IMPLEMENTATION OF MACHINE LEARNING METHODS FOR NETWORK INTRUSION DETECTION SYSTEM.

Anita Francis Archibong
Graduate Student (CIISE)
Concordia University
Montreal, Canada
anitaarchibong2@gmail.com
Student ID: 27729790

Rahul Ravi Hulli
Graduate Student (CIISE)
Concordia University
Montreal, Canada
rahulravi.hulli@gmail.com
Student ID: 40234542

Rakshith Raj Gurupura Puttaraju
Graduate Student (CIISE)
Concordia University
Montreal, Canada
rakshithraj.gp11@gmail.com
Student ID: 40235325

Mustafa Talha Ucar
Graduate Student (CIISE)
Concordia University
Montreal, Canada
talhamucar@gmail.com
Student ID: 40059335

.

Josephine Famiyeh
Graduate Student (CIISE)
Concordia University
Montreal, Canada
Josephine.famiyeh@live.concordia.ca
Student ID: 40262544

***Abstract*— In the domain of cybersecurity, an area of paramount concern for both industrial and governmental entities, this project takes strides to confront the challenges through the application of machine learning algorithms. Our primary objective revolves around the development of a model adept at precisely recognizing malicious activities, leveraging the widely used KDD CUP'99 dataset. Through a meticulous comparison of diverse machine learning models, our primary emphasis lies in attaining optimal accuracy, with a specific focus on robust recall scores to counteract the potential evasion of detection by fraudulent cyberattacks. This research not only propels the evolution of effective intrusion detection systems but also directly responds to the urgent demand for heightened cybersecurity measures in the ever-evolving digital landscape.**

I. INTRODUCTION

THE scope transcends conventional security measures in the dynamic landscape of cybersecurity, where the paramount objective is to safeguard data against theft and loss. The overarching mission extends to ensuring the sanctity of sensitive information and preserving data integrity in the face of relentless cyber threats. As industries, businesses, and individuals increasingly rely on computer networks and services, the critical role of cybersecurity becomes increasingly evident.

The imperativeness of cybersecurity is underscored by the comprehensive array of measures it entails, aiming not only to thwart illegal access but also to prevent the unauthorized retrieval of information, thus mitigating potential harm. Intrusion Detection Systems (IDS) emerge as a cornerstone in this defence, embodying a standard practice adopted by entities ranging from corporations to government agencies and individual users. A tangible manifestation of this widespread adoption is the ubiquitous use of antivirus software, serving as a frontline defence for personal computers.

This project strategically puts in on the intricate domain of network protection within the broader expanse of cybersecurity. The focus on network protection is rooted in recognizing that computer network security is foundational for data integrity and the smooth functioning of various sectors, including industry, business, and daily life.

Please note that all the sections on this report are the same for each group member EXCEPT for the section "Results - per user".

II. BACKGROUND

At the heart of network protection lie two fundamental processes:

1. Intrusion Detection System (IDS)

An Intrusion Detection System (IDS) functions as a defensive mechanism, actively monitoring and alerting network administrators to unusual activities and potential network attacks during regular computer usage [1]. IDS is broadly categorized into Network-based Intrusion Detection System (NIDS) and Host-based Intrusion Detection System (HIDS), each serving distinct functions and monitoring content.

NIDS specializes in detecting suspicious behaviour and network threats by scrutinizing the header and content of every network traffic packet within the domain [2]. Positioned on critical nodes in the network, such as the subnet's interface interacting with the firewall, NIDS effectively guards against external network threats.

On the other hand, HIDS focuses on detecting and analysing the system configurations and application actions of computers within the domain. Unlike NIDS, HIDS is versatile, capable of installation on any network device, be it a PC or a server [3]. When HIDS identifies unusual changes like file overwrites, deletions, or port openings, it promptly alerts the administrator, making it a robust defence mechanism against internal threats.

While both NIDS and HIDS offer unique advantages, they complement rather than compete with each other, as illustrated in the table. NIDS excels in efficiently intercepting external network threats and identifying real-time network traffic. However, its effectiveness is constrained in the face of internal network assaults. HIDS, with its capability to detect attacks from the internal network by comparing historical configuration snapshots, acts as a valuable supplement. The proposed integrated detection method draws inspiration from the cooperative relationship between NIDS and HIDS.

	NIDS	HIDS
Location	Crucial Point	Any Device
Content	Network Packet	Host Event
Attack Source	External Attacks	Interal Attacks
Time	Real-time	Historical-time

With the rapid development of security technology, the market puts forward higher requirements for IDS's detection performance and defence scheme. Its operation hinges on two primary methodologies:

A. Signature Matching

A signature detection system is designed to distinguish between malicious and standard traffic patterns or application data [4]. It operates by comparing network behaviour against stored patterns or signatures of known malicious code. If a match is found, the behaviour is flagged as abnormal, allowing the system to identify attacks. However, this method has limitations, particularly when faced with new types of attacks, as it relies on having a signature or label for the attack [5]. For instance, a study [6] introduces a signature-based network Intrusion Detection System (IDS) that employs sensors to match network packets against a pre-configured set of intrusion signatures. The authors also present several design schemes for a multi-threaded network intrusion detection system.

B. Anomaly-Based Detection

An anomaly detection system plays a pivotal role by comparing captured patterns in a data stream to pre-stored baseline patterns of known threats. Action is taken based on the operational needs of the time when patterns match. This system is adept at eliminating unknown attacks, often referred to as "zero-day" attacks, by simulating the network's normal operations to detect deviations [7]. However, the trade-off with anomaly detection lies in the potential for inevitable false alarms.

Anomaly-based intrusion detection methods, based on applied techniques, can be categorized according to their use of learning, statistics, and miscellaneous techniques [8]. In a statistics-based approach, network traffic activities capture and create a profile representing the random behaviour of network traffic. This profile considers various essential features such as traffic rate, the number of packets, connections, and multiple IP addresses. The anomaly detection procedure for network traffic involves two datasets: one containing the current observed profile and the other representing a previously trained statistical profile. As network events occur, the existing profile is determined, and a quantitative value is calculated by comparing the two actions. This allows intrusion detection systems to identify anomalies based on the calculated scores. While statistics-based methods increase detection accuracy, the downside is that malicious users could exploit the system by training it and its parameters with injected malicious code.

The category of miscellaneous detection techniques includes approaches not covered by statistical and machine learning groups, employing methods such as control-flow graphs [9], finite-automata [10], and description languages [11]. For example, a study [11] introduces a framework for intrusion detection based on the Unified Modelling Language (UML), allowing software developers to specify intrusions using UML notations. Another approach [11] introduces a description language (e.g., n-grams) for intrusion detection, showing both anomaly detection and classification based on n-grams. While miscellaneous detection techniques offer flexibility and scalability, they are less effective for high-quality and high-volume data due to time-consuming data management in low-power and limited-memory environments. In recent years, machine learning algorithms have gained prominence in anomaly-based intrusion detection systems. These techniques involve building explicit or implicit models that utilize pattern analysis to classify data. Machine learning-based network intrusion systems can adapt their execution strategy as they acquire new information [12]. Various machine learning algorithms, including Support Vector Machine (SVM), Bayesian, neural network, clustering, and outlier detection, are employed in anomaly-based detection systems [12]. The main drawback, however, is the need for relatively expensive computational resources for the necessary data processing.

2. Intrusion Prevention System (IPS)

An Intrusion Prevention System (IPS) is a critical cybersecurity tool designed to identify and actively mitigate malicious activities within a network or system. Often referred to as Intrusion Detection and Prevention Systems (IDPS), IPS goes beyond the capabilities of Intrusion Detection Systems (IDS) by actively preventing or blocking detected threats. Positioned in-line, directly within the communication path between source and destination, IPS analyses network traffic and takes automated actions, such as alerting administrators, dropping dangerous packets, and halting traffic from malicious sources.
IPS operates by expanding the functionalities of Intrusion Detection Systems, providing advanced filtering behind firewalls to serve as an additional layer of defence against malicious activities. Its real-time prevention capabilities are crucial for effectively countering cybersecurity threats. To ensure optimal performance, an efficient IPS must balance the need for swift responses to potential threats with the accuracy necessary to avoid false positives. Detection methods employed by IPS include signature-based detection, utilizing identifiable exploit signatures, and statistical anomaly-based detection, which involves random sampling of network traffic against a baseline performance level.

The variety of Intrusion Prevention Systems in the market offers distinct types, such as Network-based IPS (NIPS), Wireless IPS (WIPS), Network Behaviour Analysis (NBA), and Host-based IPS (HIPS). Choosing the right IPS requires careful consideration of budget, system requirements, and thorough research into available options. While IPS plays a valuable role in detecting and preventing malicious activities, it should be viewed as part of a comprehensive cybersecurity strategy. Integrating various technologies and resources is essential to achieving comprehensive protection, covering

aspects like data protection, endpoint security, and incident response. In essence, an IPS is a pivotal component in fortifying networks and systems against a dynamic landscape of cybersecurity threats.

C. *Maintaining the Integrity of the Specifications*

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

III. Motivation

In response to the alarming surge in global cyber-attacks in recent years, there is an urgent demand for sophisticated tools capable of detecting and preventing potential intrusions within our computer network systems. According to the Canadian Centre for Cyber Security, global ransomware attacks have witnessed a staggering 15% increase in the first half of 2021 compared to the same period in 2020 [13]. Such attacks, which involve restricting access to computer systems until a ransom is paid, pose severe consequences for individuals and companies alike. The potential ramifications include substantial financial burdens, reaching millions of dollars, and the risk of data loss through breaches, which can profoundly impact a company's reputation. This escalating threat landscape underscores the critical need for tools equipped to identify and mitigate malicious intrusions across the global network system. In response to this imperative, our project adopts machine learning models and algorithms as a strategic solution to combat these evolving challenges. Within the project's defined scope, our emphasis is exclusively placed on the Intrusion Detection System (IDS), with a specific focus on the anomaly-based detection process. This deliberate choice acknowledges the limitations inherent in signature matching, such as the difficulties in maintaining an updated database and effectively addressing unknown attacks. Importantly, the Intrusion Prevention System (IPS) will not be utilized in this project.

The genesis of this project lies in the recognition of the complex challenges posed by ever-evolving cyber threats and the pressing need for adaptive systems to counteract them. Subsequent sections of this work will delve into specific methodologies and approaches employed to address these challenges, with a dedicated focus on the KDD dataset, selected classifiers, and the comprehensive evaluation of intrusion detection mechanisms. Through this exploration, our objective is to contribute meaningfully to the dynamic landscape of cybersecurity and fortify defences against intruder attacks on computer networks.

IV. Project Objectives

This section outlines the primary objectives of the project, which revolve around the development of a model proficient in precisely recognizing malicious activities. Leveraging the widely used KDD CUP'99 dataset, the project aims to conduct a meticulous comparison of diverse machine learning models. The central emphasis is on achieving optimal accuracy, with a specific focus on robust recall scores to counteract potential evasion of detection by fraudulent cyberattacks. The objectives include employing supervised machine learning classification algorithms, such as K-Nearest Neighbours, Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Support Vector Machine (SVM), for classifying different types of cyber-attacks.

V. Report Organization

The rest of the report is briefly outlined, providing a roadmap for readers to navigate through the subsequent sections. The structure encompasses an exploration of methodologies and approaches, with dedicated sections on dataset acquisition, pre-processing steps, and the implementation of machine learning algorithms. The report concludes with an extensive discussion on achieved results, insights from the confusion matrix, and the overall performance of the employed machine learning models.

VI. Main Text

The main text of this report is structured into distinct sections to provide a comprehensive overview of the project's development, execution, and outcomes.

Related work

The exploration of related works is pivotal to contextualize the current project within the existing landscape of intrusion detection systems (IDS) and cybersecurity solutions. Singh [14], Devi [15] and Abdallah [16] have conducted comprehensive reviews of various algorithms applied to the KDD CUP'99 dataset. Leveraging their insights, we aim to target specific metrics and algorithms that are proven to be effective for our study.

