

ABSTRACT

Distributed Denial of Service (DDoS) in today's world, pose a serious threat to the performance and availability of the online services. These attacks overwhelm the targeted systems with flood of traffics, making them unavailable to the legitimate users. In this report, we explore the application of Machine Learning to enhance the DDoS detection capabilities. This report explains the fundamentals and challenges of DDoS attacks present to network security.

The proposed approach involves comprehensive data preprocessing, including the removal of duplicates, normalization, handling missing values, encoding categorical variables, feature extraction and feature selection. Equitable model training is ensured by balancing the dataset through techniques like synthetic minority over-sampling technique (SMOTE). Utilizing a variety of ML algorithms such as Decision Trees, Random Forests and Support Vector Machines (SVM), alongside DL models like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), we aim to detect patterns indicative of DDoS attacks in network traffic.

TABLE OF CONTENTS

Chapter 1 Introduction	
1.1 DDoS Attack	– 4
1.2 Types of DDoS attack	– 4
1.3 Need of DDoS detection	– 4
1.4 Background	– 5
1.5 Problem statement	– 5
1.6 Objectives and scope	– 6
Chapter 2 Methodology	
2.1 Flow diagram of methodology	– 7
2.2 Data collection	– 7
2.2 Data pre-processing	– 8
2.4 Feature selection	– 9
2.5 Model selection	– 10
2.6 Implementation	– 12
2.7 Model Evaluation Metrics	– 24
Chapter 3 Experimental results and analysis	
3.1 Results	– 25
3.2 Result analysis	– 30
Chapter 4 Conclusion	– 31
Reference	– 32

INTRODUCTION

1.1 DDoS attack

A Distributed Denial of Service (DDoS) attack is a cyber-attack where it attempts to disrupt the normal traffic of a target typically a server, website or a network by overwhelming it with a flood of excessive traffic. During a DDoS attack, these requests overwhelm the target's server causing legitimate users to experience slow responses or complete outages. Detecting DDoS attack using Machine learning and deep learning involves training the models to analysing, extracting and using anomaly detection algorithms to identify deviations indicative of an attack.

DDoS attacks present several challenges, primarily due to their ability to generate large volume of malicious traffic from multiple sources making it difficult to distinguish between legitimate and harmful requests. It can also pose significant challenges due to their sophistication, sheer scale and the distributed nature of the attack traffic. The distributed nature of these attacks obstructs detection and mitigation, as the traffic often originates from a large number of compromised devices across the globe known as botnets. The high volume of traffics can quickly overwhelm the network's infrastructure leading to extended downtime, financial losses and other damages.

1.2 Types of DDoS attacks

DDoS attacks encompass various techniques designed to overwhelm a target system, rendering it inaccessible to users. Some common types include:

1. Volumetric attacks: These flood the network or server with massive amount of traffic, consuming available bandwidth and resources. Examples include UDP floods and ICMP floods.
2. Protocol attacks: exploit weaknesses in network protocols or services to exhaust server resources. SYN floods and Ping of Death are examples targeting TCP and ICMP protocols.
3. Application Layer attacks: Target web applications or services by overwhelming specific parts of the application stack such as HTTP floods or Slow Loris attacks.
4. Reflective attacks: exploit servers that respond with larger data packets than the initial request, amplifying the attack's impact. DNS amplification are examples.
5. Zero-Day attacks: Exploit previously unknown vulnerabilities in software or hardware, making them difficult to defend against until patches or mitigations are developed.

1.3 Need of DDoS detection

The need for DDoS detection in today's world is paramount due to the increasing frequency and other challenges of DDoS attacks. Effective DDoS detection is essential to accurately identify and mitigate these threats. Advanced DDoS detection mechanisms such as machine learning and deep learning are necessary to quickly and accurately identify malicious traffics

and ensuring the continuity and security of digital services. As cyber threats continue to evolve, robust DDoS detection systems are vital to safeguarding the integrity and availability of essential services, ensuring business, governments and individuals can operate securely in today's world.

1.4 Background

DDoS detection is a proactive process of identifying and mitigating DDoS attacks. The DDoS attack SDN dataset was created as a part of a research initiative at Bennett university to aid in the study and development of advanced traffic classification techniques using machine learning and deep learning algorithms. This dataset is specifically tailored for Software Defined Networking (SDN) environments and was generated using the Mininet network emulator. Mininet allows for the creation of virtual network topologies that can simulate real-world networking scenarios, providing a controlled environment for studying network behaviours under various traffic conditions.

The primary purpose of creating this dataset was to provide a rich, labelled dataset for the development and testing of machine learning and deep learning models for traffic classification in SDN environments. The dataset used includes a total of 23 features that describe various aspects of network traffic. By providing detailed statistics on both benign and malicious traffic, the dataset facilitates the understanding of traffic patterns and the identification of key indicators of DDoS attacks. The project ultimately aims to enhance the security and reliability of SDN networks by improving the accuracy and effectiveness of traffic classification and anomaly detection systems.

Use:[2] for more details about the dataset in the reference.

1.5 Problem statement

To develop advanced machine learning and deep learning-based system for real time detection and mitigation of Distributed Denial of Service (DDoS) attacks.

The goal of this project is to develop an advanced system using machine learning and deep learning techniques for real-time detection and mitigation of DDoS attacks. By leveraging comprehensive network traffic analysis and anomaly detection algorithms, the system aims to accurately differentiate between the normal and malicious traffic patterns. This approach not only enhances the ability to swiftly identify and respond to DDoS attacks but also strengthens network resilience of critical services. The project emphasizes integrating these detection capabilities seamlessly into existing infrastructures to mitigate the impact of DDoS attacks effectively and proactively safeguard digital assets.

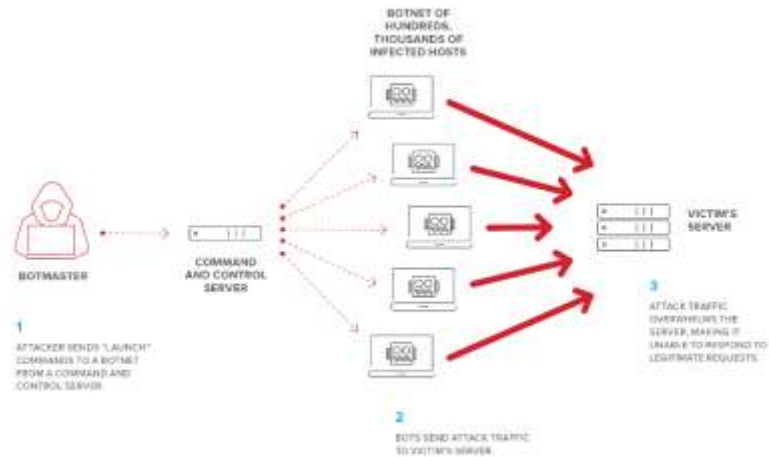


Fig.1 DDoS attack

The Fig.1 illustrates the components of a DDoS attack ecosystem: the bot master, who orchestrates the attack from a remote location; the command and control (C&C) server, which directs the compromised bots; and the botnet itself, comprising hundreds to thousands of infected computers or IoT devices. These components work in tandem to overwhelm and disrupt the services of a victim's server by flooding it with malicious traffic, illustrating the complexity and coordinated nature of DDoS attacks in cyberspace.

