International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 12 Issue 12 ǁ December 2024 ǁ PP. 63-79

Detection of DDoS Attacks Using XG Boost-Based Feature Selection and Deep Learning

Arjun R Nair
arjunnair2k18@gmail.com November 20 2024

Date of Submission: 26-11-2024 Date of acceptance: 07-12-2024

I. Introduction

In the rapidly evolving landscape of cybersecurity, network intrusion detection systems (IDS) have become essential tools for safeguarding information assets against malicious activities. As cyber threats grow in complexity and frequency, particularly Distributed Denial of Service (DDoS) attacks, there is an increasing need for advanced detection methodologies that can accurately differentiate be- tween benign and malicious network traffic. Machine learning and deep learning techniques have emerged as powerful approaches to improve the efficacy of IDS by learning intricate patterns from network data.

The Canadian Institute for Cybersecurity's Intrusion Detection System 2017 (**CICIDS2017**) dataset has been established as a benchmark for evaluating intrusion detection algorithms. This dataset contains a rich mix of benign traffic and the most up-to-date common attacks, which closely resemble real-world network environments through Packet Capture (PCAP) files. It includes the results of network traffic analysis using CICFlowMeter, offering labeled flow data based on timestamps, source and destination IP addresses, ports, protocols, and attack types in the CSV format. A significant aspect of the CICIDS2017 dataset was the generation of realistic background traffic. Utilizing the B-Profile system proposed by Sharafaldin et al. (2016), the dataset profiles the abstract behavior of human interactions to generate naturalistic benign traffic. This approach models the behavior of 25 users across various protocols including HTTP, HTTPS, FTP, SSH, and email, thereby creating a comprehensive and realistic dataset for intrusion detection research.

Despite the availability of such detailed datasets, accurately detecting anoma- lies and outliers, especially DDoS attacks—remains a challenge. Factors such as high dimensionality, redundant features, and class imbalance in the data can hinder the performance of traditional detection methods. Therefore, there is a pressing need to develop advanced models that not only improve detection accuracy but also optimize computational efficiency and resource utilization.

1.1 Problem Statement

This research addresses the challenge of detecting anomalies and DDoS attacks within network traffic by improving intrusion detection accuracy using machine learning algorithms. Specifically, it focuses on enhancing the performance of IDS by leveraging deep learning techniques to classify network flow samples into benign or malicious categories, considering both binary and multiclass classifi- cation scenarios.

1.2 Objectives

The primary objective of this study is to develop a robust intrusion detection model that achieves high accuracy while maintaining computational efficiency. The specific goals include:

1. **Feature Selection**: Implement feature selection using the Extreme Gra- dient Boosting (XGBoost) algorithm to identify the most significant fea- tures, reduce data dimensionality, and optimize memory usage without compromising detection performance.
2. **Handling Class Imbalance**: Address the issue of class imbalance in- herent in the CICIDS2017 dataset to ensure that minority classes, such as specific types of attacks, are accurately detected. Techniques such as resampling or cost-sensitive learning may be employed to mitigate this challenge.
3. **Deep Learning Model Development**: Leverage Multilayer Percep- tron (MLP) architectures through the Keras framework to construct deep learning models capable of learning complex patterns in network traffic data. The models will be configured for both binary classification (distin- guishing between benign and malicious traffic) and multiclass classification (identifying specific attack types).
4. **Performance Optimization**: Achieve state-of-the-art results with an accuracy of approximately

99% or higher. This involves fine-tuning the deep learning models and optimizing hyperparameters to enhance detec- tion capabilities while minimizing computational overhead.

5. **Validation and Evaluation**: Rigorously evaluate the proposed models using appropriate metrics such as accuracy, precision, recall, F1-score, and area under the Receiver Operating Characteristic (ROC) curve. Cross- validation and testing on unseen data will be conducted to ensure the generalizability of the results.

II. Literature Review

Intrusion Detection Systems (IDS) are critical components in modern network security infrastructure, designed to monitor network traffic and identify potential threats or unauthorized access [2]. Over the years, IDS have evolved from signature-based detection methods to incorporate advanced machine learning (ML) and deep learning techniques, enabling them to detect novel and sophisticated cyber-attacks [3].

2.1 Intrusion Detection Systems and Machine Learning

Traditional IDS rely on predefined signatures or anomaly detection rules, which can be insufficient against zero-day attacks or sophisticated evasion techniques [4]. Machine learning offers a dynamic and adaptive approach, where models learn patterns from data to detect anomalies [4]. Studies have demonstrated that ML-based IDS can significantly improve detection rates and reduce false positives [1].

2.2 Feature Engineering in IDS

Feature engineering is a pivotal process in developing ML models for IDS. It involves selecting and transforming variables to improve model performance [5]. In network intrusion detection, features can include various network flow char- acteristics such as packet sizes, durations, protocols, and statistical measures [1]. Effective feature selection reduces dimensionality, mitigates the curse of dimensionality, and enhances model interpretability [6]. Techniques like prin- cipal component analysis (PCA) and autoencoders have been used to extract significant features [7].

Sharafaldin et al. [1] emphasized the importance of comprehensive feature selection and proposed a new approach for generating reliable datasets, leading to the creation of the CICIDS2017 dataset.

2.3 CICIDS2017 Dataset in IDS Research

The CICIDS2017 dataset has become a standard benchmark for evaluating IDS models due to its realistic representation of network traffic and inclusion of contemporary attack types [1]. Several studies have utilized this dataset to assess the effectiveness of various detection techniques.

