()	Adds titles and axis labels to the scatter plots.
plt.show()	Displays the scatter plots.

3.13. Windows and Active/Idle Features

```python
window_features = ['init_win_bytes_forward', 'init_win_bytes_backward',
                   'act_data_pkt_fwd', 'min_seg_size_forward',
                   'active_mean', 'active_std', 'active_max',
                   'active_min', 'idle_mean', 'idle_std', 'idle_max', 'idle_min']

# Histograms for Active/Idle Features
CICIDS[window_features].hist(figsize=(20, 15), bins=30)
plt.suptitle("Histograms of Window and Active/Idle Features")
plt.tight_layout()
plt.show()
```

Python

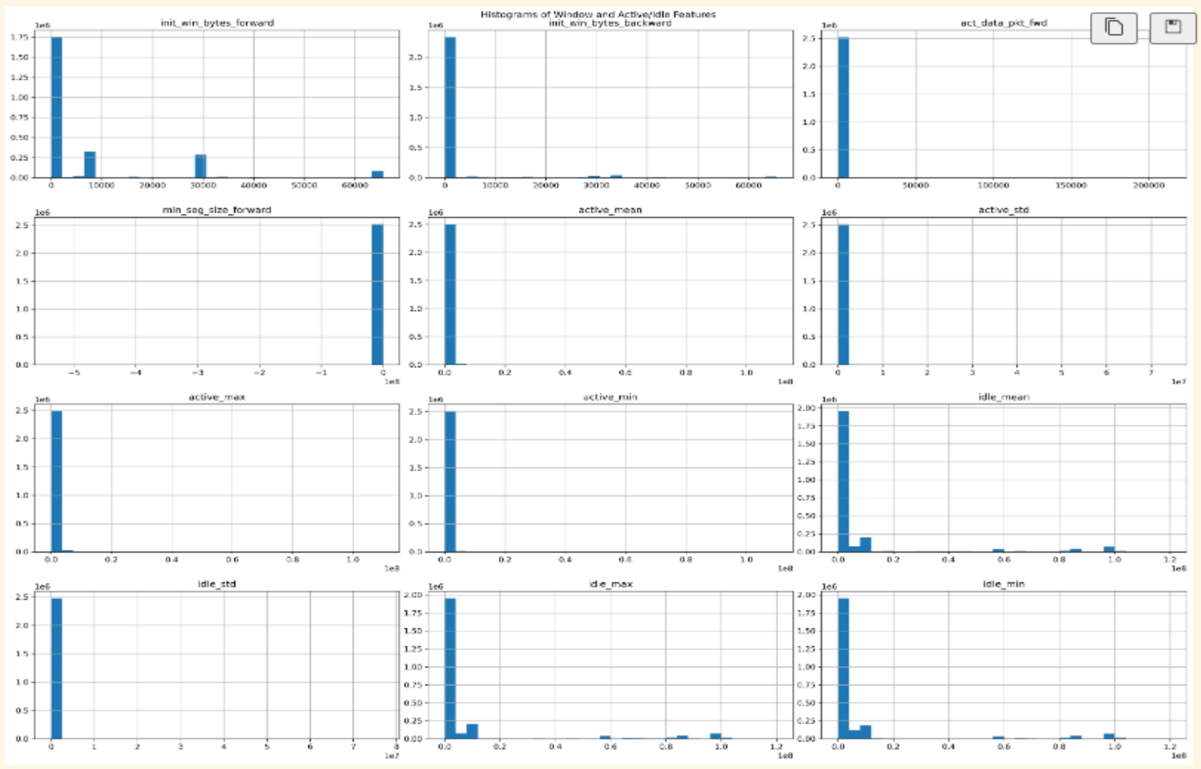


Figure 21: This code generates histograms to visualize the distribution of window size, active, and idle features in the dataset.

3.14. Correlation Matrix

```python
# Compute the correlation matrix
correlation_matrix = CICIDS.corr()

# Plot the heatmap
plt.figure(figsize=(20, 15))
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap for Numeric Features')
plt.show()
```

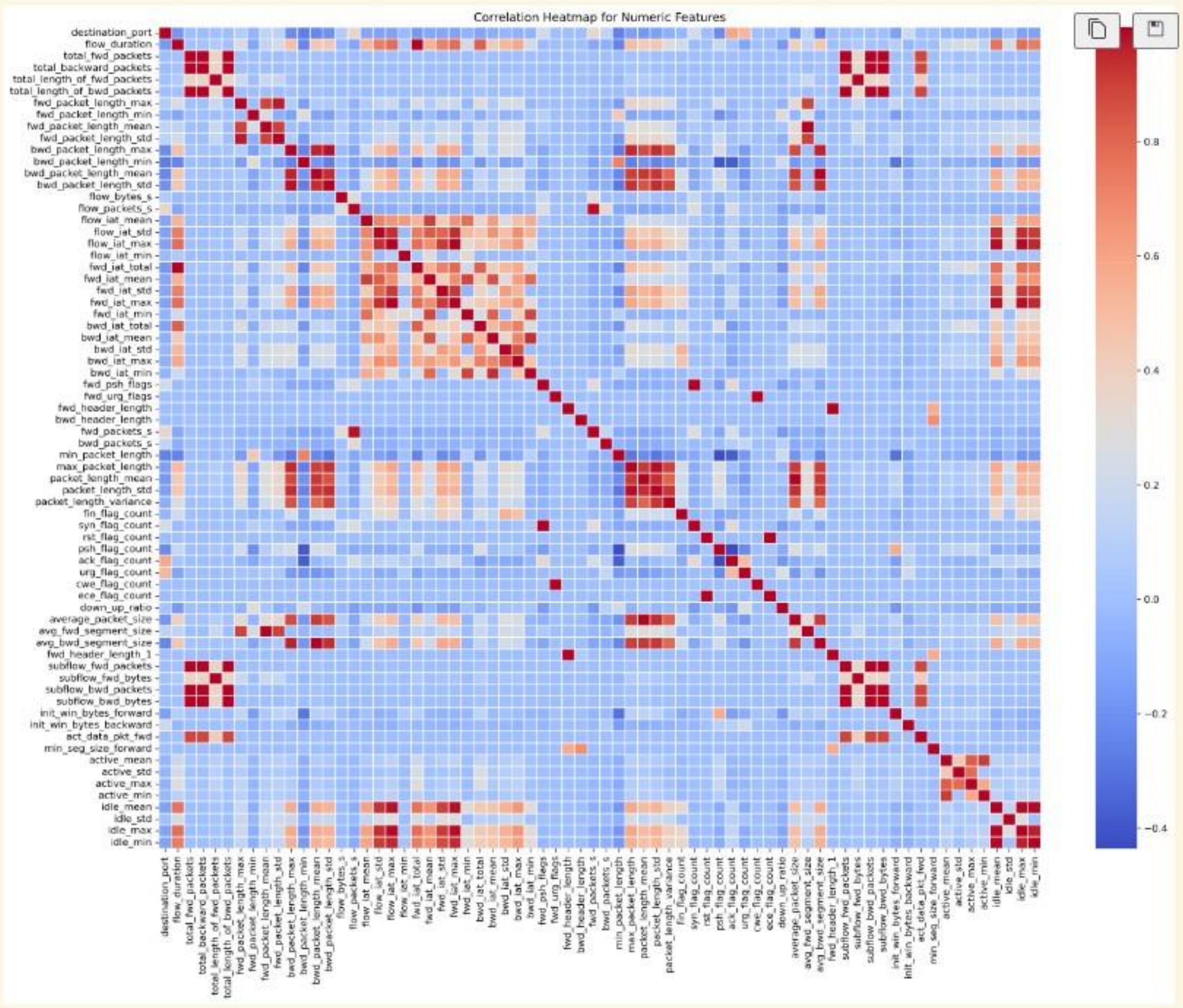
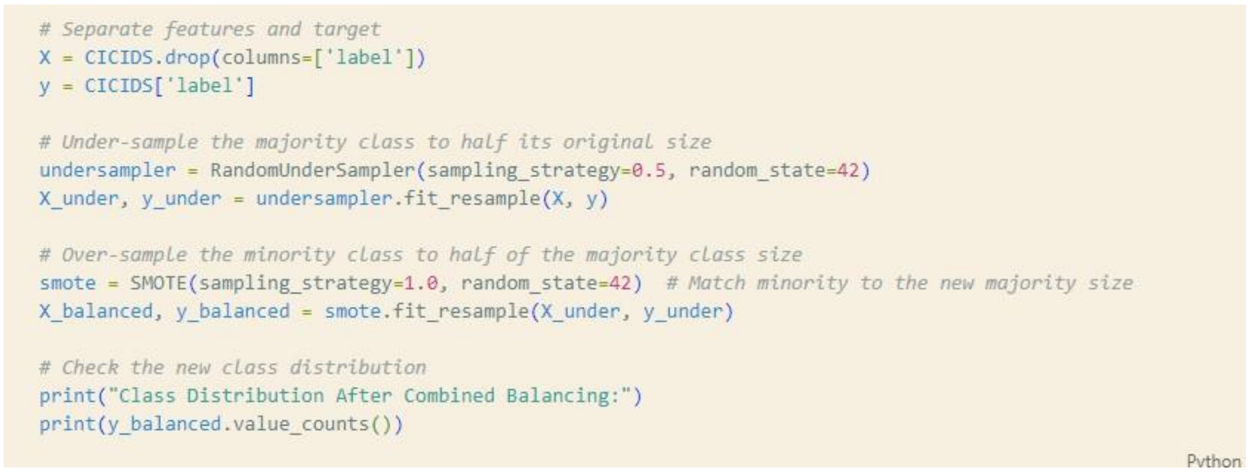