1.6 Objectives and scope

The objective of this project is to develop and implement machine learning and deep learning models capable of accurately distinguish between the normal traffic and malicious traffic in Software-defined Networking environments using the SDN dataset. The goal is to achieve high detection accuracy with minimal false positives and negative enabling reliable identification of DDoS attacks. Additionally, the project aims to facilitate real-time traffic analysis and anomaly detection enhancing the security and resilience of the networks.

The scope of this project encompasses the development and validation of machine learning and deep learning models specifically designed for detecting DDoS attacks. The project will involve training and testing models on the SDN dataset to achieve high accuracy in distinguishing between benign and malicious traffic. It also includes integrating these detection mechanisms into existing SDN infrastructures to ensure scalability, adaptability and effective real-time traffic monitoring and anomaly detection, while addressing challenges such as evolving attack vectors, data volume and maintaining low latency.

Methodology for DDoS detection

2.1 Flow diagram for Methodology

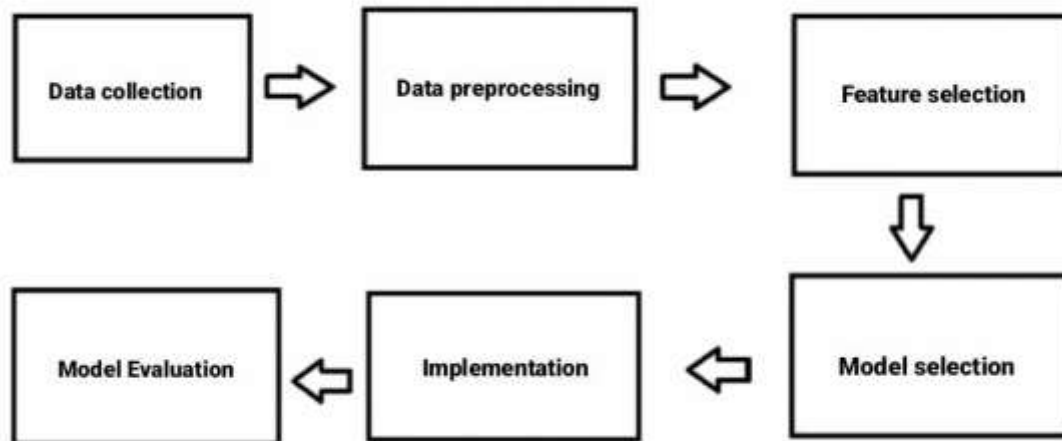


Fig.2

2.2 Data collection

Sources of Data

The data utilized in this project was derived from simulations within the Mininet emulator, a widely-used platform for emulating Software-Defined Networks (SDNs) and consists of network traffic logs collected over 250 minutes. These include benign TCP, UDP and ICMP traffic and simulated malicious activities such as TCP SYN attacks, UDP Flood attacks and ICMP attacks.

The dataset includes both benign and malicious traffic, with a total of 104345 records, each characterized by 23 features in which some are extracted from the switches such as Switch_id, Packet_count, byte_count, duration_sec, duration_nsec which is duration nanoseconds, total duration which is the sum of duration_sec and duration_nsec, Source IP, Destination IP, Port number, tx_bytes is the number of bytes transferred from the switch port and rx_bytes are the number of bytes received on the switch port. It shows the date and time which is converted into number and flow is monitored at a monitoring interval of 30 seconds. Additionally, the calculated features are Packet per flow that is packet count during single flow and byte per flow is the byte count during a single flow, Packet Rate is number of packets sent per second. These are derived to capture flow-level characteristics. Number of packets in messages, total flow entries in switch, tx_kbps, rx_kbps are data transfer and receiving rate. Last column in the dataset contains the class label indicating whether the traffic type is benign or malicious. Benign traffic has label 0 and malicious traffic has label 1.

Link: <https://www.kaggle.com/datasets/aikenkazin/ddos-sdn-dataset> download the dataset for more details.

This comprehensive approach ensured that the dataset was robust and suitable for training machine learning and deep learning models aimed at classifying network traffic into benign or malicious categories effectively.

Types of Data

Network traffic logs: Network traffic logs are generated during the simulation period within the Mininet emulator. These logs provide detailed records of network activities, encompassing various types of traffic including both benign (TCP, UDP, ICMP) and malicious (TCP SYN attacks, UDP Flood attacks, ICMP attacks) traffic.

SDN-Specific features: These features are specifically derived from switches and calculated at regular simulation intervals. These features capture specialized metrics such as packet counts, byte counts, durations, source and destination IP addresses, port number and data transfer rates tailored to software-defined networks (SDNs), providing insights into network performance and behaviour.

2.3 Data preprocessing

Data preprocessing is a fundamental step in machine learning and deep learning pipeline which ensures that the data which is used to train is clean, accurate and consistent. Preprocessing improves the quality of the data by addressing issues such as noise, missing values and inconsistencies which enhances the reliability of the models. Properly pre-processed data can lead to significant gains in the accuracy, efficiency and effectiveness of the model. Normalizing or standardizing features ensures that all input variables contribute equally to the learning process, preventing any single feature from disproportionately influencing the model due to its scale. Similarly, encoding categorical variables allows models to interpret and leverage categorical data effectively.

Handling missing values is a common preprocessing task that impacts model accuracy. Missing data can lead to biased or incomplete models if not addressed correctly. The techniques such as imputation or deleting rows or columns with missing data help in managing the issues. Another essential aspect of data preprocessing is Feature engineering which includes the creation of new features and the selection of relevant features. We can enhance the predictive power of the models by deriving the meaningful features from the raw data and selecting the most pertinent ones.

Steps in Data preprocessing

The data preprocessing steps involve several key operations aimed at preparing the dataset for model training and evaluation

1. **Loading the data:** The dataset is loaded from a CSV file into the pandas DataFrame. The dataset contains various features along with a target label.
2. **Handling missing values:** Missing values are removed from the dataset using the 'dropna()' method which ensures that only complete rows are used for further preprocessing lead to avoid errors during model training.
3. **Feature and target separation:** The independent variables are separated from the dependent variable that is label. The columns 'dt', 'src', 'dst' and 'label' are dropped

from `x` because `'dt'` might be a timestamp and `'src'` and `'dst'` might be identifiers which are not useful for the prediction tasks. The `'label'` column is the target variable and is stored separately in `'y'`.

4. Encoding categorical variables: Categorical variables are converted into numerical values using one-hot encoding that is `'pd.get_dummies()'`. This step is crucial as machine learning algorithms require numerical input.
5. Splitting the data into training and testing sets: The dataset is split into training and testing sets. This helps in evaluating the model's performance on unseen data. Here, 30% of the data is reserved for testing and 70% is used for training.
6. Reshaping data for LSTM/GRU models: For LSTM or GRU models, the data needs to be reshaped to have three dimensions. In this case, each sample is considered to have one time step.
7. Handling Class Imbalance: Class weights are computed to handle class imbalance. This ensures that the model pays equal attention to both classes during training, which is particularly important in imbalanced datasets.

These steps to create a clean and effective dataset for training machine learning and deep learning model. Additional steps in machine learning models is model training and evaluation. Various models such as Logistic Regression, Decision Making Tree, K-Nearest Neighbors, Random Forest and Gradient Boosting are trained and evaluated on the pre-processed data. Each model's performance is accessed using accuracy, confusion matrix, classification report, mean squared error (MSE), and root mean squared error (RMSE).