For instance, Maciá-Fernández et al. [8] investigated the impact of metadata features on machine-learned IDS models using the CICIDS2017 dataset. They found that including certain metadata can contaminate the model, leading to overfitting and reduced generalizability. Their study highlights the necessity of careful feature selection to avoid the inclusion of non-representative data that could skew model performance.

Other researchers have employed the dataset to explore deep learning ap- proaches. Yin et al. [9] proposed a deep learning framework using recurrent neural networks (RNN) to achieve high detection rates. Similarly, Ullah and Mahmoud [10] developed a hybrid model combining convolutional neural net- works (CNN) and long short-term memory (LSTM) networks, demonstrating improved performance on the CICIDS2017 dataset.

2.4 Machine Learning Approaches for IDS

Among machine learning algorithms, ensemble methods like Extreme Gradient Boosting (XGBoost) have gained significant attention for IDS applications [11]. XGBoost is known for its scalability, efficiency, and high predictive accuracy, especially on large and complex datasets.

XGBoost operates by building an ensemble of weak learners, typically de- cision trees, in a sequential manner where each new tree focuses on correcting the errors of the previous ones [11]. This approach makes it powerful for classi- fication tasks within IDS, particularly in anomaly detection where patterns are subtle and complex.

A key advantage of XGBoost is its ability to handle imbalanced datasets, a common issue in IDS where attack instances are rare compared to normal traffic [12]. Techniques such as weighting the loss function and using appropriate evaluation metrics help in addressing class imbalance. Furthermore, XGBoost’s parallelization capability ensures it trains faster compared to other gradient boosting methods.

Comparisons with other algorithms have been explored in the literature. LightGBM, developed by Microsoft, offers faster training and lower memory us- age by using histogram-based algorithms and leafwise tree growth [13]. It often achieves higher accuracy than level-wise growth used in many other algorithms, particularly when dealing with large datasets or high-dimensional data [14].

CatBoost, another gradient boosting algorithm, is designed to handle cate- gorical features efficiently and reduce overfitting [15]. It incorporates techniques to address the prediction shift caused by target leakage, which is beneficial in IDS applications where data integrity is crucial.

Random Forest, an ensemble method using bagging and feature randomness, constructs multiple decision trees and aggregates their results, offering robust- ness and ease of interpretation [16]. Although it may not achieve the same level of accuracy as boosted models, Random Forest is valuable for its robustness to overfitting and its performance on a wide range of datasets [17].

In the context of IDS, the choice between these algorithms depends on spe- cific dataset characteristics and computational constraints. For instance, Light- GBM might be preferred for very large datasets due to its speed and efficiency [14], while Random Forest may be suitable for problems where model inter- pretability is a priority.

2.5 Feature Interpretation and Explainability

Understanding model decisions is crucial in IDS to ensure trust and compliance. Techniques such as Shapley values provide insights into feature contributions to predictions [18]. Shapley values offer a unified approach to interpreting model outputs, helping analysts understand the importance of each feature in the detection process [19].

Moreover, tools like Pandas Profiling (now known as ydata-profiling) facili- tate exploratory data analysis by generating comprehensive reports on dataset features [20]. Such tools help in identifying data quality issues, understanding feature distributions, and uncovering patterns that may influence model perfor- mance.

Quantile functions are also used in statistical analysis to understand the distribution of features, which is valuable in preprocessing and normalizing data for ML models [21]. By analyzing quantiles, researchers can detect outliers and better prepare data for training robust IDS models.

2.6 Summary

The integration of advanced machine learning techniques in IDS has shown sig- nificant promise in enhancing detection capabilities. The CICIDS2017 dataset serves as a robust benchmark for evaluating these methods. XGBoost, among other algorithms, has demonstrated high accuracy and efficiency in handling the complexities of intrusion detection. However, the choice of algorithm should be tailored to the specific characteristics of the dataset and the requirements of the deployment environment. Feature engineering and model interpretability remain critical components in developing effective IDS solutions.

III. Methodology

3.1 Dataset Description

The Canadian Institute for Cybersecurity’s Intrusion Detection System 2017 (**CICIDS2017**) dataset is utilized in this research as the primary source of network traffic data for intrusion detection analysis. This dataset is renowned for its comprehensive and realistic representation of modern network traffic pat- terns, including both benign activities and a variety of malicious attacks [1].

The CICIDS2017 dataset was created with the goal of resembling true real- world data, incorporating Packet Capture (PCAP) files that capture network traffic over a period of five consecutive days. The data collection commenced at **9 a.m. on Monday, July 3, 2017, and concluded at 5 p.m. on Friday, July 7, 2017**, covering both normal and attack traffic periods.

Key features of the dataset include:

- **Benign Traffic**: The dataset includes naturalistic benign background traffic. To generate realistic background traffic, the B-Profile system pro- posed by Sharafaldin et al. [1] was employed. This system profiles the abstract behavior of human interactions, simulating the activity of 25 users based on protocols such as HTTP, HTTPS, FTP, SSH, and email.
- **Attack Representations**: The dataset encompasses the most up-to-date common attacks, executed during specific periods to emulate real-world scenarios. The attacks included are:

– **Brute Force FTP**: Unauthorized access attempts targeting FTP services.
– **Brute Force SSH**: Repeated login attempts to breach SSH security.
– **DoS (Denial of Service)**: Attacks aiming to render network re- sources unavailable to legitimate users.
– **Heartbleed**: Exploitation of the Heartbleed vulnerability in OpenSSL.
– **Web Attacks**: Including SQL injection, cross-site scripting, and other web-based attacks.