Figure 22: This code generates a heatmap to visualize the correlation between numeric features in the dataset.

3.15. Data Balancing

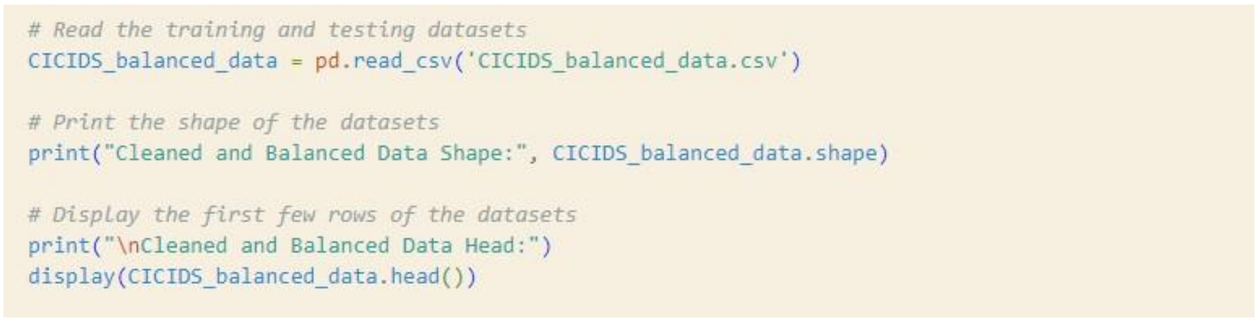


```python
# Separate features and target
X = CICIDS.drop(columns=['label'])
y = CICIDS['label']

# Under-sample the majority class to half its original size
undersampler = RandomUnderSampler(sampling_strategy=0.5, random_state=42)
X_under, y_under = undersampler.fit_resample(X, y)

# Over-sample the minority class to half of the majority class size
smote = SMOTE(sampling_strategy=1.0, random_state=42)  # Match minority to the new majority size
X_balanced, y_balanced = smote.fit_resample(X_under, y_under)

# Check the new class distribution
print("Class Distribution After Combined Balancing:")
print(y_balanced.value_counts())
```

Python

Figure 23: This code balances the dataset by under-sampling the majority class and over-sampling the minority class using SMOTE.

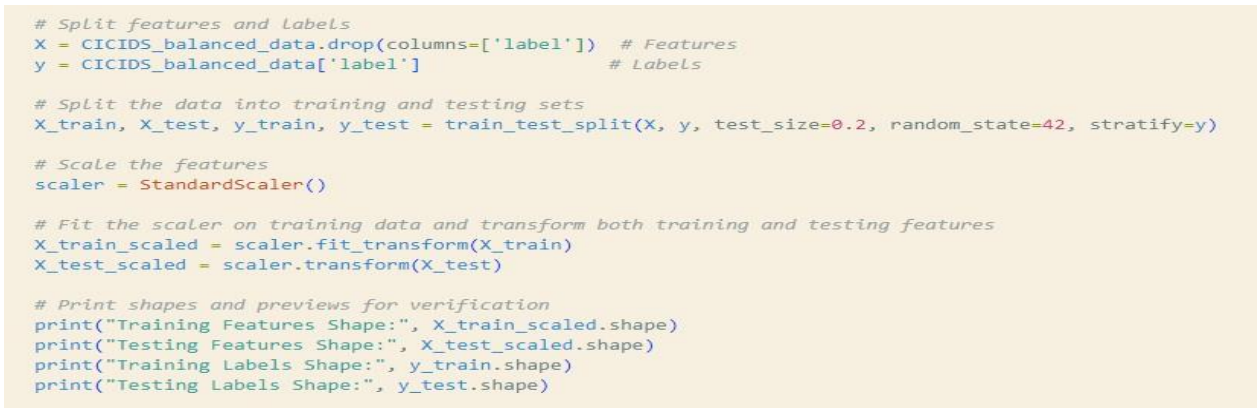
3.16. Loading the Cleaned and Balanced dataset



```python
# Read the training and testing datasets
CICIDS_balanced_data = pd.read_csv('CICIDS_balanced_data.csv')

# Print the shape of the datasets
print("Cleaned and Balanced Data Shape:", CICIDS_balanced_data.shape)

# Display the first few rows of the datasets
print("\nCleaned and Balanced Data Head:")
display(CICIDS_balanced_data.head())
```

Figure 24: This code loads the cleaned and balanced dataset from a CSV file and displays its shape and a preview of the first few rows.