Divekar's article [17] in the IEEE 3rd International Conference on Computing, Communication, and Security provides detailed information about the KDD CUP'99 dataset [18]. Almseiding et al. further contribute to the literature by presenting numerical results for key algorithms, including Random Forest, K-Nearest Neighbors, Naive Bayes, and Logistic Regression, at the 15th International Symposium on Intelligent Systems and Informatics [19]. This work serves as a benchmark for our study, guiding our selection of algorithms and offering a basis for result comparisons.

In the realm of machine learning algorithms, Sabhnanin [20] and Choudahry [21] have delved into the application of Deep Neural Networks for intrusion detection. Particularly, Choudahry showcases superior recall results using deep neural networks on other intrusion attack datasets such as NSL-KDD and UNSW NB15. These findings present an intriguing comparison to the results obtained from traditional algorithms on the KDD CUP'99 dataset. This diversity in approaches highlights the evolving nature of intrusion detection methodologies and provides valuable insights for our project.

The union of these diverse studies not only aids in the selection of appropriate algorithms for our project but also

sets the stage for a nuanced evaluation of our results. The literature survey provides a foundation for understanding the strengths and limitations of various approaches, informing our methodology, and ensuring a comprehensive exploration of intrusion detection methods.

VII. Methodology

The methodology for our research is meticulously designed to assess the efficacy of various supervised machine learning classification algorithms in identifying distinct types of cyber-attacks and determining their presence. The chosen algorithms, including K-Nearest Neighbors, Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Support Vector Machine (SVM), are scrutinized in detail in the subsequent sections.

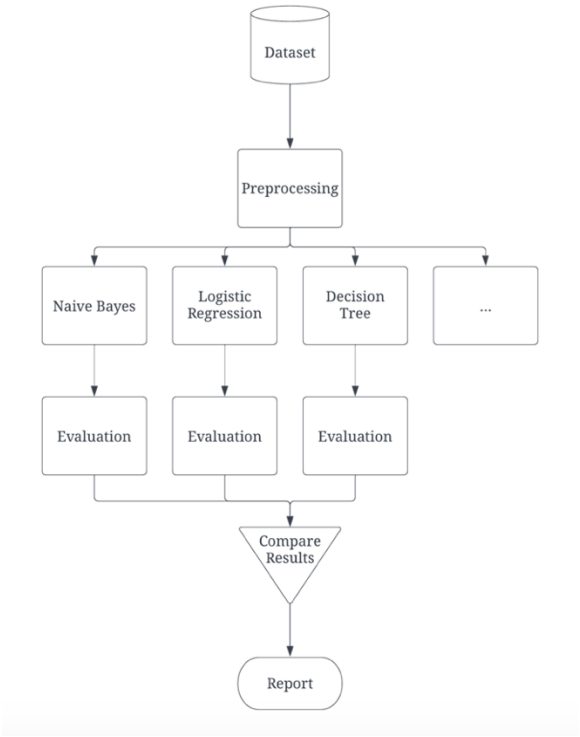


Figure 1 Showing the flowchart of our methodology.

A. *Dataset Acquisition*

Our experiments are conducted using the Knowledge Discovery and Data Mining Tools Competition (KDD Cup‘99) dataset, publicly available on Kaggle [18].

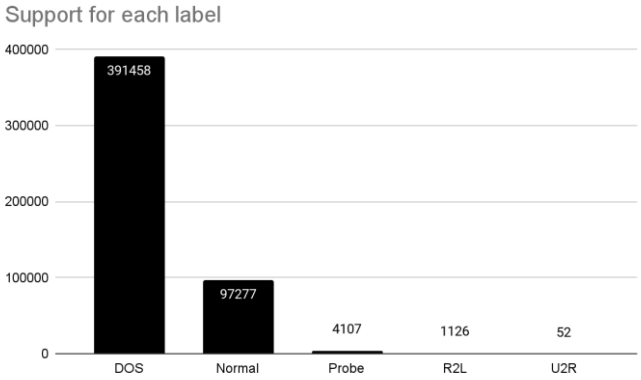
Column/Feature Name	Description of Feature
duration	Number of seconds of the connection
protocol_type	Type of Protocol used (TCP, UDP, …)
service	Service used in the remote machine (http, telnet, …)
src_bytes	Number of bytes from the source to the destination
…	…
label	Different types of cyberattacks or ‘normal’

This dataset offers a diverse range of features, such as duration, protocol type, number of bytes, login status, and the number of failed login attempts. Focused on a subset of five main classes—Denial of Service (DOS), Remote to User (R2L), User to Root (U2R), Probing, and 'normal' behavior—the dataset comprises 494,021 rows and 42 columns.

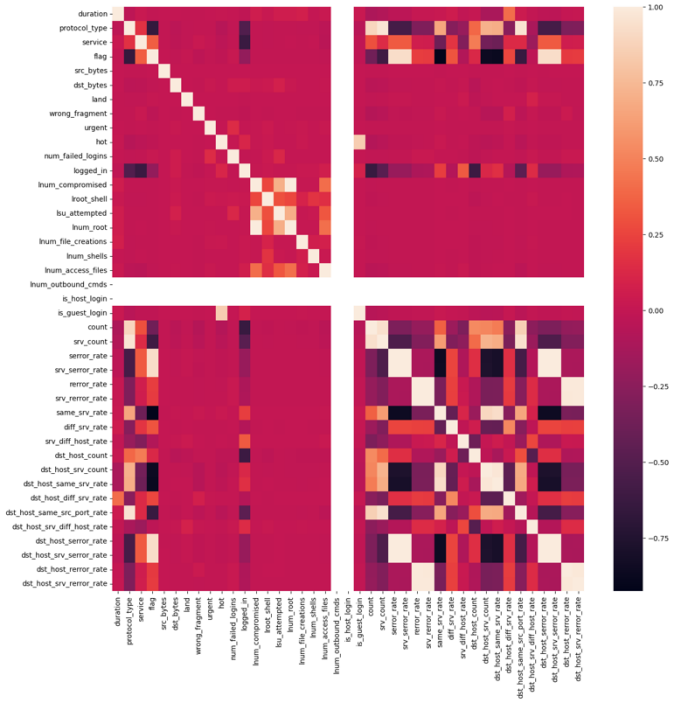
The concluding column in the dataset, labeled as 'label,' signifies the type of attack or the state of being "normal." It's noteworthy that this 'label' column encompasses a total of 22 distinct classes. Running classification algorithms on all 22 classes is impractical and ineffective. Consequently, we have refined our focus to five primary classes, four of which denote the main types of detectable cyber-attacks—namely, Denial of Service (DOS), Remote to User (R2L), User to Root (U2R), and Probing. The fifth class serves to indicate the presence or absence of a cyber-attack, characterized as 'normal' behavior. This particular column will serve as the output for our models.

B. *Preprocessing of Data*

Ensuring data quality and relevance, the preprocessing stage involves grouping the 22 original labels into five categories.



Addressing dataset imbalance, a correlation heatmap guides the selection of features, leading to dimensionality reduction by eliminating correlated columns. Additionally, three features (service, protocol type, and flag) are categorically mapped into numerical values.



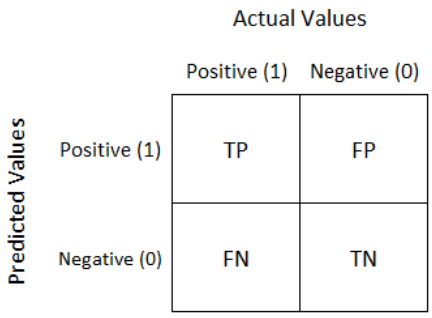
C. *Splitting Dataset into Training and Testing Data*

To prevent overfitting, we partition the data into a 75% training set and a 25% testing set using the scikit-learn library. This ensures a balance between sufficient data for model training and a robust testing set.

D. *Evaluation Metrics*

Our evaluation framework incorporates key metrics, including the Confusion Matrix, Accuracy, Precision, Recall, and Weighted F1-Score.

1. The *Confusion Matrix* visually represents prediction results, while accuracy gauges the model's overall correctness. Referring to figure below, the quantity of True Positives (TP) signifies instances where the model correctly predicted positive outcomes, and this correctness is validated. True Negatives (TN) indicate accurate negative predictions made by the model. False Positives (FP) occur when the model predicts positive outcomes incorrectly, while False Negatives (FN) represent instances where the model incorrectly predicts negative outcomes.



This confusion matrix plays a pivotal role in quantifying the subsequent evaluation metrics we will employ. It's essential to recognize that the aforementioned confusion matrix is specific to a binary classification scenario. Given that our project involves 5 distinct classes, the interpretation and calculations for TP, FP, FN, and TN become more intricate. In this context, we have leveraged the scikit-learn library for assistance. Nonetheless, a confusion matrix remains a valuable indicator for evaluating the effectiveness of a Machine Learning algorithm.

2. *Accuracy* represents the proportion of predictions that the model accurately classified. However, relying solely on accuracy as an evaluation metric proves inadequate when dealing with our class-imbalanced dataset. This inadequacy arises due to a notable imbalance in support among the classes. Consequently, it becomes essential to supplement accuracy with other evaluation metrics, such as precision and recall, which are better suited for addressing the challenges posed by class-imbalanced datasets.

3. *Precision* gauges the dependability of the algorithm, answering the question, "What proportion of positive identifications was actually correct?" In the context of our multiclass classification (with five classes), precision is computed by summing the true positives across all classes and dividing it by the sum of true positives and false positives across all classes. The mathematical formula is as follows:

$$precision = \frac{\sum_{C=1}^{5} TruePositives_c}{\sum_{C=1}^{5} (TruePositives_C + FalsePositives_C)}$$

As we have five different classes, the summation in the precision calculation will commence at (C = 1) and conclude at (C = 5). This precision metric is especially pertinent when our emphasis is on minimizing false positives.

4. *Recall* is the ratio of correctly returned data, representing the proportion of actual positives identified correctly. In the context of having more than two classes, the recall calculation is adjusted. It involves the sum of true positives (TPs) divided by the sum of true positives (TPs) and false negatives (FNs) across all classes. The mathematical formula is as follows:

$$recall = \frac{\sum_{C=1}^{5} TruePositives_c}{\sum_{C=1}^{5} (TruePositives_C + FalseNegatives_C)}$$

Considering that we have 5 different classes, the summation will commence at (C = 1) and conclude at (C = 5).

Conversely, recall becomes more pertinent when our emphasis is on minimizing false negatives, a crucial consideration in the cybersecurity context. The goal is to reduce instances where the model

erroneously predicts the absence of a cyber-attack on the network systems (i.e., 'normal' behavior) when, in reality, an attack is taking place. Accurately identifying all potentially harmful entries is paramount. Therefore, the model must exhibit strong recall, particularly for U2R, R2L, and Probe attacks (with the DOS attack being considered hazardous primarily due to its frequency).

5. *Weighted F1-Score* provides a more comprehensive insight into the model's accuracy. It is derived by computing the average of all F1 scores for each class, taking into consideration the support for each class. This averaging method ensures that the overall average considers the contribution of each class, a crucial aspect given the unbalanced nature of our dataset.

Performance Evaluation and Results

Leveraging the *classification_report* and *confusion_matrix* functions from *sklearn.metrics*, we comprehensively evaluate the developed models' performance. For the classification report, considering there are a total of 5 different reports, one for each model. A bar graph would be employed for the comparative analysis, with particular attention to the critical labels U2R and R2L due to their substantial impact if undetected in the context of network threats.

VIII. Implementation

In this section, we explore the five distinct machine learning methods implemented for a Network Intrusion Detection System (NIDS.

I. *K-Nearest Neighbors*

The k-NN algorithm is a flexible machine learning technique that may be applied to regression as well as classification. Each feature in k-NN corresponds to a dimension, and data points are represented in a multi-dimensional space. The algorithm saves all of the training data points and their labels during the training process.