- **Infiltration**: Unauthorized access and compromise of network sys- tems.
- **Botnet**: Activities related to botnet communication and coordina- tion.
- **DDoS (Distributed Denial of Service)**: Coordinated attacks from multiple sources to overwhelm network resources.

- **Data Format**: The dataset includes labeled flow-based features extracted using CICFlowMeter, providing detailed information such as timestamps, source and destination IP addresses, ports, protocols, and attack labels. The data is stored in CSV files, facilitating ease of use for machine learning applications.

The inclusion of both benign and malicious activities, along with comprehen- sive feature representation, makes the CICIDS2017 dataset suitable for develop- ing and evaluating intrusion detection models under realistic network conditions.

3.2 Data Preprocessing

Effective data preprocessing is essential to ensure the quality and reliability of the machine learning models developed for intrusion detection. The following steps were undertaken to prepare the dataset for analysis:

1. **Data Consolidation:**
   - The dataset, originally divided into multiple CSV files represent- ing different days and attack types, was consolidated into a single DataFrame for uniform processing.

2. **Handling Null Values:**
   - The dataset was checked for the presence of null values using the `isnull()` function.
   - **Null Value Detection:**

```
null_counts = df_data.isnull().sum()
total_nulls = null_counts.sum()
print(f"{total_nulls} null entries have been found
    in the dataset\n")
```

- **Null Value Removal:**
  - All records with null values were removed to prevent inconsistencies in model training.

```
df_data.dropna(inplace=True)
```

3. **Handling Duplicate Entries:**

- Duplicate records can bias the model by over-representing certain instances.
- **Duplicate Detection:**

```
duplicate_count = df_data.duplicated().sum()
print(f"{duplicate_count} duplicate entries have
    been found in the dataset\n")
```

- **Duplicate Removal:**
  - All duplicate rows were removed to ensure each record is unique.

```
df_data.drop_duplicates(inplace=True)
```

- **Index Resetting:**
  - After removal operations, the DataFrame indices were reset for consistency.

```
df_data.reset_index(drop=True, inplace=True)
```

4. **Data Type Inspection:**

- Identification of categorical and numerical features was performed.

```
categorical_columns = df_data.select_dtypes(include
    =['object']).columns.tolist()
print("Categorical columns:", categorical_columns, '
    \n')
```

- The primary categorical column identified was the 'Label' column, indicating the class of each record.

5. **Feature and Target Separation:**

- **Features** (X): All columns except 'Label' were considered as features.

```
X = df_data.drop('Label', axis=1)
```

- **Target** (y): The 'Label' column was extracted as the target variable.

```
y = df_data['Label'].copy()
```

3.3 Transformation into Binary and MultiClass Classification

The nature of intrusion detection problems can vary based on the specific objectives. In this research, the problem was approached from both binary and multiclass classification perspectives.

3.3.1 Justification for Transforming the Problem

Binary Classification

- **Objective**: To distinguish between normal (benign) and abnormal (malicious) network traffic.
- **Rationale**:
  - Simplifies the classification task, enabling the model to focus on detecting any form of intrusion.
  - Suitable for scenarios where the primary concern is to flag potential threats for further investigation.
  - Helps to address the general imbalance between normal and malicious traffic.

Multiclass Classification

- **Objective**: To identify and categorize specific types of network attacks.
- **Rationale**:
- Provides granular insights into the nature of the detected intrusions.
- Enables tailored response strategies for different attack types.
- Addresses the complex class imbalance across multiple attack categories.
- Useful for comprehensive intrusion detection systems that prioritize detailed threat analysis.

3.3.2 Implementation of Classification Configurations

Binary Classification Configuration

- **Label Binarization**:
  - The 'Label' column was transformed to represent two classes:
    - 0 for 'Benign' traffic.
    - 1 for all types of attacks.
  - **Code Implementation**:

```python
# Binarize labels: Map 'Benign' to 0 and all other
   attack labels to 1
y_b = y.map({'Benign': 0}).fillna(1)
```

- Class Distribution Analysis:
  - Understanding the ratio of benign to malicious traffic is crucial due to class imbalance concerns.
  - Minimum Baseline Accuracy:
    * A naive model predicting the majority class would achieve an accuracy equal to the proportion of the dominant class (e.g., if benign traffic is 84.92%, the baseline accuracy is 84.92%).
    * The model must outperform this baseline to be considered effective.

Multiclass Classification Configuration

- Label Encoding:
  - All unique attack types, along with 'Benign', were encoded into integer labels using Label Encoding.
  - Code Implementation:

```python
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
y_m = pd.Series(y_encoded)
```

- Class Distribution Analysis:
  - The distribution of each class was examined to identify minority classes.
  - Class Imbalance Consideration:
    * Some attack types constitute a very small fraction of the dataset.
    * Special techniques (e.g., resampling, class weighting) may be necessary to ensure these classes are adequately represented during model training.

3.3.3 Addressing Class Imbalance

- Techniques Employed:
  - Resampling Methods:
    * Oversampling minority classes or undersampling majority classes to balance the dataset.