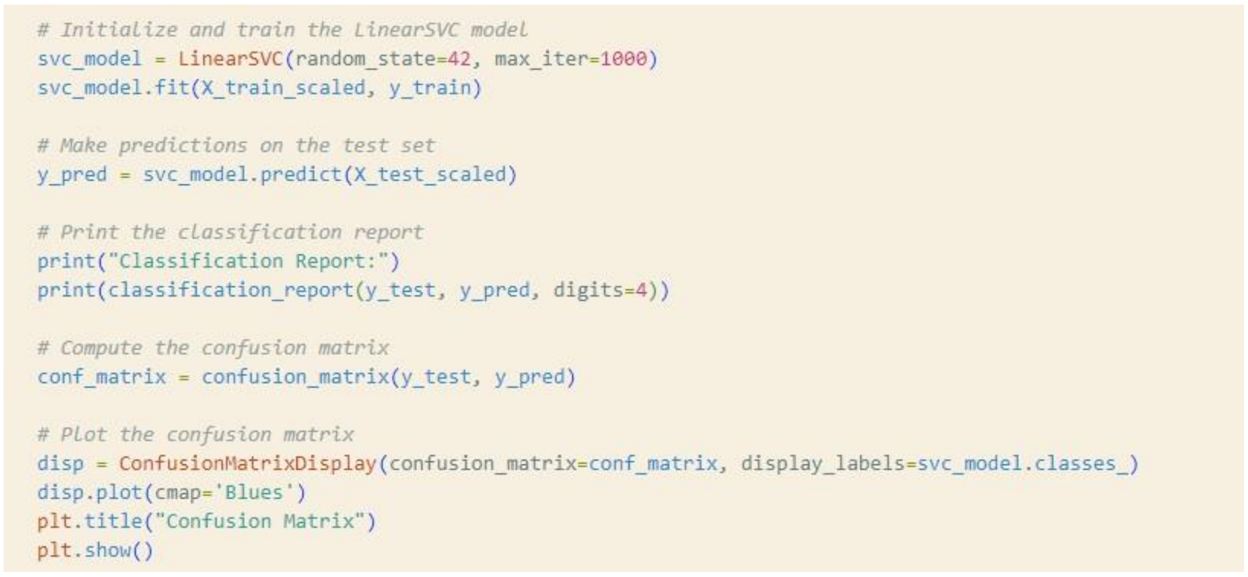
3.17. Data Scaling



```python
# Split features and labels
X = CICIDS_balanced_data.drop(columns=['label'])  # Features
y = CICIDS_balanced_data['label']                # Labels

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Scale the features
scaler = StandardScaler()

# Fit the scaler on training data and transform both training and testing features
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Print shapes and previews for verification
print("Training Features Shape:", X_train_scaled.shape)
print("Testing Features Shape:", X_test_scaled.shape)
print("Training Labels Shape:", y_train.shape)
print("Testing Labels Shape:", y_test.shape)
```

Figure 25: This code splits the balanced dataset into training and testing sets, scales the features, and verifies the shapes of the resulting datasets.

3.18. Data Modeling - SVM



```
# Initialize and train the LinearSVC model
svc_model = LinearSVC(random_state=42, max_iter=1000)
svc_model.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred = svc_model.predict(X_test_scaled)

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=svc_model.classes_)
disp.plot(cmap='Blues')
plt.title("Confusion Matrix")
plt.show()
```

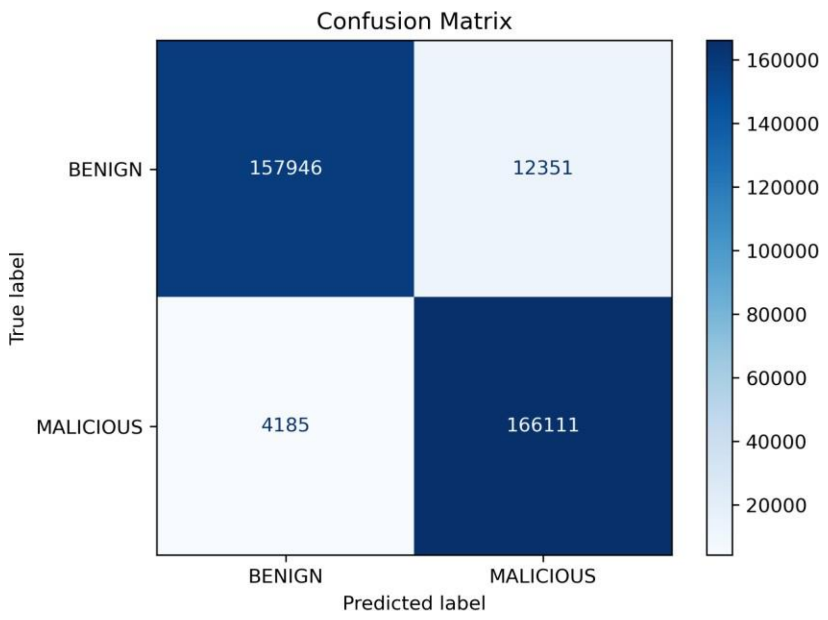
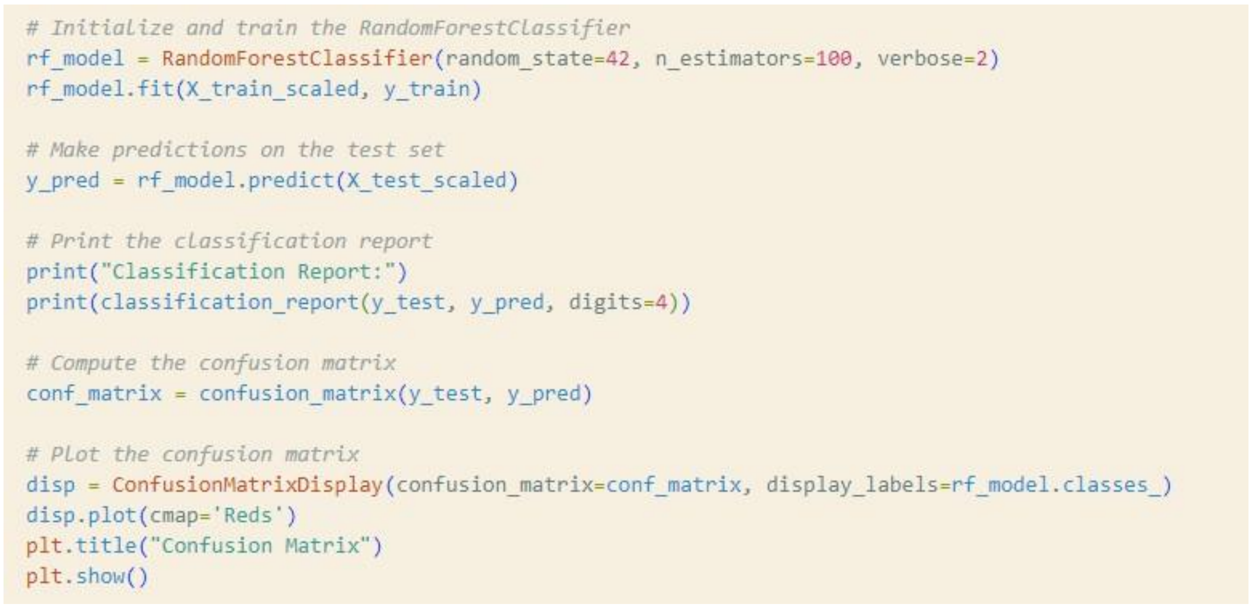


Figure 26: This code trains a Linear Support Vector Classifier, evaluates its performance with a classification report, and visualizes the confusion matrix.

Step 16: A Linear Support Vector Classifier is trained, evaluated, and its performance metrics are displayed. The confusion matrix is visualized for a detailed assessment.

3.19. Random Forest



```python
# Initialize and train the RandomForestClassifier
rf_model = RandomForestClassifier(random_state=42, n_estimators=100, verbose=2)
rf_model.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred = rf_model.predict(X_test_scaled)

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=rf_model.classes_)
disp.plot(cmap='Reds')
plt.title("Confusion Matrix")
plt.show()
```

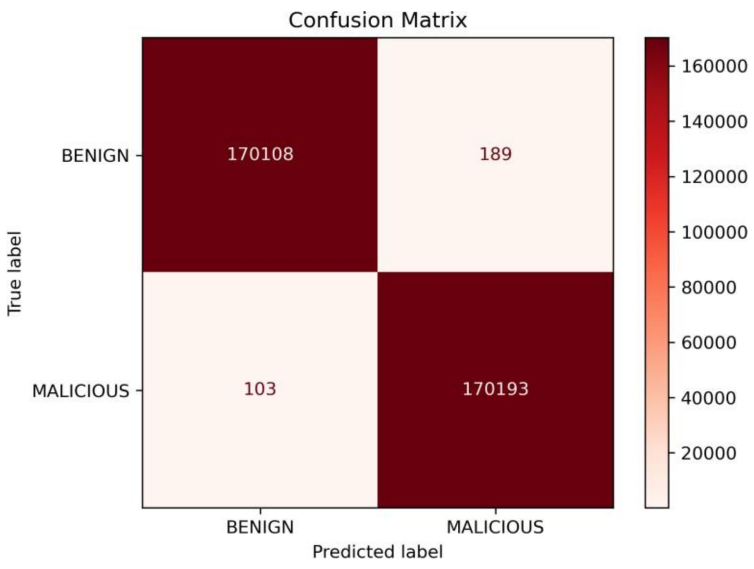


Figure 27: This code trains a Random Forest Classifier, evaluates its performance using a classification report, and visualizes the confusion matrix.