2.4 Feature selection

DDoS (Distributed Denial of Service) attack detection requires analysing network traffic data to identify malicious patterns. Effective feature selection helps in high performance of models in detecting these attacks. Feature selection is important for models because it directly impacts the model's performance and efficiency. By focusing on the relevant features, the model can achieve better predictive accuracy and generalization. Irrelevant or redundant features often introduce noise which can confuse the model and degrade its performance. Eliminating these unnecessary features reduces the risk of overfitting.

Feature selection related to DDoS detection

In the context of DDoS detection, feature selection is more crucial because DDoS attacks generate a large amount of network traffic data containing numerous irrelevant or redundant features. Efficient feature selection helps in identifying and focus on key indicators of DDoS attacks such as unexpected spikes in traffic volume, abnormal traffic patterns and unusual packet sizes. This approach improves the accuracy of the detection and enhances its efficiency enabling real-time detection and responses. Here are some Criteria and methods for feature selection:

1. Correlation Analysis: Correlation Analysis is a statical method used to evaluate the strength and direction of relationships between pairs of features in the dataset. High correlation indicates redundancy, where one feature may convey similar information as another leading to multicollinearity issues in models. It is a crucial step as it helps to

- find redundant features which then be removed to simplify the model. This helps to compute a correlation matrix by dropping one of each pair of highly correlated features.
2. Feature Importance from models: Models such as Random Forest or gradient Boosting that offers feature importance scores is effective for identifying the most influential features for the dataset. By training the such models on the dataset, we can extract feature importance scores, which rank features by their relative importance in the predictive task. This helps in identifying the feature that contribute most significantly to the model's performance.
 3. Principal Component Analysis (PCA): Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of a dataset while preserving its original variability. It is used to identify patterns and structure in high-dimensional data while minimizing information loss. It achieves this by transforming the data into new set of uncorrelated variables called principal components
 4. Statical tests: Statical tests helps to select the features that exhibit significant differences between the classes in a dataset. This includes techniques such as t-tests and chi-square. T-tests are used for continuous variables to assess if the means differ significantly between the groups. Chi-square tests help in evaluating the association between categorical variables and class labels. By using these techniques, features that contribute significantly to distinguish between classes are identified aiding in effective feature selection.
 5. Embedded methods: embedded methods integrate feature selection directly into the model training process by penalizing the less important features. Lasso that is also called as L1 regularization penalizes the absolute size of coefficients by pushing less influential features towards zero, thereby selecting features based on their predictive significance.

By selecting relevant features carefully, we can enhance model performance, improve interpretability, reduce overfitting and optimize computational efficiency. The criteria include minimizing redundancy, improving model performance, simplicity and computational efficiency. This ensures that the models are trained on the most informative and least redundant features, leading to better performance and reliability.

2.5 Model selection

Model selection is a critical aspect of building effective DDoS detection using machine learning and deep learning techniques. Model selection refers to process of choosing the best algorithm for a particular task. The primary goal of the model selection is to identify and deploy models that can accurately distinguish between normal network traffic and malicious DDoS attack traffic. This aims to enhance their ability to detect and mitigate these disruptive cyber threats effectively.

Model selection helps in optimizing the trade-off between the detection accuracy and computational efficiency. Model selection impacts the efficacy of defence mechanisms against these attacks. The models such as Logistic regression, Decision trees, K-Nearest neighbors (KNN), random forest, gradient boosting and deep learning architectures like GRU (Gated Recurrent Unit) handle the dynamic natures of network traffic data and classify between normal and malicious traffic accurately.

Overview of each model:

1. Logistic Regression:

Logistic regression is widely used supervised learning algorithm for binary classification tasks. It eliminates the probability of a binary outcome based on input features. This model is computationally useful, efficient, interpretable and suitable for problems with a linear decision boundary. Despite this, it can assume a linear relationship between the input features and log-odds of the output which limits its performance when dealing with complex relationships in data.

2. Decision Trees:

Decision Trees are supervised learning models used for both classification and regression tasks. They partition the feature space into segments that separate different classes or predict numerical values. Decision trees are crucial in DDoS detection as they independently segment the feature space, creating decision rules that differentiate between normal and DDoS attack patterns. Each node in tree defines a feature-based decision while the leaf nodes define outcomes such as benign traffic or DDoS attack traffic.

3. K-Nearest Neighbors:

K-Nearest Neighbors is an instance-based learning algorithm used for classification and regression tasks. It instances based on majority class among their nearest neighbors in the training set. KNN is non-parametric and does not assume any underlying distribution of the data. It is simple to implement but it requires selection of the number of k neighbors. This instance-based learning algorithm helps in classifying new traffic patterns by comparing with their nearest neighbors in the training data. However, in order to achieve high performance, it requires careful selection of neighbors and computational resources should handle large datasets effectively.

4. Random Forest:

Random Forest is an ensemble learning method that built decision trees and combines their predictions to improve the accuracy and robustness of the model. Each tree in the random forest is trained on a bootstrap sample and the nodes are split based on random subset of features. Due to its ensemble approach, this algorithm is highly effective in DDoS detection. Random forest mitigates overfitting by training each node on a bootstrap sample and enhances its ability to generalize to new unseen data. This algorithm can manage complexity and can easily identify the important features that differentiate between the normal and attack traffic which makes it suitable for high-dimensional network traffic data.

5. Gradient Boosting:

Gradient boosting is an ensemble learning technique where it combines the weak learners typically the decision tress to improve predictive performance sequentially. Each subsequent model corrects errors made by the previous models, improving the overall model performance. It is the powerful ensemble learning technique that minimises the prediction errors and improve overall accuracy. It is effective for complex tasks and handles

heterogeneous datatypes. Gradient boosting can capture complex interactions within network traffic data and be able to distinguish between normal and malicious activities.

6. Gate Recurrent Unit (GRU):

GRU is a type of recurrent neural network (RNN) architecture designed to define the vanishing gradient problem and capture long-term dependencies in sequential data. This algorithm uses gating mechanisms to control the flow of data within the network facilitating better gradient flow. This allows GRU to learn complex patterns in traffic data which helps in distinguishing between the normal potential DDoS activities. Due to this, GRU is highly effective in DDoS detection as GRU's capabilities to capture temporal dependencies accurately making it a valuable tool for real-time DDoS detection.

These models offer a range of capabilities and trade-offs in terms of complexity, interpretability and their performance. The choice of the model depends on the goals as well as specific characteristics of the dataset ensuring optimal performance.

2.6 Implementation

For our DDoS detection study, we utilized Google Colab, a cloud-based Jupyter notebook environment offering powerful computational resources, including access to high-performance GPUs and up to 12.72 GB of RAM. This setup allowed us to handle large datasets and complex models efficiently. We used an SDN dataset generated through network traffic simulation, comprising 104,345 rows and 23 columns. The dataset included both benign and malicious traffic, with features such as packet count, byte count, source and destination IPs, and traffic rates. Leveraging Google Colab's robust infrastructure facilitated the seamless execution of intensive computational tasks, enabling us to effectively train and evaluate various machine learning models for accurate DDoS detection.

The process involves utilizing Python as the primary programming language within a Google Colab environment.