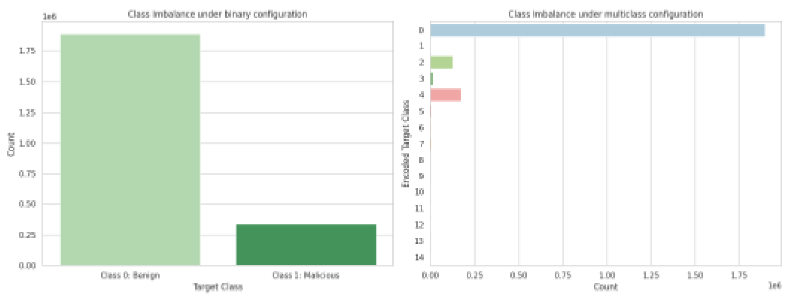


Figure 1: Class Imbalance between binary and multiclass configuration

- **Algorithmic Approaches:**
  - Using models that incorporate class weights (e.g., XGBoost) to penalize misclassification of minority classes more heavily.
- **Evaluation Metrics:**
  - Relying on metrics beyond accuracy, such as precision, recall, F1-score, and area under the ROC curve, to assess model performance on imbalanced data.

3.3.4 Benefits of Dual Approach

- **Comprehensive Evaluation:**
  - Assessing model performance across different levels of classification complexity.
  - Extracting insights on feature importance and model generalization capabilities.
- **Practical Applicability:**
  - Catering to different operational needs, from simple intrusion detection to detailed attack classification.
  - Enhancing the adaptability of the IDS in various network security scenarios.

3.4 Conclusion on Transformation

Transforming the intrusion detection problem into both binary and multiclass classification tasks allows for a robust evaluation of machine learning models. It addresses different practical needs in network security, from the quick detection of any intrusion to the precise identification of attack types. By meticulously preprocessing the data and thoughtfully configuring the classification tasks, the research sets a solid foundation for developing effective and efficient intrusion detection models.

3.5 Feature Engineering

Feature selection is a crucial step in developing efficient and effective machine learning models, especially in high-dimensional data scenarios common in intrusion detection systems. In this research, we performed feature selection using the Extreme Gradient Boosting (**XGBoost**) algorithm. XGBoost is a tree-based ensemble learning algorithm that inherently performs feature selection during its training process. This embedded method helps in selecting the most informative features while reducing the dimensionality of the input space.

3.5.1 Introduction to Feature Selection with XGBoost

XGBoost is widely used for classification and regression tasks due to its scalability and speed [11]. One of its key advantages is the ability to automatically compute feature importance scores during model training. These scores indicate the contribution of each feature to the predictive performance of the model.

Explanation of Feature Importance Calculation In decision tree-based algorithms like XGBoost, feature importance scores are derived from the impact of features on reducing impurity in decision trees. One common measure of impurity used in classification tasks is the **Gini impurity**.

Gini Impurity

Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. For a binary classification problem with classes 0 and 1, the Gini impurity G for a node with N samples is calculated as:

$$G = 1 - \sum_{i=0}^{1} p_i^2, \tag{1}$$

where p_i is the probability of class i in the node.

Feature Importance Calculation

During the construction of decision trees, at each node, possible splits on each feature are evaluated, and the split that maximally reduces the Gini impurity is selected. The feature importance score is then calculated based on the total reduction in impurity achieved by splitting on that feature across all trees in the ensemble.

Features that contribute more to reducing the impurity (i.e., lead to greater reduction in Gini impurity when used for splitting nodes) are considered more important by the algorithm. These important features can be used for further analysis or as input to other machine learning models, such as neural networks.

Relationship Between Gini Index and Feature Importance In XG-Boost, the Gini index serves as the default metric for assessing impurity when constructing decision trees. For a node with class probabilities p_1 and p_2 (where $p_1 + p_2 = 1$), the Gini index is calculated as:

$$GI = 1 - (p_1^2 + p_2^2). \tag{2}$$

Lower values of the Gini index indicate a higher degree of class separation, signifying improved model performance. During tree construction, the algorithm selects splits that maximally reduce the Gini index. Feature importance is determined by the total reduction in the Gini index achieved by splits involving that feature across all nodes of all trees in the ensemble.

3.5.2 Implementation of Feature Selection

The feature selection process was implemented using the following code.

```python
def get_feature_importances(original_data, clf_params,
    printing=False):
    """
    Calculate the feature importances using an XGBoost
        classifier trained on the provided data.

    Args:
    - original_data (list): A list containing X_train, X_val
        , X_test, y_train, y_val, and y_test data.
    - clf_params (dict): Parameters for the XGBoost
        classifier.
    - printing (bool, optional): If True, prints the feature
         names alongside their importance scores.

    Returns:
    - feature_importances (array): An array containing the
        importance scores of each feature.
    """
    # Extract original data
    X_train, X_val, X_test, y_train, y_val, y_test =
        original_data

    # Create XGBoost classifier instance
    xgb_clf = XGBClassifier(**clf_params)

    # Train the classifier
    xgb_clf.fit(X_train, y_train, eval_set=[(X_val, y_val)],
         verbose=False)

    # Get feature importances
    feature_importances = xgb_clf.feature_importances_

    if printing:
        # Print feature importances
        max_feature_name_length = max(len(name) for name in
            X_train.columns)
        print(f"{'Feature Name':<{max_feature_name_length}}\
            tImportance")
        for name, importance in zip(X_train.columns,
            feature_importances):
            print(f"{name:<{max_feature_name_length}}\t{
                importance*100:.2f}%")

    return feature_importances
```