How does it work?

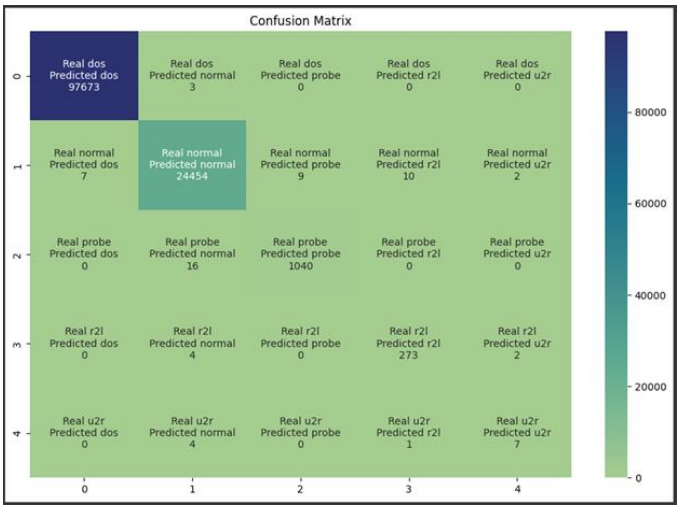
The method uses a distance metric—typically Euclidean—to find the k-nearest neighbours of a new data point in order to make predictions for classification problems. The new data point is assigned to the majority class among these neighbours. Regression analysis uses an algorithm to forecast the mean of the k-nearest neighbours' goal values. Model sensitivity is impacted by the hyperparameter k selection, which makes it vital. Prior to using k-NN, features are frequently scaled, and for large datasets, the approach may be computationally costly. Despite being straightforward, k-NN is frequently utilised in image recognition, pattern identification, and recommendation systems.

Code

```
# Data Loading
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import
StandardScaler
from sklearn.neighbors import
KNeighborsClassifier
from sklearn.metrics import
classification_report, confusion_matrix

# Load datasets
train_data =
pd.read_csv("kdd99_training_data.csv")
test_data =
pd.read_csv("kdd99_testing_data.csv")
train_labels =
pd.read_csv("kdd99_training_labels.csv")
test_labels =
pd.read_csv("kdd99_testing_labels.csv")
# Convert label columns to string type
train_labels['label'] =
train_labels['label'].astype(str)
test_labels['label'] =
test_labels['label'].astype(str)
# Convert the entire training dataset to
string type (you might want to adjust this
based on your data)
train_data = train_data.astype(str)
# Print shapes and unique labels
print("Shape of train_data:",
train_data.shape)
print("Shape of test_data:", test_data.shape)
print("Unique labels in train_labels:",
train_labels['label'].unique())

Model Building
# Normalizing the data
data_scaler = StandardScaler()# Fit and
transform the training data
X_train_scaled =
data_scaler.fit_transform(train_data)#
Transform the test data using the same scaler
X_test_scaled =
data_scaler.transform(test_data)# Training
the data using KNN
knn_classifier =
KNeighborsClassifier(n_neighbors=1)
knn_classifier.fit(X_train_scaled, y_train)
predictions =
knn_classifier.predict(X_test_scaled)

Model Evaluation
# Print results for k=1
# Build confusion matrix and print
classification report
confusion_matrix_result =
confusion_matrix(y_test, predictions)
print(classification_report(y_test,
predictions))
# Build annotations
class_labels = ['dos', 'normal', 'probe',
'r2l', 'u2r']
annotation_labels = []
for i in range(5):
    for j in range(5):
        annotation_labels.append(str(f"Real
{class_labels[i]}\nPredicted
{class_labels[j]}\n{confusion_matrix_result[i
, j]}"))
```

```
annotation_labels =
np.asarray(annotation_labels).reshape(5, 5)
# Confusion matrix visualization
plt.figure(figsize=(12, 8))
plt.title("Confusion Matrix")
sns.heatmap(confusion_matrix_result,
annot=annotation_labels, cmap='crest',
fmt='')
plt.show()
```

In conclusion, the K-Nearest Neighbors algorithm has exhibited exceptional performance with remarkable accuracy, precision, recall, and F1-score across various classes. Notably, the 'R2L' class achieved a strong 98% recall. While the model's performance in detecting 'U2R' instances with a 58% recall suggests room for improvement, the overall results underscore the efficacy of the algorithm. The following sections will further explore additional machine learning methods, providing a comprehensive evaluation of their strengths and areas for enhancement in addressing network intrusion detection challenges.



II. Random Forest

Random Forest is a powerful machine learning algorithm that combines multiple decision trees to make accurate predictions. In this guide, we will introduce Random Forest, walk you through the code implementation, and present the results of its application.

How does it work?

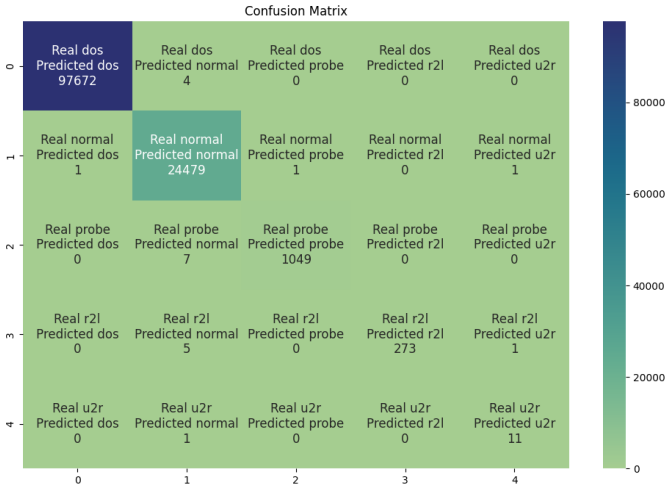
Random Forest is an ensemble learning algorithm that operates by constructing a multitude of decision trees during training and combining their outputs for robust predictions. Each decision tree in the forest is trained on a random subset of the training data, using a subset of features at each split point. This introduces diversity among the trees, reducing overfitting and enhancing generalization to new data. The algorithm employs a form of bootstrap sampling, drawing random samples with replacement from the training dataset for each tree. During the prediction phase, the ensemble of trees votes collectively for classification tasks or averages predictions for regression tasks, resulting in a more accurate and stable model. Random Forest's randomization strategies and aggregation of diverse decision trees contribute to its ability to handle high-dimensional data, mitigate overfitting, and provide reliable predictions across various types of machine learning problems.

Code

```
# Data Loading
# Loading necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import
StandardScaler
from sklearn.neighbors import
KNeighborsClassifier
from sklearn.ensemble import
RandomForestClassifier
from sklearn.metrics import
classification_report, confusion_matrix


# Loading the Dataset
# Pre-processing and dataset splitting have
previously been performed in a separate
notebook.
# Loading training and testing datasets
data_directory = "./data/"
# Load training data
x_train = pd.read_csv(data_directory +
"kdd99_training_data.csv")
# Load testing data
x_test = pd.read_csv(data_directory +
"kdd99_testing_data.csv")
# Load training labels and convert them to
strings
y_train = pd.read_csv(data_directory +
"kdd99_training_labels.csv")["label"].astype(
str)
# Load testing labels and convert them to
strings
y_test = pd.read_csv(data_directory +
"kdd99_testing_labels.csv")["label"].astype(s
tr)

# Ensure that the training features are also
treated as strings
x_train = x_train.astype(str)
# Print shape information and unique labels
in the training dataset
print(f"Shape of training dataset:
{x_train.shape}")
print(f"Shape of testing dataset:
{x_test.shape}")
print(f"Unique labels in the training
dataset: {y_train.unique()}")

# Model Building
# Training the model
# Initialize the Random Forest Classifier
with 100 estimators
random_forest_classifier =
RandomForestClassifier(n_estimators=100)
# Train the classifier on the training data
random_forest_classifier.fit(x_train,
y_train)
```

```
# Make predictions on the testing data
predictions =
random_forest_classifier.predict(x_test)

# Model Evaluation

# Evaluating the model
# Generate the confusion matrix
confusion_matrix = confusion_matrix(y_test,
predictions)

# Print the classification report
classification_report =
classification_report(y_test, predictions)

# Display the confusion matrix and
classification report
print("Confusion Matrix:")
print(confusion_matrix)
print("\nClassification Report:")
print(classification_report)

# Define the labels for the confusion matrix
annotations
y_labels = ['dos', 'normal', 'probe', 'r2l',
'u2r']
# Initialize an empty list for annotations
annotations = []

# Iterate through the label combinations and
populate the annotations list
for true_label in y_labels:
    for predicted_label in y_labels:
        i = y_labels.index(true_label)
 j = y_labels.index(predicted_label)
        count = confusion_matrix[i, j]
        annotation = f"Real
{true_label}\nPredicted
{predicted_label}\n{count}"
        annotations.append(annotation)

# Reshape the annotations into a 5x5 matrix
annotations =
np.array(annotations).reshape(5, 5)
# Visualization of the confusion matrix
# Set the figure size and title
plt.figure(figsize=(12, 8))
plt.title("Confusion Matrix")

# Create a heatmap of the confusion matrix
with annotations
sns.heatmap(confusion_matrix,
annot=annotations, cmap='crest', fmt='',
annot_kws={"fontsize": 12})
# Show the heatmap
plt.show()
```

In conclusion, the Random Forest model proved to be a highly effective machine learning algorithm, achieving remarkable overall prediction results. Across key performance metrics such as accuracy, precision (Weighted Avg), recall (Weighted Avg), and F1-score (Weighted Avg), the model consistently scored a perfect 1.00. This exceptional performance underscores the robustness of Random Forest in accurately classifying various types of cyber threats. Furthermore, the feature importance analysis provided valuable insights into the most influential features, enhancing our understanding of the model's decision-making process and contributing to its interpretability in the context of network intrusion detection.



The confusion matrix visualized above further substantiates the model's robust performance across different classes. The high accuracy and precision scores indicate that the Random Forest model is highly effective for our dataset, making it a valuable tool for predictive analysis in our domain.

III. Naïve Bayes

In the context of our project, Naive Bayes serves as a pivotal machine learning algorithm for the classification of cyber threats within network data. The essence of Naive Bayes lies in its application of Bayes' theorem, a statistical principle that calculates the probability of a certain event based on prior knowledge of conditions related to that event.

How does it work?

In simpler terms, the Naive Bayes algorithm helps us categorize network activities into different classes, such as normal behaviour or various types of cyberattacks. What makes it "naive" is the assumption that features contributing to the classification are independent, meaning the presence of one feature does not affect the presence of another. This simplifying assumption facilitates faster computations and efficient processing of large datasets, making Naive Bayes particularly well-suited for our task of intrusion detection in complex network environments.

Specifically, Gaussian Naive Bayes, a variant of the algorithm, is applied in our project. This variant assumes that the continuous features in our dataset, such as the duration of a connection or the number of bytes transmitted, follow a Gaussian (normal) distribution. This adaptation is crucial for our project, as network data often involves numerical attributes, and the Gaussian Naive Bayes allows us to model and analyze these continuous features effectively.

Throughout our project, Naive Bayes aids in predicting and classifying network activities, contributing to the overarching goal of enhancing cybersecurity by promptly identifying and mitigating potential threats within the intricate landscape of network traffic.