Fig.3

Code details:

1. Imports and Initialization:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, mean_squared_error
import time
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
```

Fig.4

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, mean_squared_error
from sklearn.utils import class_weight
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, GRU, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

Fig.5

- Pandas and NumPy: Import for data handling and numerical operations.
- Scikit-learn('sklearn'): Various modules for machine learning tasks such as model selection and classifiers.
- Time: For measuring execution times of different model training and evaluation steps.
- Matplotlib: Used for data visualization, particularly plotting.
- Tensorflow Keras ('from tensorflow.keras.*'): provides tools for organizing layers into models, defining various types of layers ('layers.LSTM', 'layers.GRU', 'layers.Dense', 'layers.Dropout'), optimizing models ('optimizers.Adams') and implementing training control mechanisms ('callbacks.EarlyStopping').

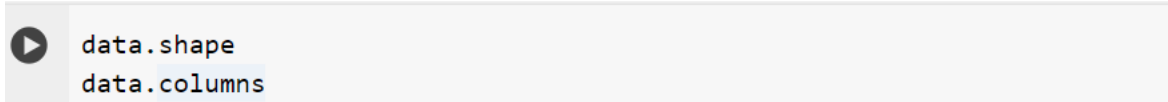
2. Data loading and preprocessing:

```
data = pd.read_csv('/content/dataset_sdn.csv')
```

Fig.6

- This part loads the dataset into a Pandas DataFrame named 'data'. This dataset is assumed to contain various features and target label indicating whether each record is benign (0) or malicious (1).

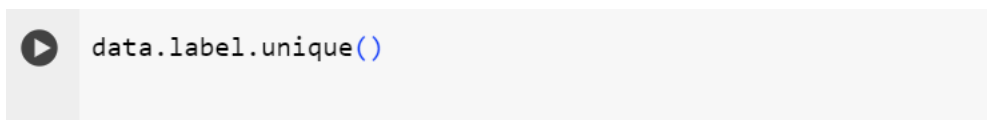
3. Exploring dataset attributes and characteristics:



```
data.shape
data.columns
```

Fig.7

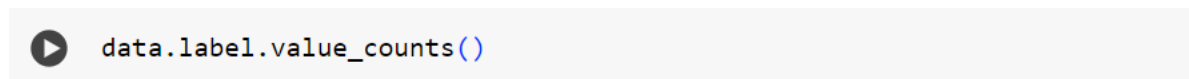
- 'data.shape' : retrieves the dimensions of the DataFrame 'data' presented as a tuple.
- 'Data.columns': accesses and returns an index object containing the names of all columns present in the DataFrame 'data', aiding in rapid identification of dataset features.



```
data.label.unique()
```

Fig.8

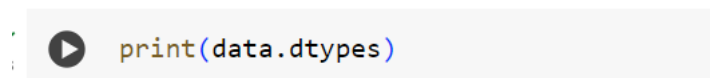
- 'Data.label.unique()': retrives an array of unique values from the 'label' column in the DataFrame 'data' aiding in identification of distinct categories or classes within the target variable.



```
data.label.value_counts()
```

Fig.9

- 'Data.label.value_counts()': counts occurrences of each unique value in the 'label' column of the DataFrame 'data' providing insights into the distribution of classes within the target variable.



```
print(data.dtypes)
```

Fig.10

- 'Print(data.dtypes)': outputs the datatypes of each column in the 'data', this is essential for data preprocessing and method selection in analysis or machine learning.

4. Pie chart for Benign and malicious requests distribution:

```

labels = ['Benign', 'Malicious']
sizes = [dict(data.label.value_counts())[0], dict(data.label.value_counts())[1]]
plt.figure(figsize = (11,6))
plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
plt.legend(['Benign', 'Malicious'])
plt.title('The percentage of Benign and Malicious Requests in dataset')
plt.show()

```

Fig.11

- Visualizes the distribution of benign and malicious requests in the dataset using a pie chart.

5. Bar plot for number requests from different IP addresses:

```

figure(figsize=(11, 7), dpi=88)
plt.barh(list(dict(data.src.value_counts()).keys()), dict(data.src.value_counts()).values(), color='blue')
plt.barh(list(dict(data[data.label == 1].src.value_counts()).keys()), dict(data[data.label == 1].src.value_counts()).values(), color='red')

for idx, val in enumerate(dict(data.src.value_counts()).values()):
    plt.text(x = val, y = idx-0.2, s = str(val), color='r', size = 13)

for idx, val in enumerate(dict(data[data.label == 1].src.value_counts()).values()):
    plt.text(x = val, y = idx-0.2, s = str(val), color='w', size = 13)

plt.xlabel('Number of Requests')
plt.ylabel('IP address of sender')
plt.legend(['All', 'malicious'])
plt.title('Number of requests from different IP address')

```

Fig.12

- Displays a horizontal bar plot illustrating the number of requests originating from different IP addresses. It distinguishes between all requests (in blue) and malicious requests (in red), with annotations showing the exact count of requests for each IP addresses.

6. Model class definition:

```
class Model:
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(self.data, self.labels, random_state=42, test_size=0.3)
```

Fig.13

- Model class: defines a python class 'Model' to encapsulate data and operations related to machine learning models.
- Initialization('__init__'): Initializes the class with data ('self.data') and labels ('self.labels'). Splits the data into training ('self.X_train', 'self.y_train') and testing ('self.X_test', 'self.y_test') sets using 'train_test_split'.

7. Logistic Regression model:

```
def LogisticRegression(self):
    solvers = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
    start_time = time.time()
    results_lr = []
    accuracy_list = []
    for solver in solvers:
        LR = LogisticRegression(C=0.03, solver=solver).fit(self.X_train, self.y_train)
        predicted_lr = LR.predict(self.X_test)
        accuracy_lr = accuracy_score(self.y_test, predicted_lr)
        results_lr.append({'solver': solver, 'accuracy': str(round(accuracy_lr * 100, 2)) + "%", 'Coefficients':
            {'W': LR.coef_, 'b': LR.intercept_}})
        accuracy_list.append(accuracy_lr)
    solver_name = solvers[accuracy_list.index(max(accuracy_list))]
    LR = LogisticRegression(C=0.03, solver=solver_name).fit(self.X_train, self.y_train)
    predicted_lr = LR.predict(self.X_test)
    accuracy_lr = accuracy_score(self.y_test, predicted_lr)
    mse_lr = mean_squared_error(self.y_test, predicted_lr)
    rmse_lr = np.sqrt(mse_lr)
```

Fig.14

```
print("#####")
print("Accuracy: %.2f%%" % (accuracy_lr * 100.0), '\n')
print("-----")
print('Best solver is : ', solver_name)
print("-----")
print(classification_report(predicted_lr, self.y_test), '\n')
print("-----")
cm = confusion_matrix(self.y_test, predicted_lr)
print("Confusion Matrix:")
print(cm)
print("-----")
print("Mean Square Error: %.2f" % mse_lr)
print("Root Mean Square Error: %.2f" % rmse_lr)
print("-----")
print("--- %s seconds --- time for LogisticRegression" % (time.time() - start_time))
print("#####")
```

Fig.15

- Logistic Regression model: The 'LogisticRegression' method within the 'Model' class trains multiple logistic regression models with different solvers, selects the best performing solver based on accuracy, evaluates the model's performance metrics(accuracy, classification report, confusion matrix, mean squared error, and root mean squared error) and prints the execution time.

8. Decision Tree model:

```
def DecisionTree(self):
    start_time = time.time()
    dtree = DecisionTreeClassifier(max_depth=5) # Set max_depth to limit the depth of the tree
    dtree.fit(self.X_train, self.y_train)
    predicted_dtree = dtree.predict(self.X_test)
    accuracy_dtree = accuracy_score(self.y_test, predicted_dtree)
    mse_dtree = mean_squared_error(self.y_test, predicted_dtree)
    rmse_dtree = np.sqrt(mse_dtree)

    print("#####")
    print("Accuracy: %.2f%%" % (accuracy_dtree * 100.0), '\n')
    print("-----")
    print("Decision Tree Classifier Results:")
    print("-----")
    print(classification_report(predicted_dtree, self.y_test), '\n')
    print("-----")
    cm = confusion_matrix(self.y_test, predicted_dtree)
    print("Confusion Matrix:")
    print(cm)
    print("-----")
    print("Mean Squared Error: %.2f" % mse_dtree, '\n')
    print("Root Mean Squared Error: %.2f" % rmse_dtree, '\n')
    print("-----")
    print("--- %s seconds --- time for Decision Tree" % (time.time() - start_time))
    print("#####")
```

Fig.16

- Decision tree model: The 'DecisionTree' method trains a decision tree classifier with a specified maximum depth, evaluates its performance metrics and prints the results.

9. K-Nearest Neighbors (KNN) model:

```
def KNN(self):
    start_time = time.time()
    knn = KNeighborsClassifier(n_neighbors=5)
    knn.fit(self.X_train, self.y_train)
    predicted_knn = knn.predict(self.X_test)
    accuracy_knn = accuracy_score(self.y_test, predicted_knn)
    mse_knn = mean_squared_error(self.y_test, predicted_knn)
    rmse_knn = np.sqrt(mse_knn)

    print("#####")
    print("Accuracy: %.2f%%" % (accuracy_knn * 100.0), '\n')
    print("-----")
    print("K-Nearest Neighbors Classifier Results:")
    print("-----")
    print(classification_report(predicted_knn, self.y_test), '\n')
    print("-----")
    cm = confusion_matrix(self.y_test, predicted_knn)
    print("Confusion Matrix:")
    print(cm)
    print("-----")
    print("Mean Squared Error: %.2f" % mse_knn, '\n')
    print("Root Mean Squared Error: %.2f" % rmse_knn, '\n')
    print("-----")
    print("--- %s seconds --- time for KNN" % (time.time() - start_time))
    print("#####")
```

Fig.17

- K-Nearest Neighbors model: The ‘KNN’ method trains a KNN classifier with a specified number of neighbors, evaluates its performance metrics and prints the results.

10. Random Forest model:

```
def RandomForest(self):
    start_time = time.time()
    rf = RandomForestClassifier(n_estimators=50, max_depth=5, min_samples_split=10, min_samples_leaf=5)
    rf.fit(self.X_train, self.y_train)
    predicted_rf = rf.predict(self.X_test)
    accuracy_rf = accuracy_score(self.y_test, predicted_rf)
    mse_rf = mean_squared_error(self.y_test, predicted_rf)
    rmse_rf = np.sqrt(mse_rf)
    print("*****")
    print("Accuracy: %.2f%%" % (accuracy_rf * 100.0), '\n')
    print("-----")
    print("Random Forest Classifier Results:")
    print("-----")
    print(classification_report(predicted_rf, self.y_test), '\n')
    print("-----")
    print("--- %s seconds --- time for Random Forest" % (time.time() - start_time))
    cm = confusion_matrix(self.y_test, predicted_rf)
    print("Confusion Matrix:")
    print(cm)
    print("-----")
    print("Mean Squared Error: %.2f" % mse_rf, '\n')
    print("Root Mean Squared Error: %.2f" % rmse_rf, '\n')
    print("-----")
    print("--- %s seconds --- time for Random Forest" % (time.time() - start_time))
    print("*****")
```

Fig.18

- Random forest model: The ‘RandomForest’ method trains a random forest classifier with specified parameters, evaluates its performance metrics and prints the results.

11. Gradient boosting model:

```
def GradientBoosting(self):
    start_time = time.time()
    gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.05, max_depth=3, random_state=42)
    gb.fit(self.X_train, self.y_train)
    predicted_gb = gb.predict(self.X_test)
    accuracy_gb = accuracy_score(self.y_test, predicted_gb)
    mse_gb = mean_squared_error(self.y_test, predicted_gb)
    rmse_gb = np.sqrt(mse_gb)

    print("*****")
    print("Gradient Boosting Classifier Results:")
    print("-----")
    print("Accuracy: %.2f%%" % (accuracy_gb * 100.0))
    print("-----")
    print("Classification Report:")
    print(classification_report(self.y_test, predicted_gb))
    print("-----")
    print("Confusion Matrix:")
    print(confusion_matrix(self.y_test, predicted_gb))
    print("-----")
    print("Mean Squared Error: %.2f" % mse_gb)
    print("Root Mean Squared Error: %.2f" % rmse_gb)
    print("-----")
    print("--- %s seconds --- time for Gradient Boosting" % (time.time() - start_time))
    print("*****")
```

Fig.19

- Gradient boosting model: the ‘GradientBoosting’ method trains a Gradient Boosting Classifier and evaluates its performance with metrics including accuracy, classification report, confusion matrix and mean squared error.

12. Data cleaning and feature engineering:

```
[10] df = data.dropna()
      x = df.drop(['dt', 'src', 'dst', 'label'], axis=1)
      y = df.label
      X = pd.get_dummies(x)
```

Fig.20

- 'df= data.dropna(): removes rows with missing values ('NaN') from 'data', storing the cleaned Dataframe in 'df'.
- 'x= df.drop(['dt', 'src', 'dst', 'label'], axis=1)': drops specified columns ('dt', 'src', 'dst', 'label') from 'df' to create 'x' which likely contains the features for modelling.
- 'y=df.label': assigns the 'label' column from 'df' to 'y' representing target variable for prediction.
- 'X= pd.get_dummies(x)': converts categorical variables in 'x' into dummy/indicator variables, potentially enhancing model performance.

13. Model Building and evaluation:

- The code defines a class 'Model' with methods for different machine learning algorithms ('LogisticRegression', 'DecisionTree', 'KNN', 'RandomForest', 'GradientBoosting', etc)
- Each method follows a similar structure:
- Initializes the model with specified parameters.
- Splits data using 'train_test_split'.
- Fits the model to the training data ('X_train', 'y_train').
- Predicts labels for the test data('X_test').
- Computes and prints accuracy metrics('accuracy_score'), error metrics('mean_squared_error', 'root_mean_squared_error') and performance reports ('classification_report', 'confusion_matrix').
- Records and prints execution time using 'time.time()'.

14. Model training and evaluation

```
➤ M = Model(X, y)
  M.LogisticRegression()
  M.DecisionTree()
  M.KNN()
  M.RandomForest()
  M.GradientBoosting()
```

Fig.21

- It is initializing model ('M') with features ('X') and labels ('y') then sequentially applying various classification algorithms to train and evaluate them on the dataset.

15. Data loading for deep learning model:

```
df = pd.read_csv('/content/dataset_sdn.csv')
```

Fig.22

- Loads the dataset into a Pandas DataFrame 'df'.

```
[17] X = df.drop('label', axis=1).values
      y = df['label'].values
```

Fig.23

- Extracts features('X') from the DataFrame by dropping the column 'label'.
- Extracts the target variable('y') from the DataFrame containing the values of the column 'label'.

16. Data preprocessing:

```
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Fig.24

- It initializes a 'LabelEncoder' object to transform categorical labels ('y') into numerical labels, encodes them into numerical values (0 and 1), initializes a 'StandardScaler' object for feature scaling and standardizes the feature matrix ('X') to have zero mean and unit variance.