The function above calculates the feature importances using an XGBoost classifier trained on the provided training data. It can optionally print out the feature names alongside their importance scores for inspection.

```python
def select_features_by_threshold(original_data, clf_params,
   threshold=0.01, printing=False):
    """
    Select features based on a given importance threshold.

    Args:
    - original_data (list): Contains X_train, X_val, X_test,
        y_train, y_val, y_test data.
    - clf_params (dict): Parameters for the XGBoost
       classifier.
    - threshold (float): Importance threshold above which
       features will be selected.
    - printing (bool, optional): If True, prints the names
       and importances of the selected features.

    Returns:
    - X_train_selected (DataFrame): Subset of X_train
       containing selected features.
    - X_val_selected (DataFrame): Subset of X_val containing
        selected features.
    - X_test_selected (DataFrame): Subset of X_test
       containing selected features.
    - feature_tuples (list): List of tuples containing
       feature names and their importances.
    """
    # Extract data
    X_train, X_val, X_test, y_train, y_val, y_test =
       original_data

    # Get feature importances
    feature_importances = get_feature_importances(
       original_data, clf_params)

    # Get feature names
    feature_names = X_train.columns

    # Select features based on threshold
    selected_features = np.where(feature_importances >
       threshold)[0]

    # Create list of feature names and importances
    feature_tuples = [(name, importance) for name,
       importance in zip(
        feature_names[selected_features],
           feature_importances[selected_features])]

    # Sort features by importance
    sorted_feature_tuples = sorted(feature_tuples, key=
       lambda x: x[1], reverse=True)
    sorted_feature_names = [tup[0] for tup in
       sorted_feature_tuples]

    # Subset data with selected features
    X_train_selected = X_train.loc[:, sorted_feature_names]
    X_val_selected = X_val.loc[:, sorted_feature_names]
    X_test_selected = X_test.loc[:, sorted_feature_names]

    if printing:
        # Print selected features
        max_feature_name_length = max(len(name) for name in
           sorted_feature_names)
        print(f"The number of selected features are: {len(
           sorted_feature_names)}\n")
        print(f"{'Feature Name':<{max_feature_name_length}}\
           tImportance\n")
        for name, importance in sorted_feature_tuples:
            print(f"{name:<{max_feature_name_length}}\t{
               importance*100:.2f}%")

    return X_train_selected, X_val_selected, X_test_selected
       , sorted_feature_tuples
```

This function selects features whose importance scores exceed a specified threshold.

3.5.3 Feature Selection Under Binary and Multiclass Configurations

We performed feature selection separately under binary and multiclass configurations.

Binary Configuration We defined the classifier parameters and executed the feature selection process:

```
# Define classifier parameters
clf_params_b = dict(
    objective='binary:logistic',
    n_estimators=50,
    eval_metric='logloss',
    max_depth=3,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.1,
    early_stopping_rounds=10,
    random_state=seed_pi
)

# Perform feature selection
print("Under Binary Configuration:\n")
X_train_selected_b, X_val_selected_b, X_test_selected_b,
   feature_tuples_b = select_features_by_threshold(
    original_data_b, clf_params=clf_params_b, threshold
        =0.01, printing=True)
```

Multiclass Configuration Similarly, for the multiclass classification task:

```
# Define number of classes
num_classes = len(class_counts_m)

# Define classifier parameters
clf_params_m = dict(
    objective='multi:softmax',
    num_class=num_classes,
    n_estimators=50,
    eval_metric='mlogloss',
    max_depth=3,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.1,
    early_stopping_rounds=10,
    random_state=seed_pi
)

# Perform feature selection
print("Under Multiclass Configuration:\n")
X_train_selected_m, X_val_selected_m, X_test_selected_m,
   feature_tuples_m = select_features_by_threshold(
    original_data_m, clf_params=clf_params_m, threshold
        =0.01, printing=True)
```

3.5.4 Visualization of Feature Importances

We plotted the feature importances for both configurations to visualize the significance of each feature.

```
# Define figure and subplots
fig, axs = plt.subplots(1, 2, figsize=(16, 8))

# Binary configuration
feature_names_b = [tup[0] for tup in feature_tuples_b]
significance_scores_b = [tup[1] for tup in feature_tuples_b]
axs[0].barh(feature_names_b, significance_scores_b, color='
    skyblue')
axs[0].set_xlabel('Feature Importance')
axs[0].set_ylabel('Features')
axs[0].set_title("Feature Importance under Binary
    Classification")
axs[0].invert_yaxis()

# Multiclass configuration
feature_names_m = [tup[0] for tup in feature_tuples_m]
significance_scores_m = [tup[1] for tup in feature_tuples_m]
axs[1].barh(feature_names_m, significance_scores_m, color='
    red')
axs[1].set_xlabel('Feature Importance')
axs[1].set_ylabel('Features')
axs[1].set_title("Feature Importance under Multiclass
    Classification")
axs[1].invert_yaxis()

# Adjust layout and display
plt.tight_layout()
plt.show()
```

3.5.5 Analysis of Unique Features

We identified features that were unique to each configuration:

```
# Features unique to binary configuration
unique_in_b = set(feature_names_b) - set(feature_names_m)

# Features unique to multiclass configuration
unique_in_m = set(feature_names_m) - set(feature_names_b)
```

3.5.6 Creation of New Features

While the primary focus was on selecting existing features based on their importance scores, we also explored the creation of new features that could potentially