Code

In the ensuing sections of this report, is the practical implementation of the Naive Bayes algorithm, bringing to life the theoretical underpinnings discussed earlier. This section serves as the technical core of our study, providing a detailed account of the steps taken to integrate the algorithm into our research framework. As we navigate through the code, meticulous attention is given to Data loading, Pre-processing - executed in the earlier algorithm, Feature Scaling, Model Training, Model Evaluation and the Examining of Label Portions which are essential for the algorithm's ability to discern patterns within our dataset. The code segments unfold systematically, reflecting the algorithm's application to our specific classification task. Emphasis is placed on transparency, enabling a comprehensive understanding of how the Naive Bayes algorithm operates within the confines of our data.

```
# Data Loading
```

```
# Section 1: Importing Libraries
```

The provided code segment imports essential libraries for data analysis and visualization in Python This code segment sets up the environment for data analysis and visualization by importing NumPy for numerical operations, Matplotlib for plotting, Pandas for data manipulation, and Seaborn for statistical data visualization.

```
# Importing libraries for data analysis
and visualization
import numpy as np                    #
NumPy for numerical operations
import matplotlib.pyplot as plt       #
Matplotlib for plotting
import pandas as pd                   #
Pandas for data manipulation
import seaborn as sns                 #
Seaborn for statistical data
visualization

# Section 2: Loading Datasets
# Mount Google Drive in Google Colab
from google.colab import drive
drive.mount('/content/drive')
```

The provided code is loading training and testing datasets in a Python environment. Training and testing datasets are loaded using the Pandas library in a Google Colab environment. The training data (X_train) and testing data (X_test) are read from CSV files stored in Google Drive, and similarly, the corresponding training labels (y_train) and testing labels (y_test) are loaded from separate CSV files. The pd.read_csv function facilitates the reading of CSV files into Pandas DataFrames, allowing for efficient data manipulation and analysis. It's important to ensure that the file paths specified in the code accurately point to the locations of the training and testing datasets in Google Drive. Additionally, verifying the existence of the CSV files and the prior import of necessary libraries, such as NumPy, Matplotlib, Pandas, and Seaborn, is crucial before executing the code.

```
# loading training and testing datasets
# Load training data
X_train =
pd.read_csv("drive/MyDrive/Data/kdd99_tr
aining_data.csv")
# Load testing data
X_test =
pd.read_csv("drive/MyDrive/Data/kdd99_te
sting_data.csv")
# Load training labels
y_train =
pd.read_csv("drive/MyDrive/Data/kdd99_tr
aining_labels.csv")
# Load testing labels
y_test =
pd.read_csv("drive/MyDrive/Data/kdd99_te
sting_labels.csv")
```

In this section of the Python code, essential information about the loaded datasets is printed for verification and analysis. The print statements output the shape of the training features dataset (X_train), the shape of the testing features dataset (X_test), and the shape of the corresponding label datasets for both training (y_train) and testing (y_test). Understanding the dimensions of these datasets is crucial for subsequent data preprocessing and model training steps. The shape information provides insights into the number of instances and features in each dataset, while the label details confirm the size of the label datasets associated with the training and testing features. This serves as an initial step to ensure that the datasets are loaded correctly and have the expected dimensions before further processing.

```
# Print shape information and label
details for training and testing
datasets
print(f"Shape of training dataset:
{X_train.shape}")  # Display the shape
of the training features dataset
print(f"Shape of testing dataset:
{X_test.shape}")    # Display the shape
of the testing features dataset
print(f"Labels for training dataset:
{y_train.shape}") # Display the shape of
the training labels
print(f"Labels for testing dataset:
{y_test.shape}")   # Display the shape
of the testing labels
```

In this segment of the code, the 'label' column is extracted separately from the training (`y_train`) and testing (`y_test`) datasets. This step is performed to isolate the target variable, which represents the class labels indicating the types of cyber-attacks or 'normal' behaviour. Subsequently, the `print` statement outputs the unique labels present in the training data (`y_train`). Understanding the unique labels is essential for classification tasks, providing insights into the distinct classes the machine learning model will aim to predict. This process ensures that the label extraction is successful and that the model will be trained to recognize the specific categories within the dataset.

```
# Extracting the 'label' column from the
training (y_train)
y_train = y_train['label']
# And testing (y_test) datasets
y_test = y_test['label']

# Print the unique labels in the
training data
print(f"Labels: {y_train.unique()}")

# Scaling
```

Feature scaling is an essential preprocessing step in the realm of machine learning, and this holds true for algorithms like Naive Bayes. In the specific case of our implementation, we opt for a technique known as Min-Max Scaling.

Min-Max Scaling is a method designed to transform the features in such a way that they all fall within a predetermined range, typically [0, 1]. This normalization process is pivotal as it prevents features with larger magnitudes from exerting undue influence on the learning process. Some machine learning algorithms, including Naive Bayes, are particularly sensitive to variations in the scale of input features.

```
# Feature Scaling using Min-Max Scaling
from sklearn.preprocessing import
MinMaxScaler
# Initialize MinMaxScaler
scaler = MinMaxScaler()
# Scale the training features
X_train =scaler.fit_transform(X_train)
# Scale the testing features using the
same scaler fitted on the training data
X_test = scaler.transform(X_test)
```

Why Min-Max Scaling?

1. Preserving Independence Assumption: Naive Bayes relies on the assumption that features are conditionally independent given the class. By employing Min-Max Scaling, we uphold this assumption. The scaling process ensures that no single feature disproportionately shapes the model, maintaining the integrity of the independence assumption.

2. Addressing Varying Scales: Min-Max Scaling becomes especially valuable when dealing with features that exhibit diverse scales. By normalizing these features to a common range, we mitigate the risk of bias in the algorithm, allowing it to treat each feature more equitably during the learning process.

In essence, Min-Max Scaling contributes to a more robust and unbiased application of the Naive Bayes algorithm, enhancing its performance and generalizability across different datasets.

```
# Model Training
```

In this code snippet, a Gaussian Naive Bayes classifier is trained using the sklearn library. The process involves initializing the model, which is a Gaussian Naive Bayes classifier in this case. The model is then trained using the scaled training features (X_train) and their corresponding labels (y_train). After training, the code evaluates the performance of the model by printing the accuracy scores on both the training and testing datasets. The accuracy score indicates the proportion of correctly predicted instances. Finally, the trained model is used to make predictions on the testing dataset (X_test), and the predicted labels are stored in the y_predicted variable for further analysis or evaluation.

```
# Training a Gaussian Naive Bayes
classifier
from sklearn.naive_bayes import
GaussianNB
# Initialize the Gaussian Naive Bayes
model
model = GaussianNB()
# Train the model using the scaled
training features and corresponding
labels
model.fit(X_train, y_train)
# Evaluate the performance of the
trained model
print(f"The training score:
{model.score(X_train, y_train)}")
# Print the accuracy on the training
data
print(f"The testing score:
{model.score(X_test, y_test)}")
# Print the accuracy on the testing data
```

The performance of the trained Gaussian Naive Bayes model is evaluated. The accuracy scores on both the training and testing datasets are calculated and printed. The accuracy score represents the proportion of correctly predicted instances in comparison to the total number of instances. Subsequently, the trained model is employed to make predictions on the testing dataset (X_test), and the predicted labels are stored in the y_predicted variable. This step allows for further analysis or comparison with the actual labels to assess the model's predictive capabilities.

```
# Make predictions on the testing
dataset using the trained model
y_predicted = model.predict(X_test)

# Model Evaluation
```

In this segment of the code, the performance of the Gaussian Naive Bayes model is further assessed using a confusion matrix and a classification report. The confusion matrix provides a detailed breakdown of the model's predictions, including true positives, true negatives, false positives, and false negatives. Additionally, the classification report offers a comprehensive summary of various evaluation metrics such as precision, recall, and F1-score for each class. These metrics provide valuable insights into the model's ability to correctly identify instances belonging to different classes,

particularly relevant in the context of multi-class classification tasks.

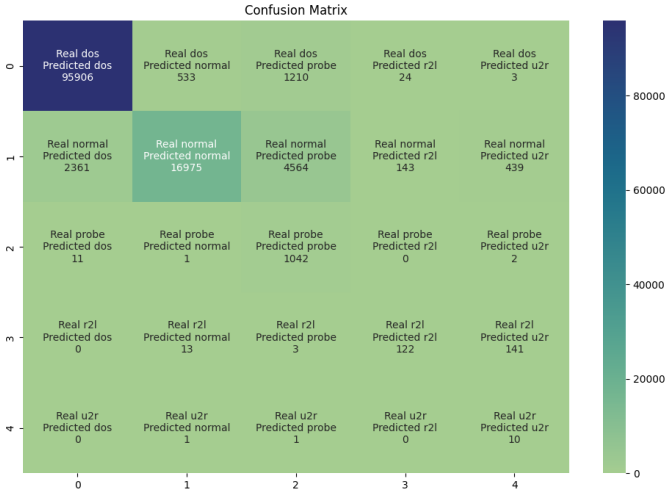
```
# Evaluate the model using confusion matrix and classification report
from sklearn.metrics import confusion_matrix, classification_report
# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_predicted)
# Generate a classification report
report = classification_report(y_test, y_predicted)
# Print the classification report
print(report)
```

This portion of the code generates annotations for a confusion matrix created during the evaluation of the Gaussian Naive Bayes model. The `y_labels` list represents the five classes: 'dos' (Denial of Service), 'normal', 'probe', 'r2l' (Remote to User), and 'u2r' (User to Root). The nested loops iterate through these classes to create annotations containing information about the real and predicted labels along with the corresponding counts in the confusion matrix.

The annotations are reshaped into a 5x5 array to align with the confusion matrix. Subsequently, a heatmap is generated using the Seaborn library, with the color map 'crest' chosen for visual appeal. The annotations, including the count values, are displayed on the heatmap, providing a comprehensive and visually intuitive representation of the model's performance across different classes. This visualization aids in understanding the model's strengths and weaknesses in classifying instances within each category.

```
# Build annotations for confusion matrix
y_labels = ['dos', 'normal', 'probe', 'r2l', 'u2r']
labels = []
# Iterate through the real and predicted labels to create annotations
for i in range(5):
    for j in range(5):
        labels.append(str(f"Real {y_labels[i]}\nPredicted {y_labels[j]}\n{conf_matrix[i, j]}"))
# Reshape the list into a 5x5 array for display
labels = np.asarray(labels).reshape(5, 5)
# Confusion matrix visualization
plt.figure(figsize=(12, 8))
plt.title("Confusion Matrix")
# Create a heatmap with annotations using Seaborn
sns.heatmap(conf_matrix, annot=labels, cmap='crest', fmt='')
# Display the plot
plt.show()
```

The confusion matrix aids in gaining a deeper understanding of how the model predicted the test values, offering insights into the performance metrics and highlighting the challenges posed by the dataset's imbalance. In the context of the Naive Bayes algorithm, focusing on the U2R label illustrates this complexity. The recall for U2R is deemed satisfactory, with only two real U2R instances being misclassified as other labels. However, a notable issue arises when observing instances where normal behaviour is incorrectly predicted as U2R. Despite 16,975 correct predictions in the normal label, 439 normal instances are mistakenly predicted as U2R. While this discrepancy is relatively small compared to the overall normal predictions, it becomes significant when considering the limited correct predictions (11) for the U2R label. This imbalance within the dataset complicates result analysis, emphasizing the importance of metrics such as recall on specific labels, like U2R and R2L, as primary decision criteria in our project.



3. Examining Label Portions

Utilizing the Pandas, Seaborn, and Matplotlib libraries to visualize the distribution of labels in the `y_test` dataset, assuming it represents the true labels for a test dataset. The labels in `y_test` are converted to a categorical series with a string type to facilitate better visualization, especially when the labels are not inherently categorical.

A countplot is generated using Seaborn to visualize the distribution of labels in the test dataset. The figure size is adjusted for clarity, and the plot is given a title. The resulting plot provides a visual representation of the frequency of each label in the test dataset.