Step 17: A Random Forest Classifier is trained and evaluated. Its performance metrics and confusion matrix provide insights into model effectiveness.

3.20. XGBoost

```python
# Encode the labels
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Train the XGBoost model
xgb_model = XGBClassifier(random_state=42, verbosity=1)
xgb_model.fit(X_train_scaled, y_train_encoded)

# Make predictions on the test set
y_pred_encoded = xgb_model.predict(X_test_scaled)

# Decode the predicted labels and actual labels for evaluation
y_pred = label_encoder.inverse_transform(y_pred_encoded)
y_test_decoded = label_encoder.inverse_transform(y_test_encoded)

# Print the classification report
print("Classification Report:")
print(classification_report(y_test_decoded, y_pred, digits=4))

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test_decoded, y_pred)

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=label_encoder.classes_)
disp.plot(cmap='Greens')
plt.title("Confusion Matrix")
plt.show()
```

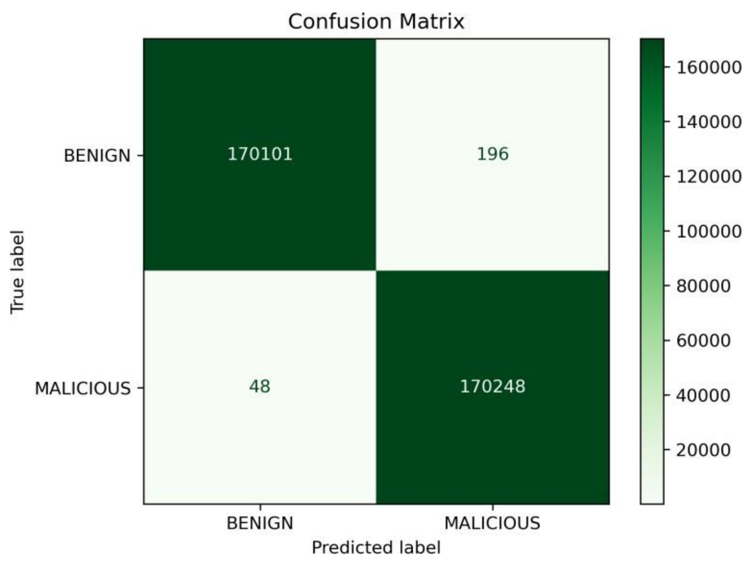
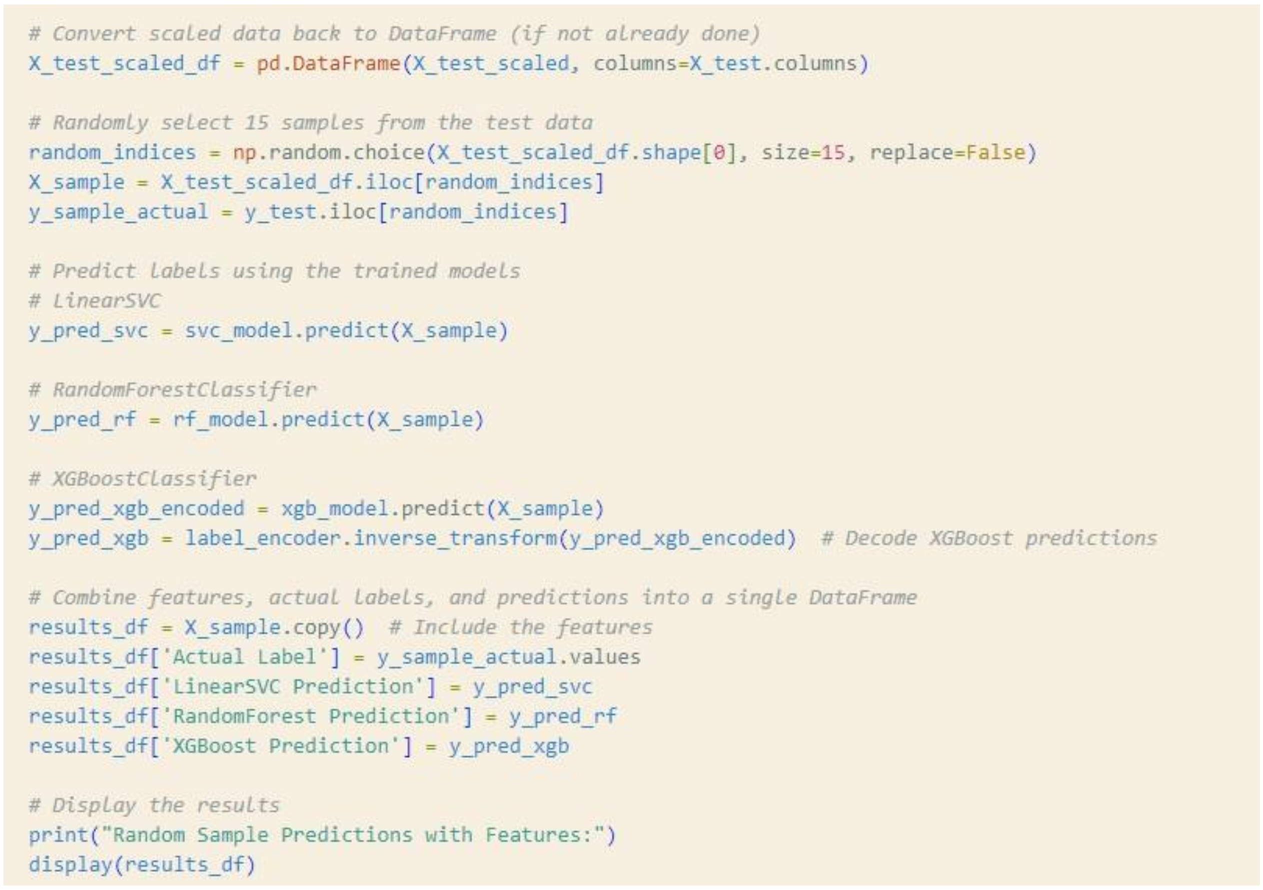


Figure 28: This code trains a Random Forest Classifier, evaluates its performance using a classification report, and visualizes the confusion matrix.

Step 18: An XGBoost Classifier is trained and evaluated similarly. This step highlights the model's suitability for the task.