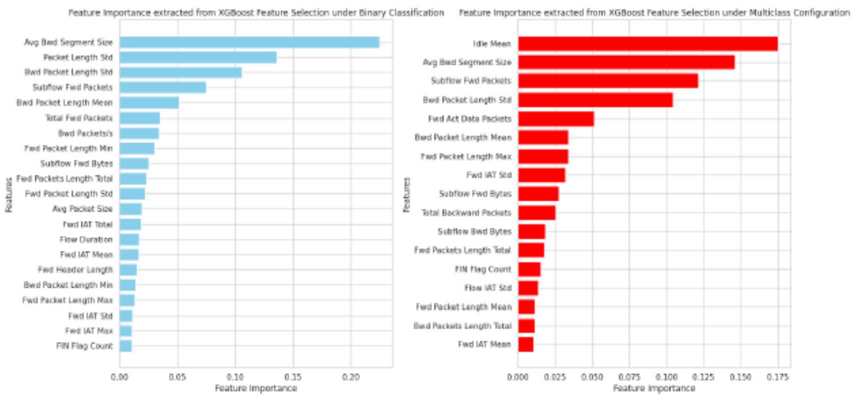


Figure 2: Feature Importance Plots for Binary and Multiclass Configurations

enhance model performance. This involved combining existing features or computing statistical measures that may capture underlying patterns in the data.

The intended impact of creating new features was to provide the model with additional information that could improve its ability to distinguish between benign and malicious traffic, as well as between different types of attacks in the multiclass configuration.

3.6 Conclusion on Feature Engineering

Feature selection using XGBoost allowed us to identify and retain the most informative features, effectively reducing the dimensionality of the dataset without compromising model performance. The reduction in dimensionality leads to lower computational costs and potentially improves the generalization of the model. The differences in selected features between the binary and multiclass configurations highlight the importance of context when performing feature selection.

3.7 Outlier Analysis

Note that outlier analysis is performed only for the **Binary Configuration**. We avoided this procedure for the multiclass setup to save time, as we expect analogous results. Since there is no outlier-handling procedure in the multiclass setting, this analysis serves to enhance our understanding of the problem.

3.7.1 Definition of Outlier

An outlier in statistics is a data point that deviates significantly from the overall pattern of the remaining data. Mathematically, there isn't a universally accepted definition, but two common approaches are often used:

Z-scores We can define an outlier based on its Z-score, which measures how many standard deviations a data point is away from the mean:

$$Z_{\text{score}} = \frac{x - \mu}{\sigma} \tag{3}$$

Where:

- x is the data point in question,
- μ is the population mean (often estimated by the sample mean), and
- σ is the population standard deviation (often estimated by the sample standard deviation).

We can then define a threshold for Z-scores (e.g., ±3 standard deviations). Data points exceeding this threshold in absolute value can be considered outliers.

Interquartile Range (IQR) This approach utilizes the quartiles of the data distribution:

- $Q1$ (First quartile): represents the 25th percentile (value below which 25% of the data lies).
- $Q3$ (Third quartile): represents the 75th percentile (value below which 75% of the data lies).

We calculate the Interquartile Range (IQR):

$$\text{IQR} = Q3 - Q1 \tag{4}$$

The lower and upper bounds are defined as:

$$\begin{aligned} \text{Lower bound} &= Q1 - 1.5 \times \text{IQR} \\ \text{Upper bound} &= Q3 + 1.5 \times \text{IQR} \end{aligned} \tag{5}$$

Outliers can be defined as data points falling outside these bounds. Data points below the lower bound or exceeding the upper bound can be considered potential outliers.

Since the IQR method is robust against skewed distributions, while Z-scores are better for normally distributed data, the IQR approach will be adopted in this analysis.

3.7.2 Implementation of Outlier Detection

We implemented the outlier detection using the IQR method on our dataset. The following code demonstrates how outliers are detected:

```
if 'X_final_b' not in globals():
    # Concatenate 'X_train_final_b', 'X_val_final_b', and '
        X_test_final_b'
    # into a DataFrame named X_final_b
    X_final_b = pd.concat([X_train_final_b, X_val_final_b,
        X_test_final_b])

# Define function for IQR outlier detection
def detect_outliers_iqr(df):
    """
    Detects outliers using the Interquartile Range (IQR)
        method.
    Args:
        df (pandas.DataFrame): The dataframe containing the
            features.
    Returns:
        pandas.DataFrame: A new dataframe with additional
            columns indicating outliers.
    """
    # Extract columns
    columns = df.columns.tolist()
    # Create a new DataFrame to store the outliers
    outliers = pd.DataFrame(columns=[col + '_outliers' for
        col in columns])
    for col in columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        # If they fall outside of boundaries, they are
            defined as outliers
        outliers[col + '_outliers'] = df[col].apply(lambda x
            :
                                        1 if (x < lower_bound or
                                            x > upper_bound)
                                            else 0)
    return outliers

# Detect outliers
df_with_outliers = detect_outliers_iqr(X_final_b.copy())
# .sum() will create the requested result
print("Number of outlier samples for every column:")
print("[for a concatenation of X_train_final_b,
    X_val_final_b, X_test_final_b dataframes] \n")
```

```
print(df_with_outliers.sum())
```