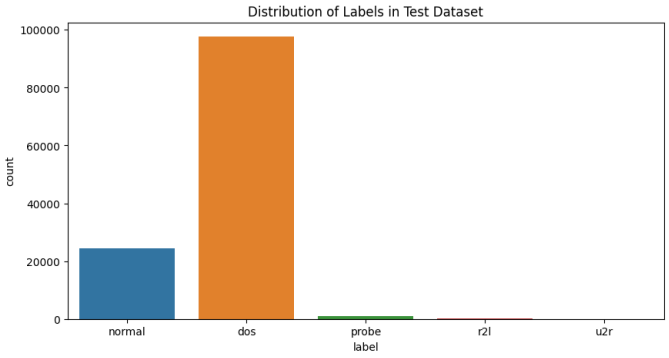
Additionally, the code calculates and prints out the relative proportions of each label using the `value_counts` method with the `normalize=True` parameter. This information offers insights into the proportion of each label category, helping to understand the overall distribution of labels in the test dataset.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
# Assuming y_test is defined and represents the true labels for a test dataset.
```

```
# Convert y_test to a categorical series with string type for better visualization.
# This is particularly useful when the labels are not inherently categorical.

y_test = pd.Series(y_test).astype(str)
# Visualize the distribution of labels in y_test using a countplot.
plt.figure(figsize=(10, 5))
# You can adjust the figure size as needed
plt.title("Distribution of Labels in Test Dataset")
sns.countplot(x=y_test)
plt.show()
# Print out the relative proportions of each label to provide insights into the label distribution.
label_proportions = y_test.value_counts(normalize=True)
print(label_proportions)
```

```
The training score: 0.9240813462342955
The testing score: 0.9234848791546901
```

The Naive Bayes model demonstrates commendable accuracy, achieving approximately 92-93% on both the training and testing datasets. This high level of accuracy indicates the model's robust ability to discern patterns and make accurate predictions.



Shifting our focus to label-specific assessments, the model excels in identifying Denial-of-Service (DOS) attacks, showcasing robust performance across all metrics. This aligns with the dataset's composition, where DOS attacks constitute a substantial portion (approximately 79.09% of all rows). However, concerning the 'Normal' label, the model, while maintaining good precision, exhibits a lower recall, indicating potential misclassifications of normal instances as attacks. To enhance accuracy in predicting normal entries, further optimization is warranted, especially considering the lower representation of 'Normal' instances (approximately 19.82% of all rows). In the 'Probe' category, the model displays high recall but low precision, suggesting a propensity to categorize other labels as probes. Achieving a balance between precision and recall is pivotal for refining the model's effectiveness in this category. Challenges emerge in accurately identifying Remote-to-Local (R2L) attacks, as reflected in low precision and recall, underscoring the imperative need for enhancements in detecting this specific attack type. Similarly, the model encounters difficulty in correctly identifying rare User-to-Root (U2R) attacks, exhibiting very low precision (0.01%) but a commendable recall (83%). Addressing this challenge is essential for augmenting efficacy against infrequent threats.

	precision	recall	f1-score	support
dos	0.98	0.98	0.98	97676
normal	0.97	0.69	0.81	24482
probe	0.15	0.99	0.26	1056
r2l	0.42	0.44	0.43	279
u2r	0.02	0.83	0.03	12
accuracy			0.92	123505
macro avg	0.51	0.79	0.50	123505
weighted avg	0.97	0.92	0.94	123505

Consideration for cybersecurity applications reveals that recall is a pivotal metric for effective network intrusion detection, especially concerning U2R, R2L, and Probe attacks, ensuring the correct identification of a majority of actual attacks. The model's application context, whether for Intrusion Prevention Systems (IPS) or Intrusion Detection Systems (IDS), significantly influences performance metrics. For IPS, striking a balance between recall on attacks and precision on normal entries is crucial. Imbalances in label support have a notable impact on precision, particularly for less frequent labels, emphasizing the need to address this imbalance for a more equitable evaluation of model performance in real-world cybersecurity scenarios.

IV. Decision Tree

Decision trees are versatile machine learning models used in AI for classification and regression tasks. The algorithm creates a hierarchical tree structure during training, where nodes represent decisions or test conditions, branches signify outcomes, and leaves represent final predictions.

How does it work?

The training process involves recursively splitting the dataset based on features to optimize separation according to a chosen criterion (e.g., Gini impurity for classification). Decision-making for new data involves traversing the tree based on feature values until reaching a leaf node, whose associated class label is the final prediction.

Code

```
#STEP-1 IMPORTING LIBRARIES AND LOADING DATASETS
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

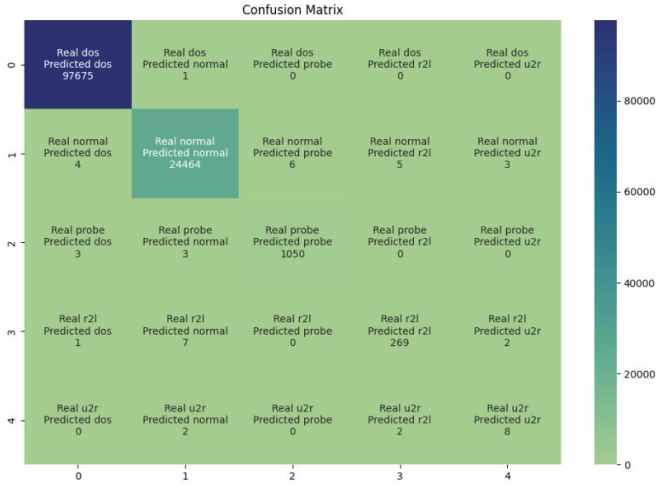
```
from sklearn.tree import
DecisionTreeClassifier
from sklearn.metrics import
confusion_matrix, classification_report
# Ensure that plots are displayed inline
in the Jupyter notebook
%matplotlib inline

# Loading datasets
X_train =
pd.read_csv("kdd99_training_data.csv")
X_test =
pd.read_csv("kdd99_testing_data.csv")
y_train =
pd.read_csv("kdd99_training_labels.csv")
y_test =
pd.read_csv("kdd99_testing_labels.csv")

#STEP-2 DATA PREPROCESSING
# Display dataset shapes
print(f"Training dataset shape:
{X_train.shape}")
print(f"Testing dataset shape:
{X_test.shape}")
# Extracting labels
y_train = y_train['label']
y_test = y_test['label']
# Display unique labels in the training
set
print(y_train.unique())

#STEP-3 TRAINING AND TESTING THE MODEL
# Initialize Decision Tree Classifier
model
model = DecisionTreeClassifier()
# Train the model
model.fit(X_train, y_train)

# Display training and testing scores
print(f"Training score:
{model.score(X_train, y_train)}")
print(f"Testing score:
{model.score(X_test, y_test)}")
# Predict labels on the test set
y_predicted = model.predict(X_test)

#STEP-4 EVALUATING THE MODEL
# Generate confusion matrix and
classification report
conf_matrix = confusion_matrix(y_test,
y_predicted)
report = classification_report(y_test,
y_predicted)
# Display classification report
print(report)
# Build annotations for confusion matrix
y_labels = ['dos', 'normal', 'probe',
'r2l', 'u2r']
labels = []
for i in range(5):
    for j in range(5):
        labels.append(f"Real
{y_labels[i]}\nPredicted
{y_labels[j]}\n{conf_matrix[i, j]}")
labels = np.asarray(labels).reshape(5,
5)

#STEP-5 VISUALIZING THE RESULTS
# Confusion matrix visualization
plt.figure(figsize=(12, 8))
plt.title("Confusion Matrix")
sns.heatmap(conf_matrix, annot=labels,
cmap='crest', fmt='')
plt.show()

# Plot distribution of true labels in
the test set
plt.figure()
plt.title("Support for each label")
sns.countplot(y_test)
plt.show()

# Display the percentage and count of
each true label in the test set
print(f"\n{y_test.value_counts() /
y_test.shape[0]}\n\n{y_test.value_counts
()}")
```

In conclusion, the Decision Tree model demonstrates exceptional performance based on the provided code outputs with outstanding training score of 0.9999973010539385 and testing score of 0.9996842233107971 indicate near-perfect accuracy in both datasets. These high scores reflect the model's robust ability to learn from the training data and make accurate predictions on unseen test data, suggesting its potential effectiveness in real-world applications. The model's performance on the testing dataset closely aligns with its training performance, highlighting its strong generalization capability.

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	97676
normal	1.00	1.00	1.00	24482
probe	0.99	0.99	0.99	1056
r2l	0.97	0.96	0.97	279
u2r	0.62	0.67	0.64	12
accuracy			1.00	123505
macro avg	0.92	0.92	0.92	123505
weighted avg	1.00	1.00	1.00	123505

In the context of the Decision Tree algorithm, focusing on the U2R label illustrates this complexity. The recall for U2R is deemed satisfactory, with only two real U2R instances being misclassified as other labels. However, a notable issue arises when observing instances where normal behaviour is incorrectly predicted as U2R. Despite 24,464 correct predictions in the normal label, 3 normal instances are mistakenly predicted as U2R. While this discrepancy is relatively small compared to the overall normal predictions, it becomes significant when considering the limited correct predictions (3) for the U2R label. This imbalance within the dataset complicates result analysis, emphasizing the

importance of metrics such as recall on specific labels, like U2R and R2L, as primary decision criteria in our project.



V. *Support Vector Machine*

Support vector machine (SVM) is a supervised machine learning technique that is used for classification and regression applications. The fundamental purpose of SVM is to find a hyperplane in an N-dimensional space (where N is the number of features) that distinguishes between different classes of data points. The approaches that are used within SVM includes:

a. Hyperplane which is a line in two dimensions that separates two classes. SVM selects the best hyperplane that best separates data points from distinct classes.
b. Support vectors which are the closest data points to the hyperplane and play an important role in setting the decision boundary. SVM maximizes the distance between the support vectors and the hyperplane.
c. Margin is the distance between the hyperplane and the nearest data point from either class.
d. Kernel Trick is a method used by SVM to efficiently handle non-linear decision boundaries by transferring the input data onto a higher-dimensional space. The kernel approach enables SVM to work implicitly in this higher-dimensional space without explicitly calculating the modified features.
e. C parameter: SVM features a regularization parameter (commonly abbreviated as C) that governs the trade-off between achieving a smooth decision boundary and properly classifying training points. A smaller C encourages a wider margin but may result in some misclassifications, whereas a greater C aims for correct classification with a tighter margin.

SVMs are widely utilized in a variety of applications, including image classification, text classification, and bioinformatics, due to their competence in handling high-dimensional data and their ability to generalize effectively to new, unseen data.

How does it work?

In machine learning, support vector machines (SVMs) are an effective tool for tasks involving regression and classification. The fundamental idea of SVM is to find the best possible hyperplane in a multidimensional space that efficiently divides data points into various classes. To do this, the method maximizes the margin, which is a measure of the separation between the closest data points from each class and the hyperplane. Determining the location and orientation of the decision boundary relies heavily on support vectors, or the data points that are closest to the hyperplane. SVMs perform very well when dealing with high-dimensional data and are particularly useful when a distinct margin of separation between classes is required.
SVM uses optimization to formally quantify the problem by balancing the goal of maximizing the margin with a penalty for incorrect classifications. This trade-off is regulated by a regularization parameter (typically represented by the letter C) introduced by the method. By implicitly mapping the input data into higher-dimensional spaces through the application of kernel functions, SVM can also handle complicated, non-linear decision boundaries. SVMs are flexible for a range of applications, including image classification, text analysis, and bioinformatics, because of their capacity to operate in higher-dimensional spaces, which allows them to capture complex correlations in the data.

To summarize, Support Vector Machines (SVMs) work by maximizing the margin between distinct classes within a dataset by identifying the ideal hyperplane. To determine the position of the hyperplane, it is important to identify data points known as support vectors. Next, a mathematical optimization problem that strikes a compromise between maximizing margin and minimizing classification mistakes must be formulated. By utilizing kernel functions, SVMs can implicitly transfer data into higher-dimensional spaces where a linear decision boundary is more obvious, enabling them to handle non-linear separations. Finding intricate linkages and patterns in the data is made easier by this procedure. SVM modifies model parameters during training, such as bias term and weights allocated to various characteristics, to produce a strong decision boundary that effectively generalizes to unknown and unseen data.