3.21. Model Comparison

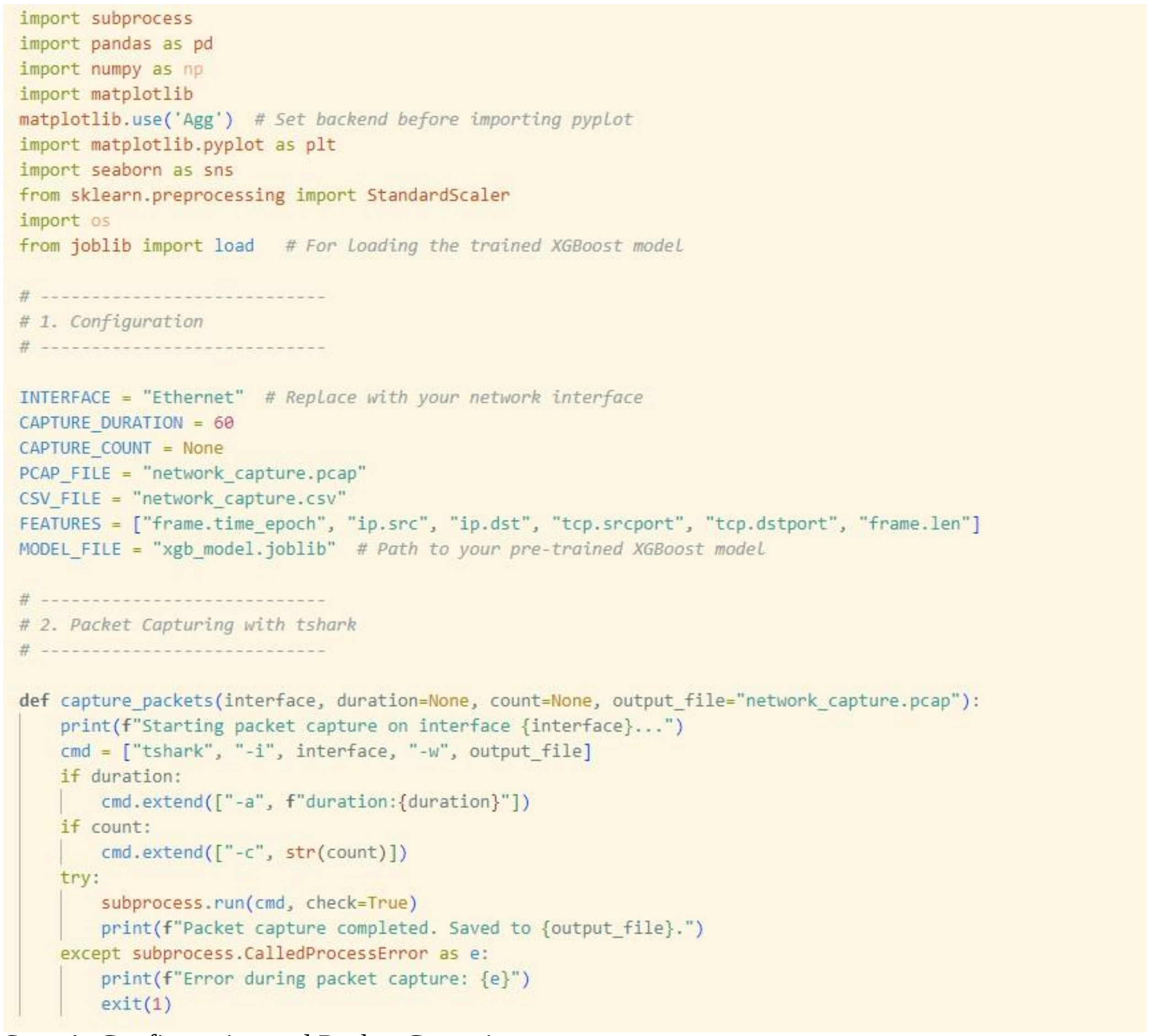


```python
# Convert scaled data back to DataFrame (if not already done)
X_test_scaled_df = pd.DataFrame(X_test_scaled, columns=X_test.columns)

# Randomly select 15 samples from the test data
random_indices = np.random.choice(X_test_scaled_df.shape[0], size=15, replace=False)
X_sample = X_test_scaled_df.iloc[random_indices]
y_sample_actual = y_test.iloc[random_indices]

# Predict labels using the trained models
# LinearSVC
y_pred_svc = svc_model.predict(X_sample)

# RandomForestClassifier
y_pred_rf = rf_model.predict(X_sample)

# XGBoostClassifier
y_pred_xgb_encoded = xgb_model.predict(X_sample)
y_pred_xgb = label_encoder.inverse_transform(y_pred_xgb_encoded)  # Decode XGBoost predictions

# Combine features, actual labels, and predictions into a single DataFrame
results_df = X_sample.copy()  # Include the features
results_df['Actual Label'] = y_sample_actual.values
results_df['LinearSVC Prediction'] = y_pred_svc
results_df['RandomForest Prediction'] = y_pred_rf
results_df['XGBoost Prediction'] = y_pred_xgb

# Display the results
print("Random Sample Predictions with Features:")
display(results_df)
```

Figure 29: This code predicts labels for a random sample of test data using LinearSVC, Random Forest, and XGBoost models, displaying the actual and predicted labels with features.

Step 19: A comparison of predictions from LinearSVC, Random Forest, and XGBoost models is presented. Actual and predicted labels are displayed for selected samples, showcasing model performance.

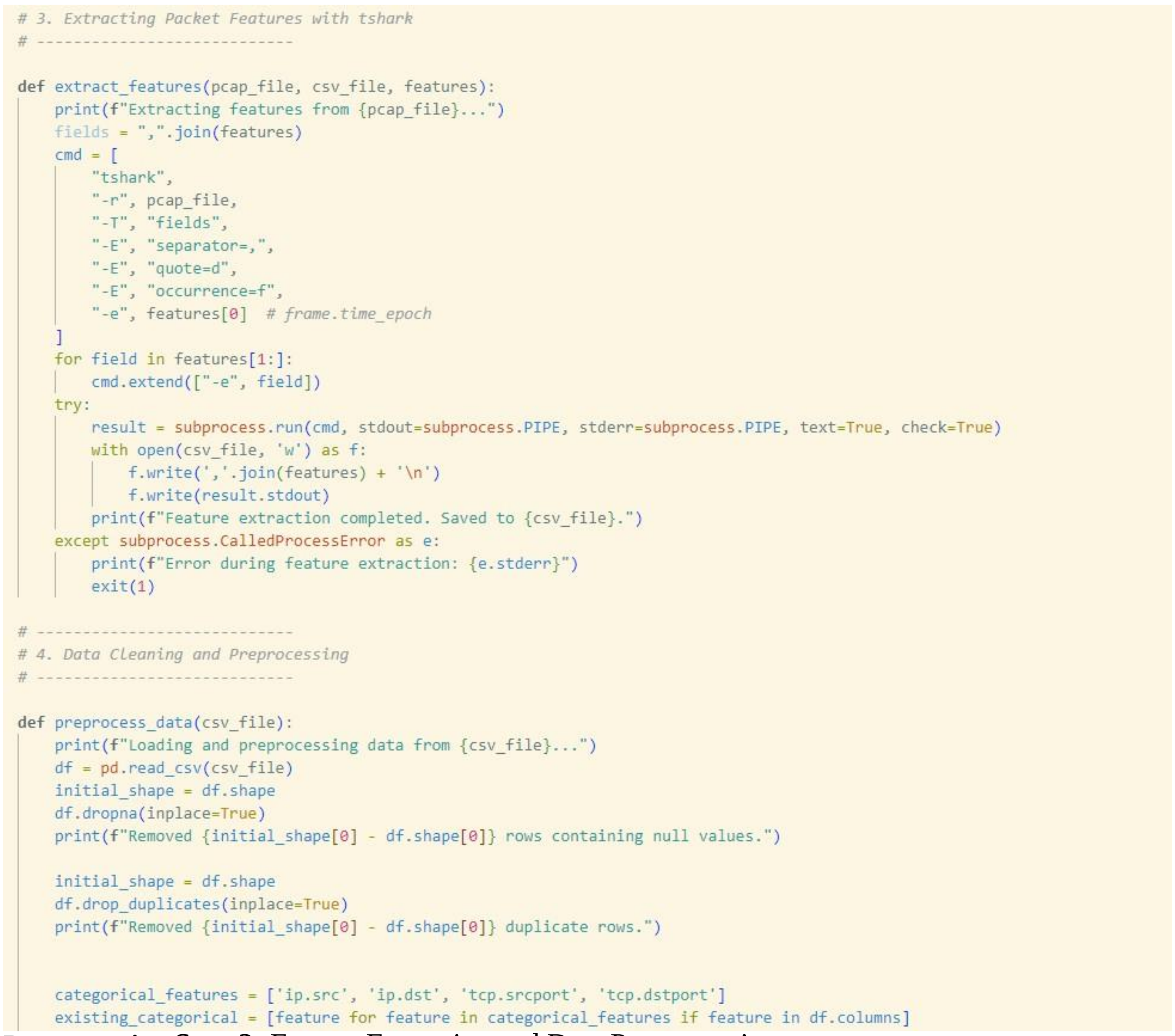
3.22. DPI Inclusion



```python
import subprocess
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('Agg')  # Set backend before importing pyplot
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
import os
from joblib import load   # For loading the trained XGBoost model

# ----------------------------
# 1. Configuration
# ----------------------------

INTERFACE = "Ethernet"  # Replace with your network interface
CAPTURE_DURATION = 60
CAPTURE_COUNT = None
PCAP_FILE = "network_capture.pcap"
CSV_FILE = "network_capture.csv"
FEATURES = ["frame.time_epoch", "ip.src", "ip.dst", "tcp.srcport", "tcp.dstport", "frame.len"]
MODEL_FILE = "xgb_model.joblib"  # Path to your pre-trained XGBoost model

# ----------------------------
# 2. Packet Capturing with tshark
# ----------------------------

def capture_packets(interface, duration=None, count=None, output_file="network_capture.pcap"):
    print(f"Starting packet capture on interface {interface}...")
    cmd = ["tshark", "-i", interface, "-w", output_file]
    if duration:
        cmd.extend(["-a", f"duration:{duration}"])
    if count:
        cmd.extend(["-c", str(count)])
    try:
        subprocess.run(cmd, check=True)
        print(f"Packet capture completed. Saved to {output_file}.")
    except subprocess.CalledProcessError as e:
        print(f"Error during packet capture: {e}")
        exit(1)
```