17. Splitting and reshaping the data for GRU:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Fig.25

```
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))  
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
```

Fig.26

- Splits the data into training and testing sets using 'train_test_split', with 80% for training and 20% for testing. Then reshapes 'x_train' and 'X_test' into the format suitable for LSTM OR GRU input, where each sample has one time step and 'X_train.shape[1]' features.

18. Handling class imbalance:

```
class_weights = class_weight.compute_class_weight(class_weight='balanced', classes=np.unique(y_train), y=y_train)  
class_weights = {i: class_weights[i] for i in range(len(class_weights))}
```

Fig.27

- Computes class weights('class_weights') to address class imbalance using 'class_weight.compute_class_weight('balanced',np.unique(y_train),y_train)' which adjusts weights inversely proportional to class frequencies in the training data. IT converts the computes weights into a dictionary format for use in model training.

19. Model definition:

```
model = Sequential([  
    GRU(128, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True),  
    Dropout(0.3),  
    GRU(64, return_sequences=True),  
    Dropout(0.2),  
    GRU(32),  
    Dropout(0.2),  
    Dense(1, activation='sigmoid')  
])
```

Fig.28

- Define a sequential model('Sequential') with layers for a GRU-based neural network. It includes a GRU layer with 128 units('GRU(128, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=true)'), followed by drop out layers('Dropout(0.3)', 'Dropout(0.2)',etc) after each GRU layer to prevent overfitting. The model ends with a dense output layer that uses a sigmoid activation function for binary classification.

20. Model compilation:



```
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
```

Fig.29

- Configures the model for training by specifying the Adam optimizer with a learning rate of 0.001. It uses 'binary_crossentropy' as the loss function for binary classification tasks. Additionally, 'accuracy' is specified as the metric to optimize and evaluate the model during training.

21. Early stopping setup and model training:



```
[25]
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test),
                    class_weight=class_weights, callbacks=[early_stopping])
```

Fig.30

- Configures early stopping to monitor the validation loss during training. 'patience=5' specifies that training will stop after 5 epochs with no improvement in validation loss.
- 'restore_best_weights=True' ensures that the model weights are restored from the epoch with the best validation loss when training concludes.
- Trains the defined model ('model.fit(...)') on the training data (X_train,y_train'). 'epoch=50' specifies the number of training epochs, 'batch_size=32' defines the batch size used for gradient descent. 'validation_data=(X_test,y_test)' specifies the validation data used to monitor model performance during training.
- 'class_weight=class_weights' assigns class weights to handle class imbalance and 'callbacks=[early_stopping]' utilizes early stopping to prevent overfitting and improve generalization of the model.

22. Prediction and evaluation:

```

y_pred_prob = model.predict(X_test)

y_pred = (y_pred_prob > 0.5).astype(int).flatten()

```

Fig.31

- Predicts probabilities for the test data using the trained model('model.predict(_test)'). Converts the predicted probabilities into binary predictions ('y_pred') by applying a threshold of 0.5 using '(model.predict(X_test)>0.5)'. '.astype(int).flatten()' ensures the predictions are converted to integers and flattened into a one dimensional array.

23. Model evaluation and performance evaluation:

```

loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: (accuracy * 100:.2f)%')

print("Classification Report:")
print(classification_report(y_test, y_pred))

# Calculate mean squared error (MSE) and root mean squared error (RMSE)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print(f"Mean Squared Error: (mse:.4f)")
print(f"Root Mean Squared Error: (rmse:.4f)")

# Compute and print confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

```

Fig.32

- Evaluated the trained model on the test data('X_test,y_test') using 'model.evaluate(_test,y_test)'.
- Prints the test accuracy by accessing the second element ('[1]') of the returned list (which contains accuracy and loss), multiplying it by 100 to convert to percentage and formatting it into 2 decimal places(':.2f').
- Prints the classification report 'classification_report(y_test,y_pred)' to evaluate precision, recall, F1 score and other metrics.
- Computes mean squared error and root mean squared error using 'mean_squared_error(y_test,y_pred)' and 'mean_squared_error(y_test, y_pred, squared=False)' respectively.

2.7 Model Evaluation metrics

Model evaluation metrics are used to assess the performance of machine learning models

Here's a brief description of evaluation metrics:

1. **Accuracy:** Accuracy measures the proportion of correctly classified instances out of the total instances. It is suitable for balanced datasets where the classes are evenly distributed. However, it can be misleading for imbalanced datasets.
2. **Precision:** Precision measures the proportion of true positive predictions out of all positive predictions made by the model. It indicates the model's ability to avoid false positives.
3. **Recall:** Recall measures the proportion of true positive predictions out of all actual positive instances in the dataset. It indicates the model's ability to identify all positive instances.
4. **F1-score:** F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall making it useful when the classes are imbalanced.
5. **Confusion matrix:** A confusion matrix is a table that summarizes the performance of a classification report. It presents the counts of true positive, true negative, false positive, false negative predictions. From the confusion matrix, various evaluation metrics like accuracy, precision, recall and F1-score can be derived.
6. **Mean Squared Error (MSE):** MSE is the average of the squared differences between the predicted values and the actual values, indicating the average squared error of the predictions.
7. **Root Mean Squared Error (RMSE):** RMSE is the square root of the average of the squared differences between the predicted values and the actual values, providing an error measure in the same units as the target variable.

These evaluation metrics provide insights into different aspects of a model's performance and help stakeholders make informed decisions about model deployment and improvement. Choosing the appropriate evaluation metrics depends on the specific requirements and characteristics of the problem at hand.

EXPERIMENTAL RESULTS AND ANALYSIS

3.1 Results:

Pie chart for Benign and malicious requests:

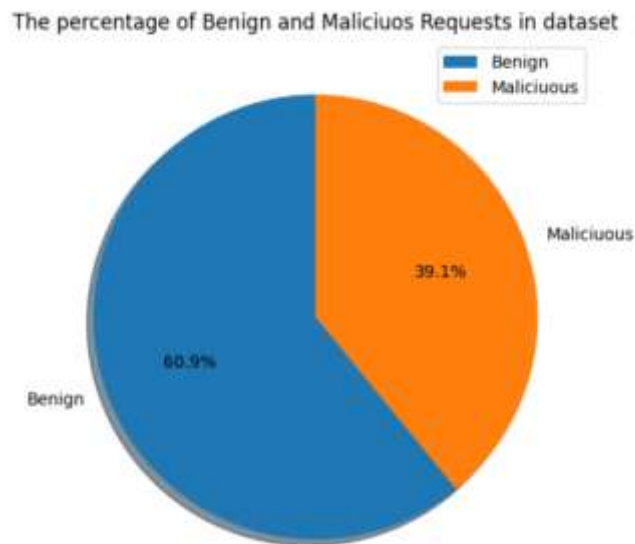


Fig.33

The pie chart visually represents the composition of requests categorized as “Benign” and “Malicious” within the dataset used for DDoS detection. The dataset used for DDoS detection analysis reveals that benign requests constitute the majority, comprising 60.9% of all requests, while malicious requests make up 39.1%. Understanding this imbalance is crucial for developing effective detection models, as it highlights the challenge of accurately identifying malicious activities amidst a larger volume of benign traffic.