Code

Importing Libraries : This code includes four data analysis and visualization libraries and these are the NumPy, Matplotlib, Pandas, and Seaborn. NumPy handles numerical operations, Matplotlib handles graphing, Pandas handles data processing, and Seaborn does statistical data visualization. These libraries are imported into the code using the import statement. Once imported, the libraries can be used to conduct a variety of data operations.

```
# Importing libraries for data analysis
and visualization
import numpy as np                    #
NumPy for numerical operations
import matplotlib.pyplot as plt       #
Matplotlib for plotting
```

```
import pandas as pd                    #
Pandas for data manipulation
import seaborn as sns                  #
Seaborn for statistical data
visualization

# Mount Google Drive in Google Colab
from google.colab import drive
drive.mount('/content/drive')
```

Loading training and testing datasets : This code loads the SVM model's training and testing datasets. The pd.read_csv() function is used to read data from CSV files into dataframes. The features of the training and testing datasets are contained in the x_train and x_test dataframes, respectively. The labels for the training and testing datasets are stored in the y_train and y_test dataframes, respectively. Finally, the print() function is used to show the shape of the training and testing datasets, as well as the training and testing labels.

```
# loading training and testing datasets
# Load training data
X_train =
pd.read_csv("drive/MyDrive/Data/kdd99_tr
aining_data.csv")
# Load testing data
X_test =
pd.read_csv("drive/MyDrive/Data/kdd99_te
sting_data.csv")
# Load training labels
y_train =
pd.read_csv("drive/MyDrive/Data/kdd99_tr
aining_labels.csv")
# Load testing labels
y_test =
pd.read_csv("drive/MyDrive/Data/kdd99_te
sting_labels.csv")
# Print shape information and label
details for training and testing
datasets
print(f"Shape of training dataset:
{X_train.shape}")
# Display the shape of the training
features dataset
print(f"Shape of testing dataset:
{X_test.shape}")
# Display the shape of the testing
features dataset
print(f"Labels for training dataset:
{y_train.shape}")
# Display the shape of the training
labels
print(f"Labels for testing dataset:
{y_test.shape}")
# Display the shape of the testing
labels
```

Extracting the label: From the training and testing datasets, this code pulls the "label" column. The labels for the training and testing datasets are found in the respective y_train and y_test dataframes. The 'label' column is the only one that can be extracted from the data frames using the ['label'] algorithm. After being removed, y_train and y_test are given the "label" column back, accordingly. For the purpose of training and testing machine learning models, this is helpful when we wish to isolate the labels from the features in the dataset.

```
# Extracting the 'label' column from the
training (y_train)
y_train = y_train['label']
# And testing (y_test) datasets
y_test = y_test['label']

# Print the unique labels in the
training data
print(f"Labels: {y_train.unique()}")
```

Scaling: The MinMaxScaler class from the sklearn.preprocessing module is used in this code to scale the features of the training and testing datasets. The fit_transform() method is used to fit and transform the scaler to the training data, whereas the transform() method is used to apply the same scaling to the test data. The MinMaxScaler adjusts each feature independently so that it is within the training set's defined range, e.g. between zero and one. This transformation is frequently used in place of zero mean, unit variance scaling. It does not lessen the impact of outliers, but rather linearly adjusts them down into a set range, where the largest occurring data point corresponds to the maximum value and the smallest to the minimum value.

```
# scaling
from sklearn.preprocessing import
MinMaxScaler
# Imports the MinMaxScaler from scikit-
learn's preprocessing module
scaler = MinMaxScaler()

# Create an instance of the MinMaxScaler
X_train = scaler.fit_transform(X_train)
# Use the fit_transform method to scale
the features of the training set
(X_train)
X_test = scaler.transform(X_test)
# Use the transform method to apply the
same scaling to the test set (X_test)
```

Model Training: The scikit-learn module to train a Support Vector Classification (SVC) model. The code includes the SVC class from the svm module of the sklearn library. It then builds an instance of the SVC class and applies the model to the x_train and y_train training data. The Score method of the SVC class is used to calculate the model's accuracy on both training and test data. The trained model is used to predict the labels for the test set using the 'predict' method of the SVC class.

What is SVC?: SVC is a supervised machine learning technique used for classification, regression, and outlier detection. It is a form of Support Vector Machine (SVM)

technique that finds the hyperplane that best divides the data into distinct classes. The hyperplane is set so that the margin between the two classes of data points is maximized. SVC is effective in high-dimensional spaces and remains successful when the number of dimensions exceeds the number of samples. Different kernel functions can be given for the decision function, and common kernels are provided, but custom kernels can also be specified.

```
# training
from sklearn.svm import SVC
# Import the Support Vector
Classification model from scikit-learn's
svm module
model = SVC()
# Create an instance of the Support
Vector Classification model
model.fit(X_train, y_train)
print(f"Score for Training:
{model.score(X_train, y_train)}")
# Print the training score of the model
using the training data
print(f"Score for Testing:
{model.score(X_test, y_test)}")
# Print the testing score of the model
using the test data
y_predicted = model.predict(X_test)
# Predict the labels for the test set
using the trained model
```

Evaluating the model: This code imports the functions confusion_matrix and classification_report from the sklearn.metrics module of the scikit-learn library. The confusion_matrix function is then used to construct the confusion matrix for the test set using the actual and expected labels. The classification_report function produces a classification report that includes precision, recall, and F1-score. Finally, the categorization report is displayed using the 'print' function.

```
#Evaluating the Model
from sklearn.metrics import
confusion_matrix, classification_report
# Import necessary metrics from scikit-
learn
conf_matrix = confusion_matrix(y_test,
y_predicted)
# Compute the confusion matrix using the
actual and predicted labels for the test
set
report = classification_report(y_test,
y_predicted)
# Generate a classification report,
including precision, recall, and F1-
score
print(report)
from sklearn.metrics import
confusion_matrix, classification_report
# Import necessary metrics from scikit-
learn
conf_matrix = confusion_matrix(y_test,
y_predicted)
# Compute the confusion matrix using the
actual and predicted labels for the test
set
report = classification_report(y_test,
y_predicted)
# Generate a classification report,
including precision, recall, and F1-
score
print(report)
```

Building annotations and visualization of the confusion matrix: This script generates structured confusion matrix labels for easier interpretation. The class labels are defined in the code as ['dos', 'normal', 'probe', 'r2l', 'u2r'] . It then creates an empty list in which to store structured confusion matrix labels. The function goes through each actual and anticipated class combination and appends a formatted string to the labels list. The label list is transformed into a NumPy array and molded into a 5x5 matrix 1. A confusion matrix is visualized where Plt.figure(figsize=(12, 8)) specifies the size of the figure, while plt.title("Confusion Matrix") specifies the plot's title for the confusion matrix . The sns.heatmap(conf_matrix, annot=labels, cmap='crest,' fmt='') command generates a heatmap of the confusion matrix with label annotations.

```
# building annotations
y_labels = ['dos', 'normal', 'probe',
'r2l', 'u2r']
# Define the class labels for better
interpretation
labels = []
# Initialize an empty list to store
formatted confusion matrix labels
for i in range(5):
# Loop through each combination of
actual and predicted classes
  for j in range(5):
    labels.append(str(f"Real
{y_labels[i]}\nPredicted
{y_labels[j]}\n{conf_matrix[i, j]}"))
# Append a formatted string to the
labels list
labels = np.asarray(labels).reshape(5,
5)

# Convert the list of labels to a NumPy
array and reshape it to a 5x5 matrix
# visualization of confusion matrix
plt.figure(figsize=(12, 8))
# Set the size of the figure
plt.title("Confusion Matrix")
# Set the title of the plot
sns.heatmap(conf_matrix, annot=labels,
cmap='crest', fmt='')
# Create a heatmap of the confusion
matrix with annotations using labels
plt.show()
# Display the plot
```

Reminder on label proportions: The pandas, seaborn, and matplotlib libraries are used in this Python code to examine and display the label distribution in the y_test variable, which most likely corresponds to the target labels in a machine learning classification problem. The labels are handled as strings thanks to the data preparation process. The code then uses seaborn to build a countplot that shows the number of times each unique label appears in y_test, giving label support a clear visual representation. It also determines and prints each label's proportions, providing information on the relative occurrence of various classes. This kind of study can be helpful in evaluating the effectiveness and any biases in a classification model, as well as being essential for comprehending the class distribution.

```
# Reminder on label proportions:

# Import necessary libraries for data
manipulation and visualization
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

y_test = pd.Series(y_test).astype(str)
# Convert y_test to a pandas Series and
ensure its values are treated as strings
plt.figure(figsize=(8,6))
# Set the size of the figure
plt.title("Label Support")
# Set the title of the plot
sns.countplot(x=y_test, )
# Create a countplot to visualize the
distribution of labels in y_test
plt.show()
# Display the plot
# Calculate and print the proportions of
each label in y_test
label_proportions =
y_test.value_counts(normalize=True)
print(label_proportions)
```

Conclusion

The support vector machine model performed well and achieved an overall accuracy of 100% however, on both the training and testing datasets with a training result of 99.88% and a test result of 99.87%. For both the 'dos' and 'normal' classes, the model obtained flawless precision, recall, and F1-score. However, the Recall for the "probe" class was marginally lower, at 96% suggesting potential areas for improvement in detecting this type of intrusion. The 'R2L' class scored 83%, while the 'U2R' class scored 75%. An unbalanced representation is seen in the testing dataset's class distribution, which is typical of IDS scenarios in which the number of normal instances greatly exceeds that of intrusions.

In the majority of classes, the model seems to function remarkably well, attaining great accuracy and precision. The results' interpretation may be impacted by the unequal distribution of classes, particularly in the case of minority classes. Additional analysis could shed more light on the behaviour of the model and identify possible areas for improvement. Examples of this analysis include feature importance, confusion matrix evaluation, and correcting class imbalance. The diagram and a picture below display a true depiction of the labels.

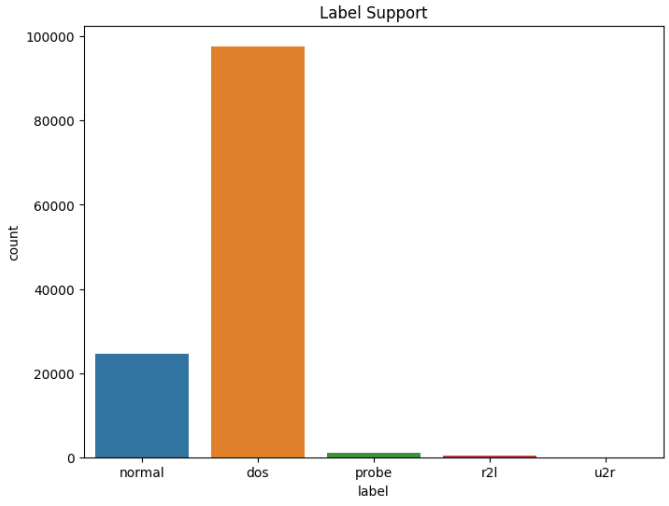
Shape of training dataset: (370515, 31)
Shape of testing dataset: (123505, 31)
Labels for training dataset: (370515, 1)
Labels for testing dataset: (123505, 1)