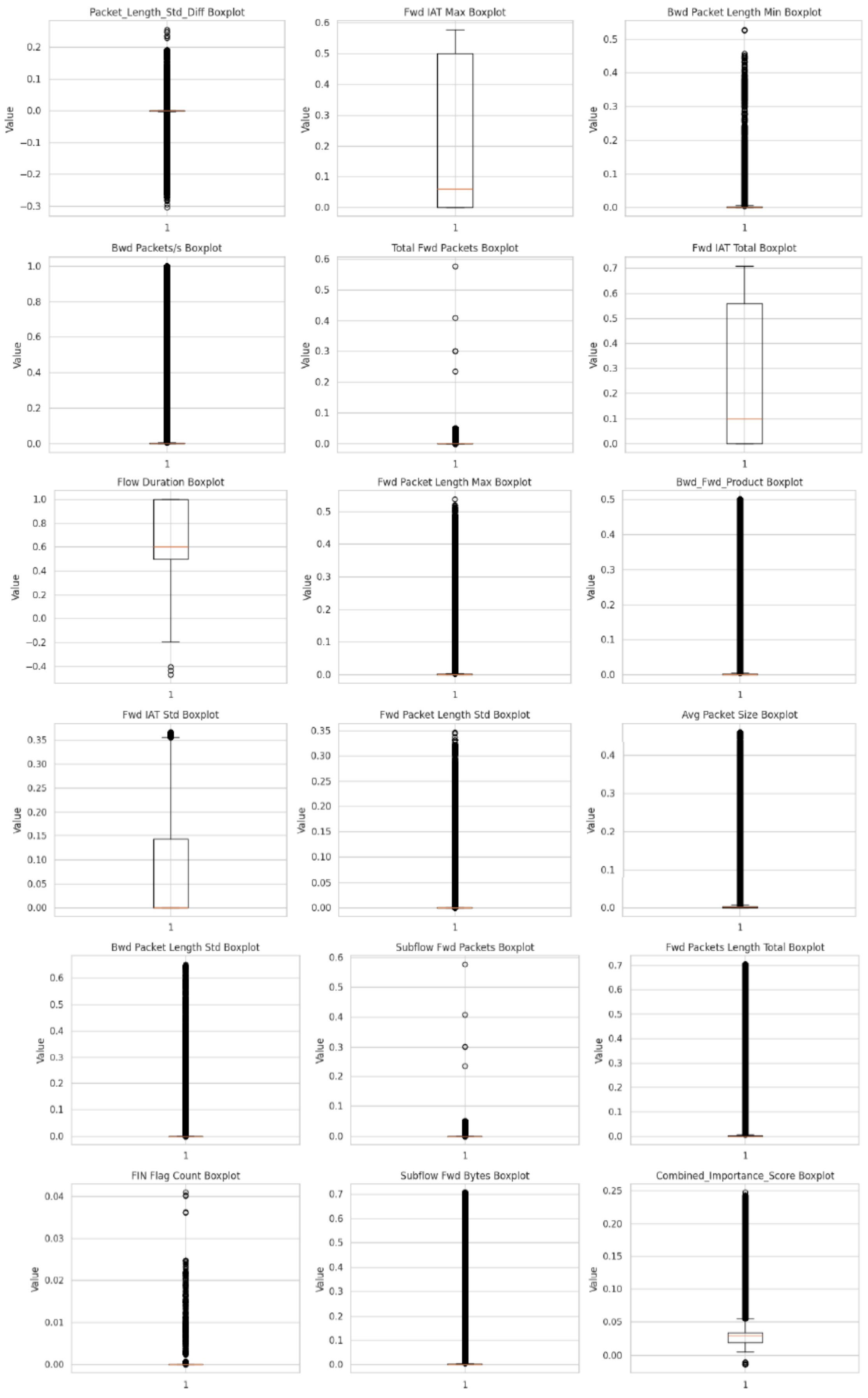
3.7.3 Outlier Boxplot Visualization

To visualize the distribution of features and the presence of outliers, we used boxplots. The following function generates boxplots for the specified features:

```
def visualize_boxplots(df, features, figsize=(15, 28)):
    """
    Visualizes boxplots for the specified features using a
        grid layout.
    Args:
        df (pandas.DataFrame): The dataframe containing the
            features.
        features (list): A list of feature names to
            visualize.
        figsize (tuple, optional): The size of the figure.
            Defaults to (15, 28).
    """
    n_features = len(features)  # Example: 20 features
    rows = int(np.ceil(n_features / 3))  # Calculate number
        of rows for a 3-column grid
    fig, axes = plt.subplots(rows, 3, figsize=figsize)

    # Flatten the axes array for easy iteration
    axes_flat = axes.ravel()

    for i, feature in enumerate(features):
        axes_flat[i].boxplot(df[feature])
        axes_flat[i].set_title(f"{feature} Boxplot")
        axes_flat[i].set_ylabel("Value")

    plt.tight_layout()
    plt.show()


# Visualize boxplots
visualize_boxplots(X_final_b, X_final_b.columns.tolist())
```

Interpretation of Boxplots The boxplot elements provide insights into the distribution of data along the y-axis:

- Whiskers: Extend from the first quartile (Q1) to the third quartile (Q3), indicating the range where most data points lie (within 1.5 times the IQR).
- Box: Represents the Interquartile Range (IQR), emphasizing the central 50% of the data distribution.
- Median: Depicted as a horizontal line within the box, indicates the value that separates the lower and upper halves of the data.
- Outliers: Individual data points plotted beyond the whiskers, may signify values significantly deviating from the primary distribution.