Step 1: Configuration and Packet Capturing
In this step, the system configuration is set up, and network traffic packets are captured using Tshark:

- Configuration Variables: Key parameters such as INTERFACE (network interface), CAPTURE_DURATION (duration of capture), and PCAP_FILE (output file name for captured packets) are defined.
- Packet Capturing Function: The capture_packets function initiates Tshark to capture network traffic. Optional parameters for duration and count can refine the capture process. Captured packets are saved in the specified PCAP file.

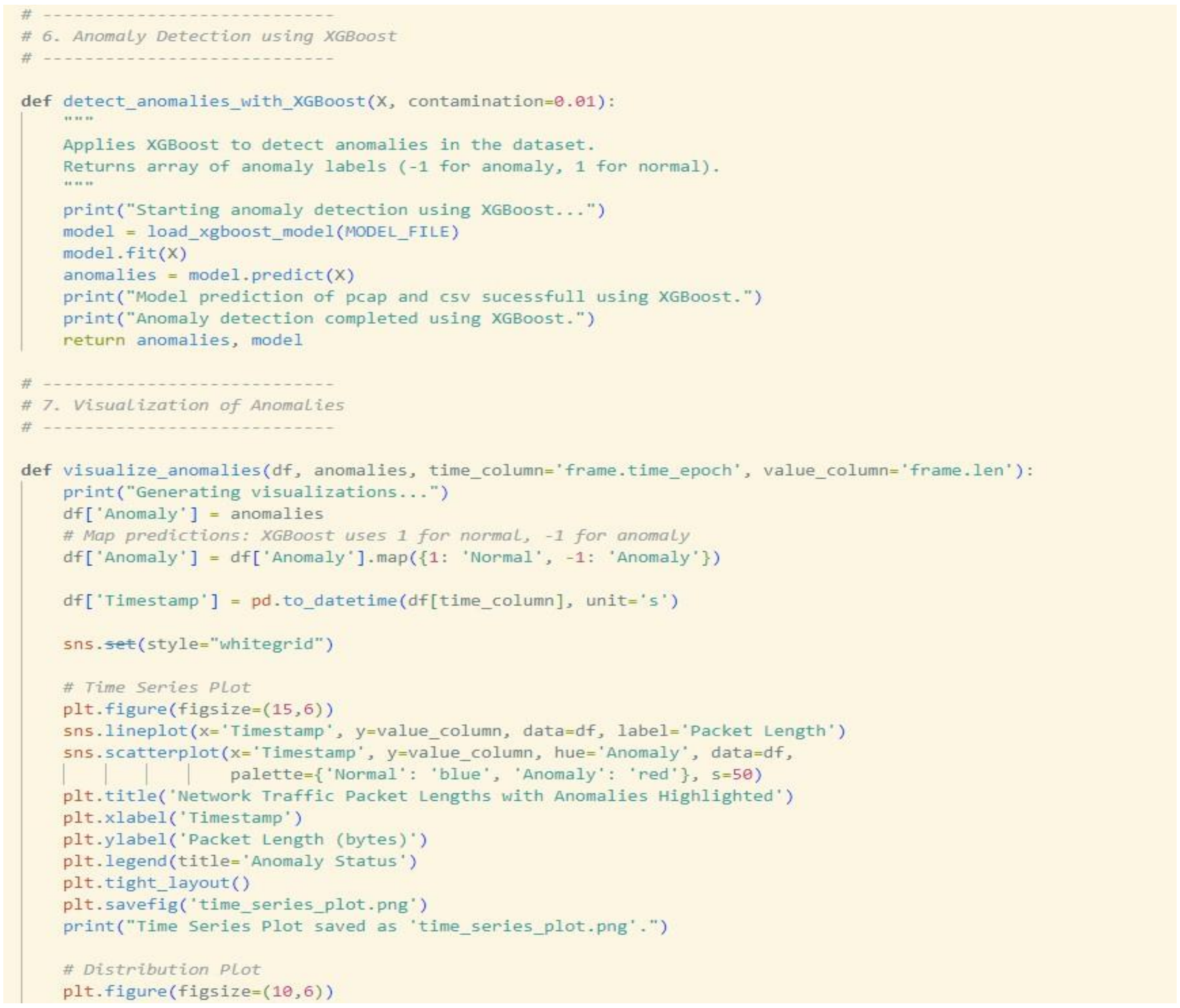


```
# 3. Extracting Packet Features with tshark
# ---------------------------

def extract_features(pcap_file, csv_file, features):
    print(f"Extracting features from {pcap_file}...")
    fields = ",".join(features)
    cmd = [
        "tshark",
        "-r", pcap_file,
        "-T", "fields",
        "-E", "separator=,",
        "-E", "quote=d",
        "-E", "occurrence=f",
        "-e", features[0]  # frame.time_epoch
    ]
    for field in features[1:]:
        cmd.extend(["-e", field])
    try:
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True, check=True)
        with open(csv_file, 'w') as f:
            f.write(','.join(features) + '\n')
            f.write(result.stdout)
        print(f"Feature extraction completed. Saved to {csv_file}.")
    except subprocess.CalledProcessError as e:
        print(f"Error during feature extraction: {e.stderr}")
        exit(1)

# ---------------------------
# 4. Data Cleaning and Preprocessing
# ---------------------------

def preprocess_data(csv_file):
    print(f"Loading and preprocessing data from {csv_file}...")
    df = pd.read_csv(csv_file)
    initial_shape = df.shape
    df.dropna(inplace=True)
    print(f"Removed {initial_shape[0] - df.shape[0]} rows containing null values.")

    initial_shape = df.shape
    df.drop_duplicates(inplace=True)
    print(f"Removed {initial_shape[0] - df.shape[0]} duplicate rows.")


    categorical_features = ['ip.src', 'ip.dst', 'tcp.srcport', 'tcp.dstport']
    existing_categorical = [feature for feature in categorical_features if feature in df.columns]
```