Score for Training: 0.9988475500317127
Score for Testing: 0.9987045058904498

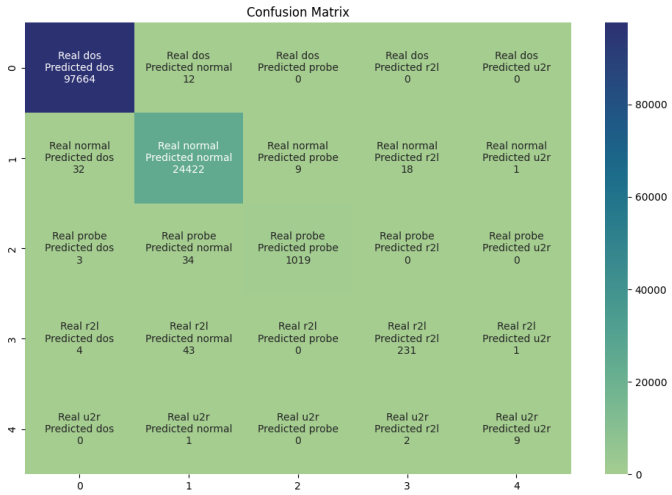
Output

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	97676
normal	1.00	1.00	1.00	24482
probe	0.99	0.96	0.98	1056
r2l	0.92	0.83	0.87	279
u2r	0.82	0.75	0.78	12

Accuracy			1.00	123505
Macro avg	0.95	0.91	0.93	123505
Weighted Avg	1.00	1.00	1.00	123505



Overall, the false positives and false negatives detected throughout the use of the SVM model are displayed below in the confusion matrix



IX. Results – Per User

We decided to gather some more information on the proposed algorithms to help make better comparisons and look for downsides. The snippets of the code doing such comparisons are as below:

I. K Nearest Neighbor Algorithm

Confusion matrix in text format was as below

```
# Print results for k=1
# Build confusion matrix and print classification report
confusion_matrix_result = confusion_matrix(test_labels['label'], predictions)
# Change y_test to test_labels['label']
print(classification_report(test_labels['label'], predictions))
```

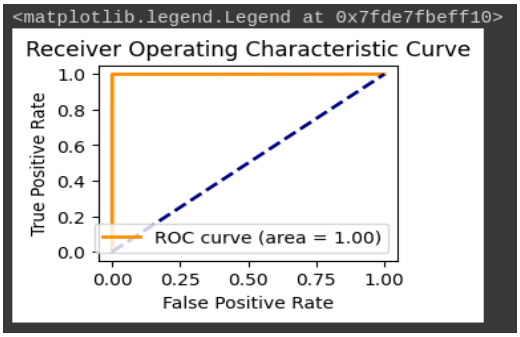
The output was as below

	precision	recall	F1-score	support
dos	1.00	1.00	1.00	97676
normal	1.00	1.00	1.00	24482
probe	0.99	0.98	0.99	1056
r2l	0.96	0.98	0.97	279
u2r	0.64	0.58	0.61	12
accuracy			1.00	123505
Macro avg	0.92	0.91	0.91	123505
Weighted avg	1.00	1.00	1.00	123505

The ROC Curve was plotted, the code and graph is as below:

```
# Convert labels to binary format
binary_test_labels = label_binarize(test_labels['label'], classes=['dos', 'normal', 'probe', 'r2l', 'u2r'])
binary_predictions = label_binarize(predictions, classes=['dos', 'normal', 'probe', 'r2l', 'u2r'])

# ROC Curve
fpr, tpr, _ = roc_curve(binary_test_labels.ravel(), binary_predictions.ravel())
roc_auc = auc(fpr, tpr)

# Display ROC Curve
plt.subplot(2, 2, 2)
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Curve')
plt.legend(loc='lower right')
```

Output:



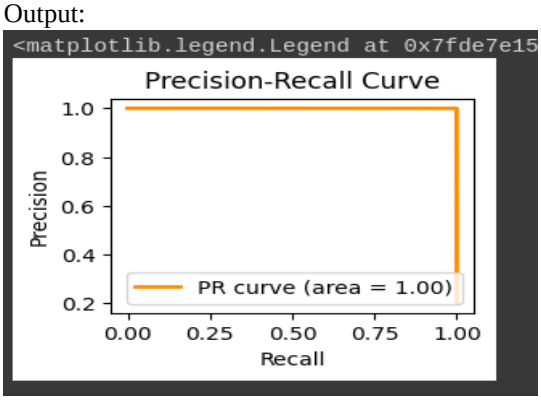
There are no unusual dips in performance against both axis, which means this algorithm performs as intended.

The PRC Curve was plotted, the code and graph is as below

```
# Convert labels to binary format
binary_test_labels = label_binarize(test_labels['label'], classes=['dos', 'normal', 'probe', 'r2l', 'u2r'])
binary_predictions = label_binarize(predictions, classes=['dos', 'normal', 'probe', 'r2l', 'u2r'])

# Precision-Recall Curve
precision, recall, _ = precision_recall_curve(binary_test_labels.ravel(), binary_predictions.ravel())
pr_auc = auc(recall, precision)

# Display Precision-Recall Curve
plt.subplot(2, 2, 3)
plt.plot(recall, precision, color='darkorange', lw=2, label='PR curve (area = {:.2f})'.format(pr_auc))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
```

plt.legend(loc='lower right')

Output:



There are no unusual dips in performance against both axis, which means this algorithm performs as intended.

The accuracy, precision, recall, f1 and Matthews Corelation Coefficient were calculated, and their code and values are as below

```
# Additional Evaluation Metrics
accuracy = accuracy_score(test_labels['label'], predictions)
precision = precision_score(test_labels['label'], predictions,
average='weighted')
recall = recall_score(test_labels['label'], predictions,
average='weighted')
f1 = f1_score(test_labels['label'], predictions,
average='weighted')
mcc = matthews_corrcoef(test_labels['label'], predictions)

# Display results
print(f"\nResults for K-Nearest Neighbors:")
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
print(f"Matthews Correlation Coefficient: {mcc}")
```

Output:

Results for K-Nearest Neighbors:
Accuracy: 0.999530383385288
Precision: 0.999528514247723
Recall: 0.999530383385288
F1 Score: 0.9995290701535438
Matthews Correlation Coefficient: 0.9985987315946491

This information would mean that K Nearest Neighbour algorithm displays almost perfect levels of accuracy, precision, Recall, F1 Score and Matthews Correlation Coefficient.

Improvisation:
The best improvisation technique that can work for K-Nearest Neighbour algorithm is the Voting Classifier. At points where this algorithm is not perfect, through voting between the other protocols, a different algorithm can be chosen which, under the same instance, showcases better results.

Using the right performance indicators is essential for evaluating the efficacy of ensemble models. Metrics like recall, accuracy, precision, and F1-score shed light on many facets of a model's operation. Selecting models that complement one another is made possible by analyzing each model individually inside the ensemble and contrasting their performance measures. By strategically combining the benefits of various models, the flaws of one model are counterbalanced, leading to an overall more robust prediction capability.

Optimizing hyperparameters is yet another crucial component in creating successful ensemble models. The performance of any individual model in the ensemble may be influenced by particular hyperparameters. By adjusting these hyperparameters, you can make sure that every model runs at peak efficiency and gives the ensemble its greatest performance. Finding the best-performing ensemble for a given problem area requires experimenting with various model and hyperparameter combinations. To attain the best possible prediction accuracy and generalization, the ensemble's configuration is adjusted through an iterative process.

An algorithm to employ this would be as follows:

Steps:

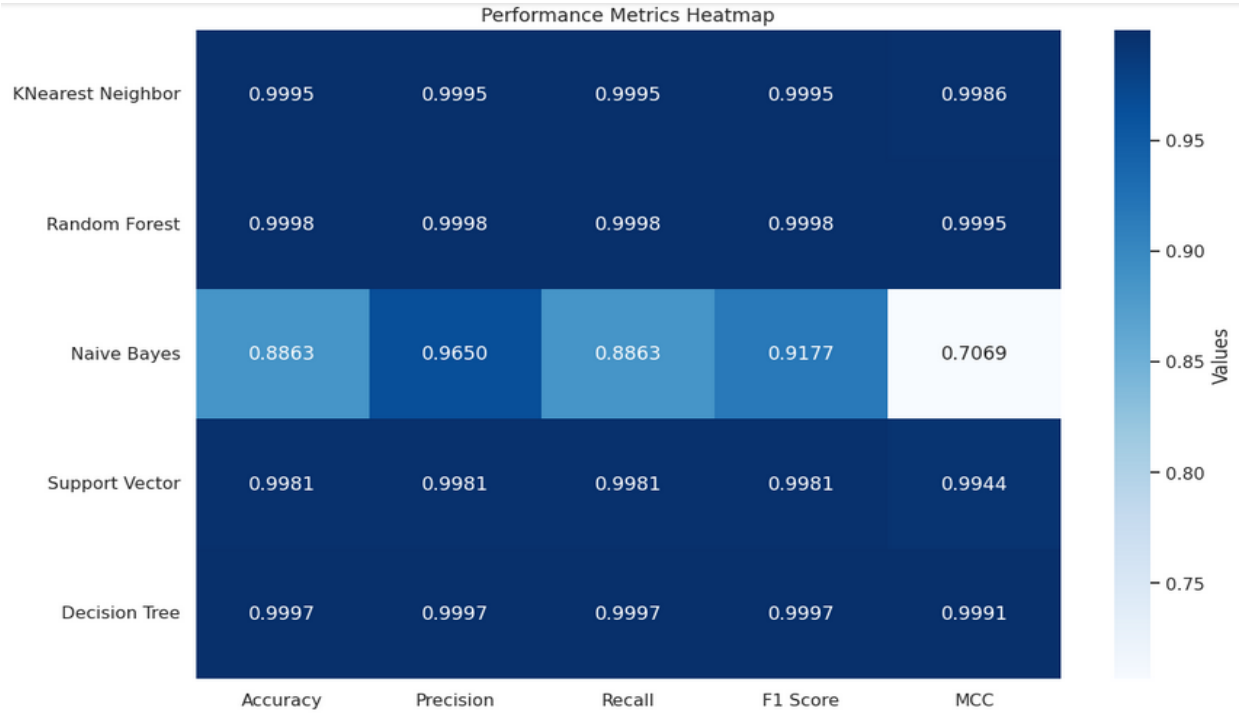
1. Split the dataset into training and test sets using the train_test_split function:
   - X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
2. Create individual models:
   - knn_model = KNeighborsClassifier(n_neighbors=5)
   - tree_model = DecisionTreeClassifier(random_state=42)
   - svm_model = SVC(probability=True, random_state=42)
3. Create a Voting Classifier:
   - voting_classifier = VotingClassifier(

estimators=[('knn', knn_model), ('tree', tree_model), ('svm', svm_model)],
voting='soft' # Use 'hard' for hard voting
)

4. Train the ensemble model:
   - voting_classifier.fit(X_train, y_train)
5. Make predictions on the test set:
   - predictions = voting_classifier.predict(X_test)
6. Evaluate the performance of the ensemble model:
   - accuracy = accuracy_score(y_test, predictions)
   - Print the accuracy of the ensemble model:
   - print(f'Ensemble Accuracy: {accuracy}')

II. Comparision:

A heatmap was calculated for the values, and the graph is as below.



The KNearest Neighbour algorithm performed remarkably well, achieving 99.95% F1 score, accuracy, precision, and recall. Additionally noteworthy is the high Matthews Correlation Coefficient (MCC) of 99.86% for KNearest Neighbour, which shows a significant correlation between the actual and projected categories.

With an F1 score of 99.98% and impressive accuracy, precision, and recall, the Random Forest algorithm proved to be an excellent choice. At 99.95%, Random Forest's Matthews Correlation Coefficient (MCC) is remarkably high, indicating a strong correlation between the actual classifications and the predictions.

Conversely, the Naive Bayes algorithm demonstrated strong performance with a comparatively lower accuracy of 88.63%. It did, however, show good accuracy (96.50%) and a respectable F1 score of 91.77%. Naive Bayes's Matthews Correlation Coefficient (MCC) is 70.69%, which suggests a moderate correlation.

Excellent results were obtained using the Support Vector Machine (SVM), which had an F1 score of 99.81% and high accuracy, precision, and recall. SVM's Matthews Correlation Coefficient (MCC) is 99.44%, indicating a high degree of agreement between the actual and projected classifications.