Packet_Length_Std_Diff Boxplot
Fwd IAT Max Boxplot
Bwd Packet Length Min Boxplot
Bwd Packets/s Boxplot
Total Fwd Packets Boxplot
Fwd IAT Total Boxplot
Flow Duration Boxplot
Fwd Packet Length Max Boxplot
Bwd_Fwd_Product Boxplot
Fwd IAT Std Boxplot
Fwd Packet Length Std Boxplot
Avg Packet Size Boxplot
Bwd Packet Length Std Boxplot
Subflow Fwd Packets Boxplot
Fwd Packets Length Total Boxplot
FIN Flag Count Boxplot
Subflow Fwd Bytes Boxplot
Combined_Importance_Score Boxplot
Value
1

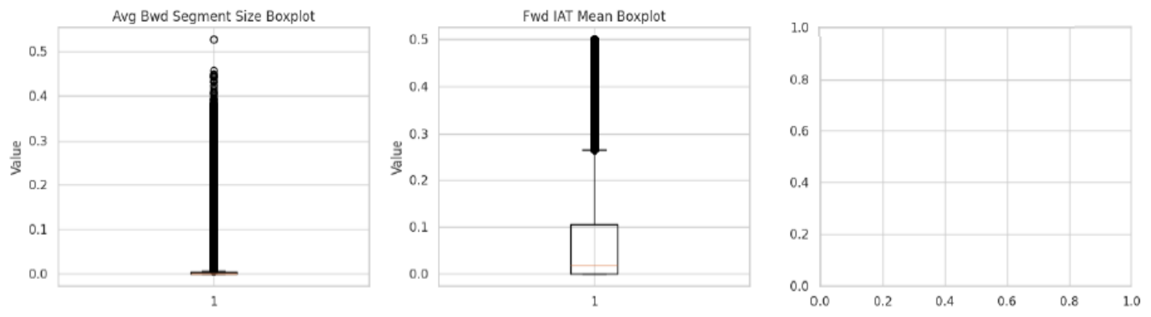


Figure 3: Boxplots of Features for Outlier Analysis

3.7.4 Conclusion

Our statistical analysis identified a significant number of outliers, about 25% of the data. To address these outliers, efficient data scaling techniques will be preferred over traditional methods like winsorization, in order to minimize potential information loss.

While other outlier detection methods like Isolation Forest and Local Outlier Factor exist, their high computational costs make them impractical for the large size of the current dataset.

References

[1] Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2018). Toward generat- ing a new intrusion detection dataset and intrusion traffic characterization. In Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP) (pp. 108–116).

[2] Scarfone, K., & Mell, P. (2007). Guide to Intrusion Detection and Preven- tion Systems (IDPS). NIST Special Publication, 800(2007), 94.

[3] Ahmed, M., Mahmood, A. N., & Hu, J. (2016). A survey of network anomaly detection techniques. Journal of Network and Computer Appli- cations, 60, 19–31.

[4] Sommer, R., & Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In 2010 IEEE Symposium on Security and Privacy (pp. 305–316). IEEE.

[5] Hall, M. A. (1999). Correlation-based feature selection for machine learning. PhD Thesis, University of Waikato.

[6] Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. Journal of Machine Learning Research, 3(Mar), 1157–1182.

[7] Wang, W., Sheng, Y., Wang, J., Zeng, X., Ye, X., & Huang, Y. (2018). HAST-IDS: Learning hierarchical spatial-temporal features using deep neu- ral networks to improve intrusion detection. IEEE Access, 6, 1792–1806.

[8] Maciá-Fernández, G., Garc´ıa-Teodoro, P., & Mirsky, Y. (2022). Establish- ing the Contaminating Effect of Metadata Feature Inclusion in Machine- Learned Network Intrusion Detection Models. In Engineering Secure Soft- ware and Systems (pp. 21–37). Springer. Retrieved from https://link. springer.com/chapter/10.1007/978-3-031-09484-2_2

[9] Yin, C., Zhu, Y., Fei, J., & He, X. (2017). A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access, 5, 21954– 21961.

[10] Ullah, I., & Mahmoud, Q. H. (2020). A hybrid model for anomaly-based intrusion detection in software-defined networks. *Journal of Network and Computer Applications*, 157, 102563.

[11] Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting sys- tem. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794).

[12] Sun, Y., Kamel, M. S., Wong, A. K., & Wang, Y. (2007). Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognition*, 40(12), 3358–3378.

[13] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, 30.

[14] Wang, C., Pang, Y., Li, Y., & Yuan, F. (2020). LightGBM: A novel en- semble boosting model for accurate soccer result prediction based on book- maker odds. *Entropy*, 22(4), 437.

[15] Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: Unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems*, 31.

[16] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.

[17] Liaw, A., & Wiener, M. (2002). Classification and regression by random- Forest. *R News*, 2(3), 18–22.

[18] Lundberg, S. M., & Lee, S. I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, 30. Molnar, C. (2020). *Interpretable Machine Learning*. Retrieved from https: //christophm.github.io/interpretable-ml-book/shapley.html

[19] DataCamp. (2023). Pandas Profiling (ydata-profiling) in Python: A Guide for Beginners. Retrieved from https://www.datacamp.com/tutorial/ pandas-profiling-ydata-profiling-in-python-guide

[20] Ross, K. D. (2021). Quantile functions. In *Probability and Statistics* (pp. 111–119). Open Educational Resource. Retrieved from https://bookdown. org/kevin_davisross/probsim-book/quantile-functions.html