Bar plot for number of requests from different IP addresses:

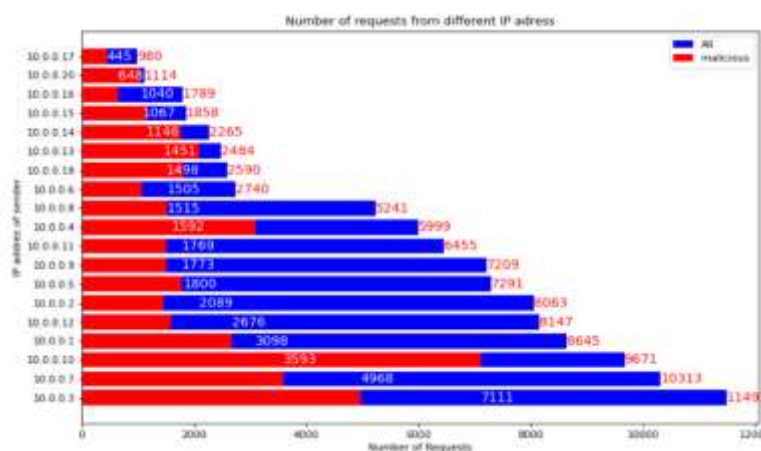


Fig.34

The bar chart illustrates the distribution of request originating from different IP addresses in the dataset, distinguishing between benign and malicious requests. The blue bars represent the total number of requests from each unique IP addresses, while the red bars specifically indicate the number of requests originating from IPs associated with malicious activity. This visualization helps in understanding the frequency and potential concentration of malicious requests compared to overall traffic. Analysing such patterns is crucial for identifying potential sources of DDoS attacks and designing effective mitigation strategies to enhance network security.

Logistic Regression Result:

```

Accuracy: 74.33%

-----
Best solver is : newton-cg
-----
              precision    recall  f1-score   support

      0               0.83       0.77       0.80       20453
      1               0.61       0.69       0.65       10699

 accuracy               0.74       0.74       0.74       31152
 macro avg              0.72       0.73       0.72       31152
 weighted avg           0.75       0.74       0.75       31152

-----
Confusion Matrix:
[[15721  3265]
 [ 4732  7434]]

-----
Mean Square Error: 0.26
Root Mean Square Error: 0.51

-----
--- 57.25685453414917 seconds --- time for LogisticRegression
-----

```

Fig.35

The Logistic regression model achieved an accuracy of 74.33% indicating its effectiveness in distinguishing between benign and malicious requests. Precision values of 83% for benign and 61% for malicious requests demonstrate its capability to correctly classify each category, while the corresponding recall rates of 77% and 69% suggest moderate performance in identifying all instances of each class. The F1-scores of 0.80 for benign and 0.65 for malicious requests reflect a balanced measure of precision and recall. The confusion matrix shows that the model correctly classified 15,721 benign requests and 7,434 malicious requests, while misclassifying 3,265 benign and 4,732 malicious requests. The mean squared error (MSE) of 0.26 and root mean squared error (RMSE) of 0.51 indicate relatively low prediction errors. Further refinement could enhance the model's ability to accurately detect malicious requests, thereby strengthening its utility in DDoS detection applications.

Decision Tree Result:

```

Accuracy: 96.38%

Decision Tree Classifier Results:
-----
precision    recall  f1-score   support

     0       0.96       0.98       0.97      18597
     1       0.97       0.94       0.95      12555

 accuracy          0.96          0.96          0.96      31152
 macro avg          0.96          0.96          0.96      31152
weighted avg          0.96          0.96          0.96      31152

-----
Confusion Matrix:
[[18228  758]
 [ 369 11797]]
-----
Mean Squared Error: 0.04
Root Mean Squared Error: 0.19

--- 0.39217591285785566 seconds --- time for Decision Tree

```

Fig.36

The Decision Making Tree Classifier exhibited exceptional performance with an accuracy of 96.38%, indicating robust capability in distinguishing between benign and malicious requests. Precision scores of 96% for benign and 97% for malicious requests demonstrate its accuracy in correctly identifying each class, while recall rates of 98% for benign and 94% for malicious requests highlight its effectiveness in capturing true positives. The F1-scores of 0.97 for benign and 0.95 for malicious requests underscore a balanced measure of precision and recall. The confusion matrix reveals that the model accurately classified 18,228 benign requests and 11,797 malicious requests, with only 758 benign and 369 malicious requests misclassified. The low mean squared error (MSE) of 0.04 and root mean squared error (RMSE) of 0.19 indicate minimal prediction errors, reaffirming the model's reliability in DDoS detection scenarios. Its rapid execution time of 0.39 seconds enhances its practicality for real-time applications.

K-Nearest Neighbors (KNN) Result:

```

Accuracy: 88.52%

K-Nearest Neighbors Classifier Results:
-----
precision    recall  f1-score   support

     0       0.93       0.89       0.91      19768
     1       0.82       0.88       0.85      11383

 accuracy          0.89          0.89          0.89      31152
 macro avg          0.87          0.88          0.88      31152
weighted avg          0.89          0.89          0.89      31152

-----
Confusion Matrix:
[[17598 1396]
 [ 2179  9987]]
-----
Mean Squared Error: 0.11
Root Mean Squared Error: 0.34

--- 15.01522421836853 seconds --- time for KNN

```

Fig.37

The K-Nearest Neighbors (KNN) classifier achieved an accuracy of 88.52% indicating strong performance in distinguishing between benign and malicious requests. Precision scores of 93% for benign and 82% for malicious requests highlight its ability to accurately classify each class, while recall rates of 89% for benign and 88% for malicious requests demonstrate its effectiveness in capturing true positives. The F1-score of 0.91 for benign and 0.85 for malicious requests underscore a balanced measure of precision and recall. The confusion matrix shows that the model correctly classified 17,590 benign requests and 9,987 malicious requests, with 1,296 benign and 2,179 malicious requests misclassified. The model exhibits a mean squared error (MSE) of 0.11 and root mean squared error (RMSE) of 0.34, indicating low prediction errors and confirming its reliability for DDoS detection tasks. Despite a longer execution time of 15.02 seconds, its accuracy and robustness make it a viable choice for application requiring precise classification with moderate computational resources

Random Forest Result:

```

Accuracy: 97.55%

-----
Random Forest Classifier Results:
-----
              precision    recall  f1-score   support

     0         0.96         1.00         0.98      18252
     1         1.00         0.94         0.97      12136

 accuracy          0.98         0.97         0.98      31152
  macro avg          0.98         0.97         0.97      31152
 weighted avg          0.98         0.98         0.98      31152

-----

--- 3.23469480405838 seconds --- time for Random Forest
Confusion Matrix:
[[18252   734]
 [    30 12136]]
-----
Mean Squared Error: 0.02
Root Mean Squared Error: 0.16

-----
--- 3.2428178787231445 seconds --- time for Random Forest

```

Fig.38

The random forest classifier achieved an impressive accuracy of 97.55% demonstrating robust performance in distinguishing between benign and malicious requests. It exhibited high precision scores of 96% for benign and 100% for malicious requests, indicating minimal false positives and excellent classification accuracy. The recall rates of 100% for benign and 94% for malicious requests underscore its ability to capture nearly all true positives. The F1-scores of 0.98 for benign and 0.97 for malicious requests reflect a harmonious balance between precision and recall. The confusion matrix reveals that the model correctly classified 18,252 benign requests and 12,136 malicious requests, with only 734 benign and 30 malicious requests misclassified. With a means squared error of 0.02 and root mean squared error with 0.16, the model demonstrates very low prediction errors, confirming its reliability for DDoS detection tasks. Despite a moderate execution time of 3.24 seconds, its exceptional accuracy and efficiency make it a highly suitable choice for real-time applications requiring precise and fast classification.