At last, the Decision Tree algorithm demonstrated remarkable performance, achieving an F1 score of 99.97% with excellent accuracy, precision, and recall. A strong correlation between the anticipated and actual classifications is highlighted by the Decision Tree's extremely high Matthews Correlation Coefficient (MCC), which stands at 99.91%.

X. CONTRIBUTION

Please note that the below table talks about the work distribution of every member to this project.

Name	**Contribution**
Anita Francis Archibong	Naïve Bayes Algorithm implementation and Improvement
Rahul Ravi Hulli	K Nearest Neighbor Algorithm Implementation and Improvement
Rakshith Raj Gurupura Puttaraju	Random Forest Algorithm Implementation and Improvement
Josephine Famiyeh	Support Vector Machine Algorithm Implementation and Improvement

Mustafa Talha Ucar	Decision Tree Algorithm Implementation and Improvement

This copy of the report solely deals with my contribution to this project. I worked on **K Nearest Neighbor Algorithm,** basically implemented a working model of the algorithm, added a few more parameters, which will be useful for comparison, and provided a comparison for the same. I also provided a suggestion for improvement of this method and provided an algorithm of the improvement. My codes for the same will be included in the submission.

Also note that these responsibilities also include creating the project report where everyone has contributed equally – which denotes that everyone has had equal amounts of responsibilities handed out. Their responsibilities will be provided just like how I've provided mine here.

If in case of any discrepancies, we welcome any queries at any of our email addresses mentioned above.

XI. Conclusion

We have reviewed the implementation sections for the five supervised machine learning classification algorithms: K-Nearest Neighbors, Random Forest, Naïve Bayes, Decision Tree, and Support Vector Machine (SVM), as used with the KDD CUP'99 dataset in your report. Here are the conclusions and suggestions for improvement for each section:

1. K-Nearest Neighbors (K-NN)

Performance: Exhibited high accuracy, precision, recall, and F1-score across various classes, with strong performance particularly in the 'R2L' class (98% recall).

Improvement: Performance in detecting 'U2R' instances can be improved (58% recall). Considering hyperparameter tuning (e.g., varying the number of neighbors 'k') and feature selection or engineering might enhance this.

2. Random Forest

Performance: Achieved remarkable overall prediction results with perfect scores in accuracy, precision, recall, and F1-score. Demonstrated robustness in accurately classifying various types of cyber threats.

Improvement: While the results are exceptional, exploring feature importance analysis in more detail could provide deeper insights. Additionally, investigating model performance on a more balanced dataset could be beneficial.

3. Naïve Bayes

Performance: Presented good accuracy (92-93%) but faced challenges in classifying 'U2R' and 'R2L' attacks accurately.

Improvement: Considering alternative feature scaling methods or feature selection might help. Additionally, exploring the impact of different Naïve Bayes variants (e.g., Bernoulli or Multinomial) could also be useful.

4. Decision Tree

Performance: Demonstrated near-perfect accuracy in training and testing, suggesting strong learning and generalization capabilities.

Improvement: To counter overfitting, which is common in decision trees, methods like pruning or employing ensemble methods like Random Forest could be explored. Also, examining the tree depth and feature importance can provide further insights for improvement.

5. Support Vector Machine (SVM)

Performance: Achieved perfect scores in key metrics but showed lower recall in the 'U2R' category.

Improvement: Experimenting with different kernels (e.g., RBF, polynomial) and tuning hyperparameters like C and gamma could improve performance. Considering class weights to address class imbalance might also be beneficial.

Overall Conclusions and Recommendations:
Results Comparison: Random Forest and SVM showed the most promising results in terms of overall metrics. However, all models struggled to some extent with the 'U2R' category, indicating a common area for improvement.

XII. Discussion

Enhancing Results:

- Making sure that all classes—including minority ones—have enough representation in the dataset is essential to addressing class imbalances. To establish a more balanced training set, this could entail undersampling the dominant class and oversampling the minority class.
- The practice of fine-tuning a model's configuration settings to maximize performance is known as hyperparameter tuning. This is enhanced by cross-validation, which divides the dataset systematically, trains the model on distinct subsets, and assesses the model's performance in a range of configurations.
- To enhance the model's capacity to identify patterns in the data, feature engineering entails the creation or modification of features. For improved predictive performance, this may entail creating new features, altering current ones, or choosing the most pertinent ones.
- By combining the advantages of several models, ensemble approaches improve overall prediction

accuracy. Methods like bagging, boosting, and stacking integrate predictions from many models to improve performance and mitigate the shortcomings of each model alone.

- Improved Solution: By merging these algorithms in an ensemble manner, performance may be improved overall. A voting classifier or a stacking classifier that incorporates the output of several models as an input to a final model could be used in this situation.

A. Group Techniques

Machine learning approaches known as ensemble methods combine several models to create a more resilient and powerful system. The key premise is that you may typically obtain higher performance than any single model alone by combining the predictions from numerous models. There are two main categories of ensemble methods that may apply in this situation:

a. The Voting Classifier Concept selects the class with the highest number of votes as the final prediction. Each model in the ensemble "votes" for a class.

Implementation: You may use your five models (K-NN, Random Forest, Naïve Bayes, Decision Tree, SVM) to vote on the output in a voting classifier. Either "hard" voting (based on the most common class) or "soft" voting (based on probability estimates, if available) may be used.

Benefit: When there is diversity among the models and they each have unique strengths and weaknesses, this approach can be quite successful. It's an attempt to profit on the "wisdom of the crowd."

b. Stacking Classifier Idea: To combine the predictions of many models, a new model is trained using a process called stacking. The original models make up the first level, and a new model trained using the first-level models' outputs makes up the second level.

Implementation: One machine learning model (such as a logistic regression) might learn to combine the predictions of the K-NN, Random Forest, Naïve Bayes, Decision Tree, and SVM by using them as inputs.

Advantage: Because stacking can learn the optimal way to combine the predictions instead of depending merely on a basic rule like majority voting, it can be more effective than simple voting.

B. Why This Might Be Better

Complementary Strengths: The advantages and disadvantages of various models vary. Because they might complement one another, you can frequently get higher performance by mixing them. For example, a model may perform better in each set of conditions whereas another model may be better at processing a particular sort of data.

Decreased Overfitting: Since the final prediction is based on several models rather than primarily on the peculiarities of a single model, ensemble approaches can also aid in the reduction of overfitting.

Better Generalization: In general, ensemble approaches are thought to enhance the model's ability to generalize to new data, producing predictions that are more dependable and sturdier.

Using the KDD CUP'99 dataset, you may be able to develop a more reliable and accurate network intrusion detection system by implementing these ensemble techniques.

REFERENCES

[1] Fadi Salo, Mohammadnoor Injadat, Ali Bou Nassif, Abdallah Shami, and Aleksander Essex. Data mining techniques in intrusion detection systems: A systematic literature review. IEEE Access, 6:56046–56058, 2018.

[2] Jimmy Shun and Heidar A. Malki. Network intrusion detection system using neu- ral networks. In 2008 Fourth International Conference on Natural Computation, volume 5, pages 242–246, 2008.

[3] Shijoe Jose, D. Malathi, Bharath Reddy, and Dorathi Jayaseeli. A survey on anomaly-based host intrusion detection system. Journal of Physics: Conference Series, 1000:012049, apr 2018.

[4] Handong Wu, Stephen Schwab, and Robert Lom Peckham. Signature based network intrusion detection system and method, September 9 2008. US Patent 7,424,744.

[5] Bruno Bogaz Zarpela˜o, Rodrigo Sanches Miani, Cla´udio Toshio Kawakani, and Sean Carlisto de Alvarenga. A survey of intrusion detection in internet of things. Journal of Network and Computer Applications, 84:25–37, 2017.

[6] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In International Workshop on Information Security Applications, pages 188–203. Springer, 2004.

[7] Bibudhendu Pati, Chhabi Rani Panigrahi, Rajkumar Buyya, and Kuan-Ching Li. Ad- vanced Computing and Intelligent Engineering: Proceedings of ICACIE 2018, Volume 1. Springer Nature. Google-Books-ID: cHfRDwAAQBAJ.

[8] Mahbod Tavallaee, Natalia Stakhanova, and Ali Akbar Ghorbani. Toward credible evaluation of anomaly-based intrusion-detection methods. IEEE Transactions on Systems,

Man, and Cybernetics, Part C (Applications and Reviews), 40(5):516–524, 2010.

[9] David Wagner and R Dean. Intrusion detection via static analysis. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001, pages 156–168. IEEE, 2000.

[10] Joel Branch, Alan Bivens, Chi Yu Chan, Taek Kyeun Lee, and Boleslaw K Szyman- ski. Denial of service intrusion detection using time dependent deterministic finite automata. In Proc. Graduate Research Conference, pages 45–51, 2002.

[11] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In Proceedings of the 2013 ACM workshop on Artificial intelligence and security, pages 67–76, 2013.

[12] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Macia´-Ferna´ndez, and Enrique Va´zquez. Anomaly-based network intrusion detection: Techniques, systems and chal- lenges. computers & security, 28(1-2):18–28, 2009.

[13] Canadian Centre for Cyber Security », Canadian Centre for Cybersecurity, 26 November 2023. https://cyber.gc.ca/en/guidance/cyber-threat-bulletin-ransomware-threat-2021

[14] P. Singh et A. Tiwari, « A Review Intrusion Detection System using KDD'99 Dataset, International Journal of Engineering Research, vol. 3, nº 11, p. 6.6

[15] Canadian Centre for Cyber Security », Canadian Centre for Cybersecurity, 26 November 2023. https://cyber.gc.ca/en/guidance/cyber-threat-bulletin-ransomware-threat-2021

[16] E. E. Abdallah, W. Eleisah, et A. F. Otoom, Intrusion Detection Systems using Supervised Machine Learning Techniques: A survey », Procedia Computer Science, vol. 201, p. 205-212, janv. 2022, doi: 10.1016/j.procs.2022.03.029.

[17] A. Divekar, M. Parekh, V. Savla, R. Mishra, et M. Shirole, « Benchmarking datasets for Anomaly-based Network Intrusion Detection: KDD CUP 99 alternatives », in 2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS), oct. 2018, p. 1-8. doi: 10.1109/CCCS.2018.8586840.

[18] KDD99 dataset | Kaggle ». https://www.kaggle.com/datasets/toobajamal/kdd99-dataset

[19] M. Almseidin, M. Alzubi, S. Kovacs, et M. Alkasassbeh, Evaluation of machine learning algorithms for intrusion detection system », in 2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY), Subotica, Serbia, sept. 2017, p. 000277-000282. doi: 10.1109/SISY.2017.8080566.

[20] M. Sabhnani et G. Serpen, Application of Machine Learning Algorithms to KDD Intrusion Detection Dataset within Misuse Detection Context., janv. 2017, p. 209-215.

[21] S. Choudhary et N. Kesswani, « Analysis of KDD-Cup'99, NSL-KDD and UNSW-NB15 Datasets using Deep Learning in IoT, Procedia Computer Science, vol. 167, p. 1561-1573, 2020, doi: 10.1016/j.procs.2020.03.367.

APPENDIX

For the research and implementations, the following software, hardware, and dependent libraries were utilized:

Software	***Platforms Utilized for Implementation***
Google Colab	The primary platform used for coding, running Jupyter Notebooks, and leveraging GPU resources for machine learning tasks.
Python	The primary programming language for implementing machine learning algorithms and conducting data analysis.
Scikit-learn	A machine learning library in Python, providing efficient tools for data analysis and modelling.
Pandas	A data manipulation and analysis library for Python.
NumPy	A library for numerical operations in Python.
Matplotlib	A plotting library for creating visualizations in Python.
Seaborn	A statistical data visualization library based on Matplotlib.

Hardware	***Platforms Utilized for Implementation***
Google Colab	The research and implementations were conducted on Google Colab, which provides cloud-based resources, including GPU acceleration.

***Note*:** The source code for the project will be submitted separately on Moodle.