Preprocessing Step 2: Feature Extraction and Data Preprocessing

This step focuses on extracting relevant features from the PCAP file and cleaning/preprocessing the dataset:

- Feature Extraction: The extract_features function uses Tshark to extract specified features, such as frame.time_epoch, ip.src, and frame.len. The extracted data is saved to a CSV file for further processing.
- Data Cleaning: The preprocess_data function cleans the dataset by removing null values and duplicates. Categorical features (ip.src, ip.dst, etc.) are dropped to simplify the analysis.
- Feature Scaling: Numerical features like frame.len and frame.time_epoch are scaled using StandardScaler to ensure consistent data ranges for modeling.



```
# ---------------------------
# 6. Anomaly Detection using XGBoost
# ---------------------------

def detect_anomalies_with_XGBoost(X, contamination=0.01):
    """
    Applies XGBoost to detect anomalies in the dataset.
    Returns array of anomaly labels (-1 for anomaly, 1 for normal).
    """
    print("Starting anomaly detection using XGBoost...")
    model = load_xgboost_model(MODEL_FILE)
    model.fit(X)
    anomalies = model.predict(X)
    print("Model prediction of pcap and csv sucessfull using XGBoost.")
    print("Anomaly detection completed using XGBoost.")
    return anomalies, model

# ---------------------------
# 7. Visualization of Anomalies
# ---------------------------

def visualize_anomalies(df, anomalies, time_column='frame.time_epoch', value_column='frame.len'):
    print("Generating visualizations...")
    df['Anomaly'] = anomalies
    # Map predictions: XGBoost uses 1 for normal, -1 for anomaly
    df['Anomaly'] = df['Anomaly'].map({1: 'Normal', -1: 'Anomaly'})

    df['Timestamp'] = pd.to_datetime(df[time_column], unit='s')

    sns.~~set~~(style="whitegrid")

    # Time Series Plot
    plt.figure(figsize=(15,6))
    sns.lineplot(x='Timestamp', y=value_column, data=df, label='Packet Length')
    sns.scatterplot(x='Timestamp', y=value_column, hue='Anomaly', data=df,
                    palette={'Normal': 'blue', 'Anomaly': 'red'}, s=50)
    plt.title('Network Traffic Packet Lengths with Anomalies Highlighted')
    plt.xlabel('Timestamp')
    plt.ylabel('Packet Length (bytes)')
    plt.legend(title='Anomaly Status')
    plt.tight_layout()
    plt.savefig('time_series_plot.png')
    print("Time Series Plot saved as 'time_series_plot.png'.")

    # Distribution Plot
    plt.figure(figsize=(10,6))
```

Figure 32: Anomaly Detection and Visualization

Step 3: Anomaly Detection and Visualization

In this final step, anomalies are detected using a pre-trained XGBoost model, and the results are visualized:

- Loading the Model: The load_xgboost_model function loads the trained XGBoost model from a
  .joblib file. Model parameters are displayed for verification.
- Anomaly Detection: The detect_anomalies_with_XGBoost function uses the model to classify data points as normal (1) or anomalous (-1).
- Visualization: The visualize_anomalies function creates multiple plots:
  - Time Series Plot: Highlights anomalies over time.
  - Distribution Plot: Shows the density of packet lengths for normal and anomalous traffic.
  - Anomaly Counts: Displays the number of anomalies detected per hour.
    Generated plots are saved as PNG files for documentation and analysis.

References

1. NumPy Documentation: NumPy is a library for numerical computing with support for multi- dimensional arrays and matrices. Available at: https://numpy.org/doc/ .
2. Pandas Documentation: Pandas provides tools for data manipulation and analysis. Available at: https://pandas.pydata.org/pandas-docs/stable/ .
3. Matplotlib Documentation: Matplotlib offers plotting capabilities for data visualization. Available at: https://matplotlib.org/stable/contents.html.
4. Seaborn Documentation: Seaborn simplifies statistical data visualization based on Matplotlib. Available at: https://seaborn.pydata.org/ .
5. Scikit-learn Documentation: Scikit-learn provides machine learning tools for predictive analysis. Available at: https://scikit-learn.org/stable/documentation.html .
6. XGBoost Documentation: XGBoost is a scalable and flexible gradient boosting library. Available at: https://xgboost.readthedocs.io/en/stable .
7. Imbalanced-learn Documentation: A library for dealing with imbalanced datasets in machine learning. Available at: https://imbalanced-learn.org/stable/ .
8. CICIDS2017 Dataset: Canadian Institute for Cybersecurity Intrusion Detection Dataset (2017). Available at: https://www.unb.ca/cic/datasets/ids-2017.html .

Appendix: Detailed Code Documentation

This appendix provides detailed explanations of the code, including the functionality of key components, datasets, and outputs. It complements the configuration manual by offering insights for researchers who wish to understand the implementation in depth.

A.1 Setup and Configuration

The following table clearly highlights the complete step by step guide on how to reproduce the code:

Step No.	Action	Commands/Guides

1	Install Chocolatey package manager (Windows only).	Open PowerShell as Administrator and run:
		Set-ExecutionPolicy Bypass -Scope Process -Force;
		[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
		iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))

2	Install Python using Chocolatey with custom installation (ensure Tcl/Tk and IDLE are checked).	Run choco install python in an Administrator PowerShell. During installation, select "Customize installation" and check Tcl/Tk and IDLE.

3	Install necessary Python libraries like NumPy, Pandas, Scikit-learn, XGBoost, and Imbalanced-Learn.	Run pip install numpy pandas scikit-learn xgboost imbalanced-learn.
4	Set up the nDPI library and tools like Wireshark or Tshark for packet inspection.	Follow nDPI setup guide; use apt-get install tshark for packet capture.
5	Load the CICIDS 2017/2018 dataset into a DataFrame using Pandas.	Use pd.read_csv() to load CSV files into a single DataFrame.
6	Perform data cleaning: remove duplicates, handle missing/infinite values, and standardize column names.	Use df.drop_duplicates() and df.replace([np.inf, -np.inf], np.nan).dropna() to handle duplicates and infinite values. Normalize column names with df.columns.str.replace().
7	Apply data balancing techniques: RandomUnderSampler for majority class and SMOTE for minority class.	Use RandomUnderSampler and SMOTE from imbalanced-learn.
8	Use the nDPI library to extract protocol-specific metadata such as packet size, timing, and flow details.	Configure nDPI to analyze headers and metadata; refer to nDPI documentation.
9	Perform feature selection and scaling using techniques like correlation analysis and MinMaxScaler.	Use correlation matrix (df.corr()) and scale features with MinMaxScaler().fit_transform().
10	Split the dataset into training and testing subsets while maintaining class distribution.	Use train_test_split(X, y, test_size=0.3, stratify=y) from scikit-learn.
11	Train machine learning models (Random Forest, SVM, XGBoost) on the preprocessed dataset.	Use RandomForestClassifier, SVC, and XGBClassifier from respective libraries.
12	Evaluate models using metrics like accuracy, precision, recall, and F1-score.	Use classification_report() and confusion_matrix() from scikit-learn.
13	Integrate the trained model with the DPI framework using Python scripts and joblib/pickle.	Use joblib.dump() or pickle.dump() to save models for deployment.
14	Analyze encrypted metadata using keyword-based DPI to detect threats while preserving privacy.	Use nDPI's protocol analysis tools to inspect metadata.
15	Test system efficiency i.e. measure latency, throughput, and resource utilization under high traffic.	Use htop for resource monitoring and tshark for packet analysis.