Gradient Boosting Result:

```
*****
Gradient Boosting Classifier Results:
-----
Accuracy: 98.55%
-----
Classification Report:
      precision    recall  f1-score   support

     0       1.00      0.98      0.99      18986
     1       0.97      1.00      0.98      12166

   accuracy          0.98          0.99          0.99      31152
  macro avg          0.98          0.99          0.98      31152
 weighted avg          0.99          0.99          0.99      31152

-----
Confusion Matrix:
[[18559  427]
 [   26 12140]]
-----
Mean Squared Error: 0.01
Root Mean Squared Error: 0.12
-----
--- 27.35 seconds --- time for Gradient Boosting
-----
```

Fig.39

The gradient boosting classifier delivers exceptional performance with an accuracy of 98.55%, demonstrating its efficacy in distinguishing between benign and malicious requests. Precision scores were near- perfect at 100% for benign requests and 97% for malicious requests indicating very few false positives. High recall rates of 98% for benign and 100% for malicious requests highlight the model's ability to correctly identify nearly all true positives. The F1-scores of 0.99 for benign and 0.98 for malicious requests underscore the excellent balance between precision and recall. The confusion matrix reveals that the model accurately classified 18,559 benign requests and 12,140 malicious requests, with only 427 benign requests and 26 malicious requests misclassified. With a mean squared error (MSE) of 0.01 and root means squared error (RMSE) of 0.12, the model demonstrates extremely low prediction errors, affirming its robustness for DDoS detection tasks. Despite a longer execution time of 27.35 seconds, its outstanding accuracy and minimal error rates make it a highly effective solution for real-world applications requiring precise and reliable classification.

GRU Result:

```
Test accuracy: 50.50%
Classification Report:
      precision    recall  f1-score   support

     0       0.48      0.33      0.39        96
     1       0.52      0.66      0.58       104

   accuracy          0.50          0.51          0.51       200
  macro avg          0.50          0.50          0.49       200
 weighted avg          0.50          0.51          0.49       200

Mean Squared Error: 0.4950
Root Mean Squared Error: 0.7036
Confusion Matrix:
[[32 64]
 [35 69]]
```

Fig.40

The test results indicate that the model achieved an accuracy of 50.50%, which is only slightly better than random guessing. The classification report reveals a precision of 0.48 and recall of 0.33 for class 0(Benign) and a precision of 0.52 and recall of 0.66 for class 1(malicious), indicating better performance in identifying malicious requests. The F1-score, which balances precision and recall, is 0.39 for class 0 and 0.58 for class 1, suggesting moderate overall performance. The confusion matrix shows that the model correctly classified 32 benign and 69 malicious requests, but it misclassified 64 benign as malicious and 35 malicious as benign. The mean squared error(MSE) of 0.4950 and root mean squared error (RMSE) of 0.7036 reflect substantial prediction errors, indicating room for improvement in model accuracy and robustness. Further refinement and optimization of the model are necessary to enhance its performance for more reliable DDoS detection in practical applications.

3.2 Result Analysis

<u>Model</u>	<u>Accuracy</u>	<u>Recall</u>	<u>F1-score</u>	<u>Mean Squared Error (MSE)</u>	<u>Root Mean Squared Error (RMSE)</u>	<u>Confusion Matrix</u>
Logistic Regression	74.33%	0.73	0.72	0.26	0.51	[[15721 3265] [4732 7434]]
Decision Tree	96.38%	0.96	0.96	0.04	0.19	[[18228 758] [369 11797]]
K-Nearest Neighbors(KNN)	88.52%	0.88	0.88	0.11	0.34	[[17590 1396] [2179 9987]]
Random Forest	97.55%	0.97	0.97	0.02	0.16	[[18252 734] [30 12136]]
Gradient Boosting	98.55%	0.99	0.98	0.01	0.12	[[18559 427] [26 12140]]
GRU	50.50%	0.50	0.49	0.495	0.703	[[32 64] [35 69]]

In conclusion, Gradient Boosting and Random Forest models demonstrate the highest efficacy for DDoS detection, with Gradient Boosting slightly outperforming Random Forest. Both models provide high accuracy, minimal errors and strong recall and F1-scores, making them highly reliable for practical deployment. Logistic Regression and KNN also offer reasonable performance but require further tuning. The decision Tree model, while robust, is slightly less effective than Gradient Boosting and Random Forest. The GRU model, however, falls short and is not recommended for this task in its current form.

CONCLUSION

In this study, we explored a range of machine learning and deep learning models for their efficacy in detecting Distributed Denial-of-Service (DDoS) attacks. The evaluation encompassed logistic regression, Decision Tree, K-Nearest Neighbors (KNN), Random Forest, Gradient Boosting and a GRU-based neural network. Among these, Gradient Boosting and Random Forest stood out with exceptionally high accuracies of 98.55% and 97.55%, respectively. These models demonstrated network traffic, achieving high precision and recall scores above 0.97.

Conversely, Logistic Regression, KNN and Decision Tree, while still effective exhibited slightly lower accuracies and comparatively higher mean squared errors, suggesting a higher rate of misclassification. Notably, the GRU model, designed for sequential data processing, yielded the lowest accuracy of 50.50%, indicating challenges in effectively capturing the underlying patterns of DDoS attacks with the current architecture. Moving forward, optimizing the top-performing models and possibly exploring hybrid approaches or ensemble methods could further enhance detection capabilities and robustness against evolving DDoS threats in real-world scenarios. This research underscores the importance of leveraging advanced machine learning techniques for enhancing cybersecurity measures and defending against sophisticated network attacks.

REFERENCES

- [1] R. Vishwakarma and A. K. Jain, "A survey of DDoS attacking techniques and defence mechanisms in the IoT network," *Journal of Network and Systems Management*, vol. 27, no. 3, pp. 629-655, 2019, doi: 10.1007/s11235-019-00599-z.
- [2] N. Ahuja, G. Singal, and D. Mukhopadhyay, "DDoS attack SDN Dataset," Mendeley Data, 2020. doi: 10.17632/jxpfjc64kr.1.
- [3] A. Kazin, "DDoS SDN Dataset," Kaggle, Sep. 27, 2020. Available: <https://www.kaggle.com/datasets/aikenkazin/ddos-sdn-dataset>.
- [4] M. Zekri, S. El Kafhali, N. Aboutabit, and Y. Saadi, "A Survey on Machine Learning and Deep Learning Techniques for DDoS Attack Detection," *Journal of Information Security and Applications*, vol. 65, 102828, Aug. 2021. DOI: 10.1016/j.jisa.2021.102828.
- [5] S. Sriram, R. Vinayakumar, Mamoun Alazab, and S. K. P. Soman, "Network Flow based IoT Botnet Attack Detection using Deep Learning," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2461-2470, Dec. 2020. doi: 10.1109/TNSM.2020.3044587.
- [6] J. Zhang, Y. Li, and K. Wang, "Real-Time DDoS Attack Detection Using Deep Learning Mechanisms," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 1987-1996, Jul. 2022.
- [7] S. Goyal, M. S. Gaur, and D. Sangwan, "DDoS Attack Detection and Mitigation Using Machine Learning Algorithms: A Review," *Int. J. Comput. Appl.*, vol. 185, no. 32, pp. 40-45, Apr. 2018. doi: 10.5120/ijca2018916825.