A.2 Dataset Loading and Preparation

This project uses the CIC-IDS-2017 dataset for deep packet inspection.

```
import os
import pandas as pd

dataframes = []
for root, dirs, files in os.walk('./data'):
    for file in files:
        if file.endswith('.csv'):
            df = pd.read_csv(os.path.join(root, file))
            dataframes.append(df)

CICIDS = pd.concat(dataframes, ignore_index=True)
display(CICIDS.head())
```

1. The os.walk function iterates through the data/ folder.
2. All .csv files are loaded into individual DataFrames using pd.read_csv().
3. These DataFrames are concatenated into a single dataset.

A.3 Data Cleaning

Cleaning includes handling null values, duplicates, and infinite values.

```
null_values = CICIDS.isnull().sum().sort_values(ascending=False)
print(null_values[null_values > 0])
CICIDS.dropna(inplace=True)
```

Identifies columns with null values.
Removes rows containing null
values.

```
duplicates = CICIDS.duplicated().sum()
if duplicates > 0:
    CICIDS.drop_duplicates(inplace=True)
```

Checks for and removes duplicate rows.

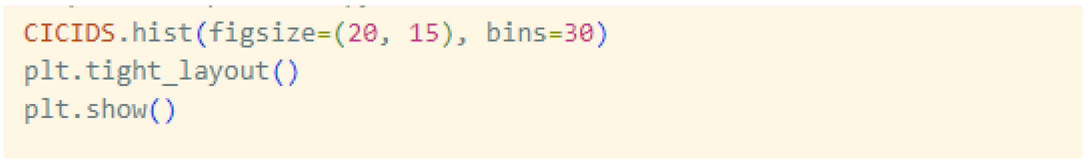
```
CICIDS.replace([np.inf, -np.inf], np.nan, inplace=True)
CICIDS.dropna(inplace=True)
```

Replaces infinite values with NaN and removes them.

A.4 Exploratory Data Analysis (EDA)

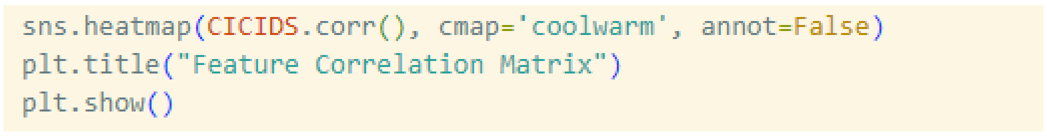
```
sns.barplot(data=CICIDS['label'].value_counts().reset_index(), x='index', y='label')
plt.title("Label Distribution")
plt.show()
```

Visualizes the distribution of benign and malicious labels.



```
CICIDS.hist(figsize=(20, 15), bins=30)
plt.tight_layout()
plt.show()
```

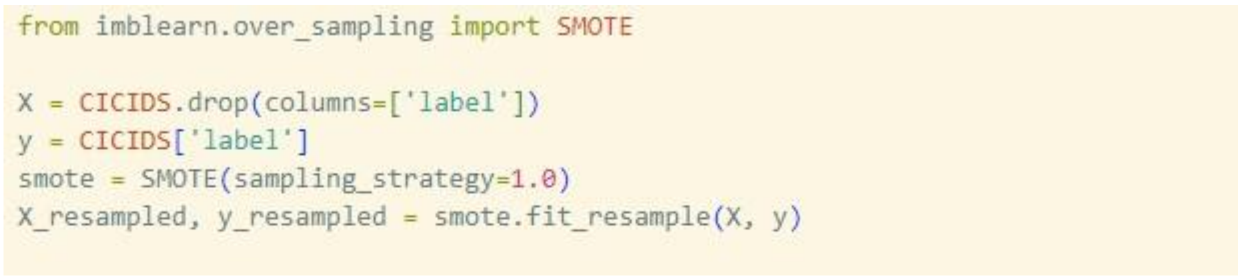
Displays histograms for numeric features.



```
sns.heatmap(CICIDS.corr(), cmap='coolwarm', annot=False)
plt.title("Feature Correlation Matrix")
plt.show()
```

Highlights correlations between features.

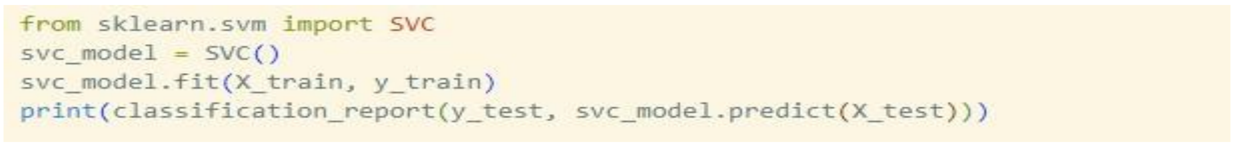
A.5 Data Balancing



```
from imblearn.over_sampling import SMOTE

X = CICIDS.drop(columns=['label'])
y = CICIDS['label']
smote = SMOTE(sampling_strategy=1.0)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

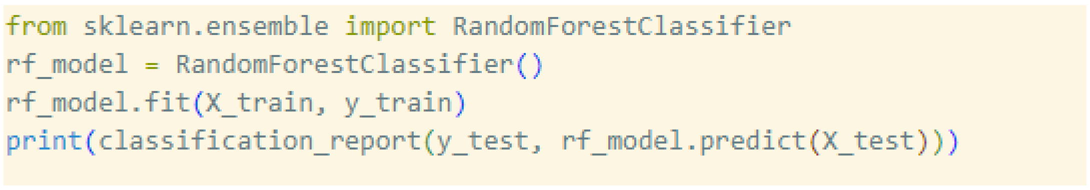
Balances the dataset by oversampling the minority class.

A.6 Model Training and Evaluation



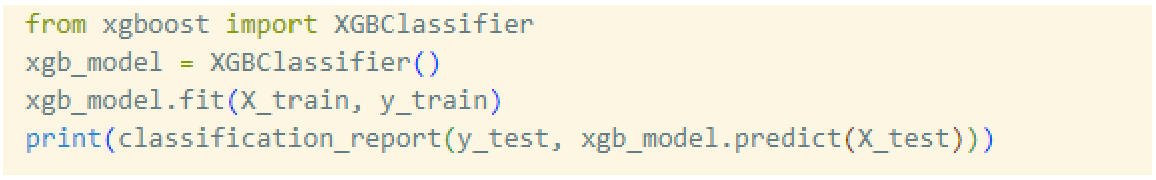
```
from sklearn.svm import SVC
svc_model = SVC()
svc_model.fit(X_train, y_train)
print(classification_report(y_test, svc_model.predict(X_test)))
```

Support Vector Classifier (SVC)



```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)
print(classification_report(y_test, rf_model.predict(X_test)))
```

Random Forest



```
from xgboost import XGBClassifier
xgb_model = XGBClassifier()
xgb_model.fit(X_train, y_train)
print(classification_report(y_test, xgb_model.predict(X_test)))
```

XGBoost

A.7 Visual Outputs

```
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_estimator(svc_model, X_test, y_test)
plt.show()
```

Confusion Matrix

```
importances = rf_model.feature_importances_
plt.barh(range(len(importances)), importances)
plt.title("Feature Importance")
plt.show()
```

Feature Importance (Random Forest)

A.8 Troubleshooting Guide

Ensure all libraries in requirements.txt are installed.

```
pip install -r requirements.txt
```

Check that the dataset files are in the data/ directory.

Verify that the dataset is cleaned and properly balanced before training.

```
numpy==1.23.5
pandas==1.5.3
scikit-learn==1.2.2
xgboost==1.7.5
matplotlib==3.7.1
seaborn==0.12.2
imbalanced-learn==0.10.1
warnings==0.1.5
os-sys==2.1.4
```